DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: Butas Rafael-Dorian
GROUP: 30424

CONTENTS

1. Assignment Objective

a. Main Objective:

Main Objective: To design and implement a queues management application in Java that minimizes client waiting time.

b. Sub-Objectives:

Sub-Objective	***Section***
Define functional requirements	*Problem Analysis, Modeling, Scenarios, Use Cases*
Create use cases with diagrams and descriptions	*Problem Analysis, Modeling, Scenarios, Use Cases*
Design object-oriented programming structure	*Design*
Create UML package design and class diagram design	*Design*
Choose the best fitting data structures	*Design*
Implement the Task class used for client object manipulation	*Implementation*
Implement the Server class used for queue object manipulation and for defining the running of the server threads	*Implementation*
Implement the logic for the shortest queue strategy	*Implementation*
Implement the logic for the shortest time strategy	*Implementation*
Implement the logic required for the proper running of the time thread in the Time Manager Class	*Implementation*
Implement the Scheduler class which starts the server threads and creates the server objects, dispatches tasks to them, assigns the wanted strategy, etc.	*Implementation*
Implement the Simulation Manager class which creates the time object and starts the time thread, creates the scheduler, creates the random tasks, computes the data required at the end of the simulation(average waiting.service time, peak hour), deals with the running of the main thread of the application while updating the data every second in a text file and in the graphical interface.	*Implementation*
Implement the TextFile class which updates the text file every second with information about the clients that are not in queues yet but also about the clients in every queue. After the final second, the end data will be also printed.	*Implementation*
Implement the user-friendly graphical interface that is splitted in two: the interface for the simulation inputs and the interface for the real time queue evolution.	*Implementation*
Draw conclusions and propose future developments	*Conclusions*

2. Problem Analysis, Modeling, Scenarios, Use Cases

The functional requirements of the queue management application describe the core features and functionalities that define its behavior and capabilities. These requirements outline what the application should do to meet the needs of its users. Here's an overview of the functional requirements (in order) for the project:

a. Functional Requirements:

1. Setup Simulation:
- The simulation application should allow users to input values for the number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time.
- The application should provide a "validate input data" button for users to confirm their inputs.
- Upon clicking the button, the application should validate the input data.
- If the data is valid, the application should display a message instructing the user to start the simulation.
- If the data is invalid, the application should display an error message and prompt the user to input valid values.

2. Start Simulation:
- The simulation application should allow users to initiate the simulation after setting up the parameters.
- Upon starting the simulation, the application should execute the simulation processes, including client arrival, queue management, service, and departure.
- The application should display real-time updates of the queue evolution during the simulation process.

3. Display Real-time Queue Evolution:
- The simulation application should provide a real-time display of the queues' evolution during the simulation.
- Users should be able to observe changes in the queues' statuses, such as client arrivals, queue lengths, and service completions.

b. Non-Functional Requirements:

1. User-Friendly Interface:
- The simulation application should be designed with an intuitive and user-friendly interface.
- Users should find it easy to navigate through the setup process and start the simulation without encountering difficulties.
- Clear instructions and prompts should be provided to guide users through each step of the simulation setup.

2. Performance:
- The application should be responsive and perform efficiently, even when handling a large number of clients and queues.
- Simulations should execute in a timely manner, providing quick results to the user.
- The system should be able to handle real-time updates of queue evolution without significant delays or lagging.

3. Reliability:
- The simulation application should be reliable, ensuring accurate representation of queueing dynamics and simulation results.
- It should handle various scenarios gracefully, such as invalid input data, and provide appropriate error handling mechanisms.
- Users should have confidence in the consistency and accuracy of the simulation outputs.

By incorporating these functional and non-functional requirements, the simulation application aims to provide users with a reliable experience in setting up and executing queue simulations.

c. Use Cases for Queue Simulation Application:

1. Setup Simulation:
- Success Scenario:
1. User inserts values for the number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time.
2. User clicks on the "Start Simulation" button.
3. Application validates the data.

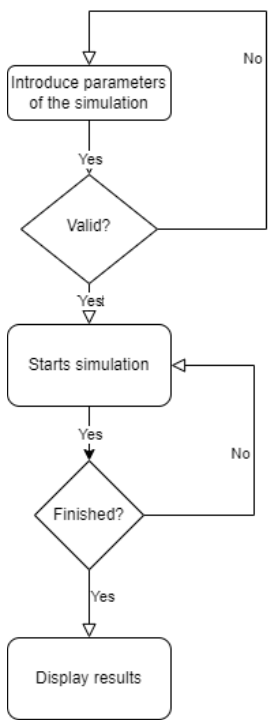
- Alternative Scenario for Invalid Values:
- User inserts invalid values for the setup parameters.
- Scenario returns to step 1.

2. Start Simulation:
- Scenario:
1. User initiates the simulation after setting up the parameters by pressing the "Start Simulation" button.
2. Application executes the simulation processes, including client arrival, queue management, service, and departure.
3. Application displays real-time updates of queue evolution during the simulation.

3. Display Real-time Queue Evolution:
- Scenario:
1. User observes real-time updates of the queues evolution during the simulation.
2. Application provides visual representation of changes in queue statuses, such as client arrivals, queue lengths, and service completions.

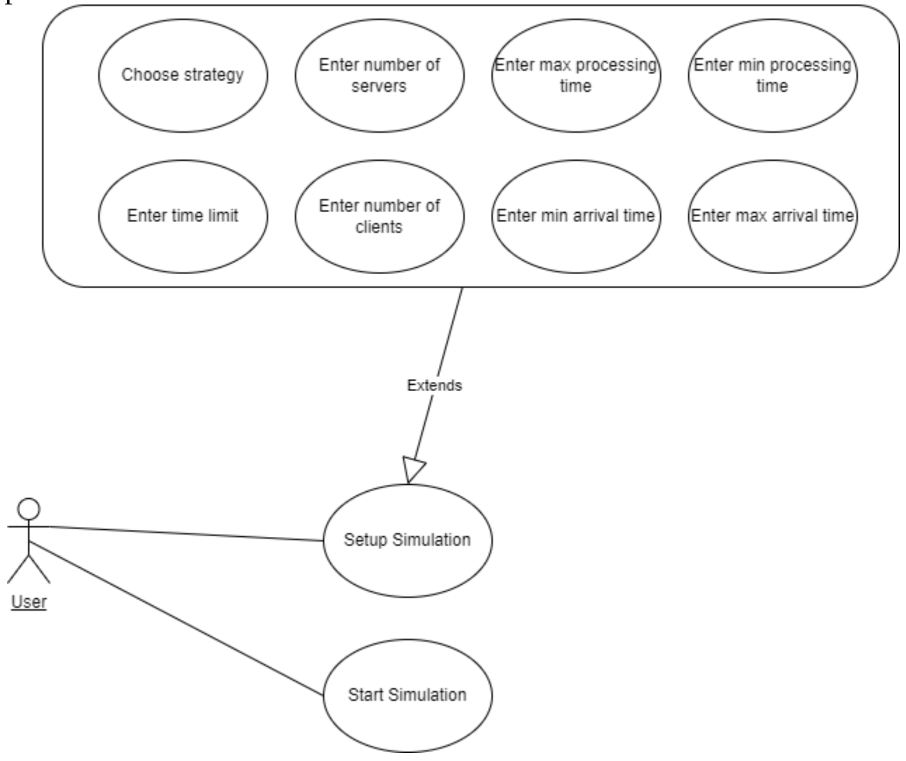
These use cases outline the interactions between the user and the simulation application, covering the setup process, simulation execution, and real-time visualization of queue evolution.

A flowchart is a visual representation of a process, algorithm, or workflow. It uses symbols and arrows to illustrate the sequence of steps in a sequential manner, starting from an initial task and progressing through decision points and actions until reaching a conclusion or end point. Flowcharts are widely used in various fields to visualize and analyze processes, aiding in understanding, problem-solving, and decision-making:



b. Use Case Diagram

A use case diagram is a graphical representation of the interactions between users (actors) and a system, depicting the various use cases or functionalities of the system and how users interact with them. It provides a high-level view of the system's behavior from the perspective of its users. Here's a use case diagram representing the queue management system application:



3. Design

The OOP design of the queue management system application promotes modularity, encapsulation, abstraction and maintainability. It adheres to key principles of object-oriented programming, enabling flexibility, extensibility, and robustness.

a. ***OOP design:***

1. Class Hierarchy:
   - The application consists of various classes representing different components such as 'Task', 'Server', 'Scheduler', 'SimulationManager', 'TextFile', 'QueueEvolutionGUI', and 'SimulationSetupGUI'.
   - These classes are organized in a hierarchical manner based on their functionalities and relationships.
   - For example, the 'Task' class represents a task entity in the simulation, while the `Server` class represents a server entity managing tasks.
   - Higher-level classes like 'Scheduler', 'SimulationManager', 'TextFile', 'QueueEvolutionGUI', and 'SimulationSetupGUI' coordinate the interactions between lower-level components.

2. Encapsulation:
   - Each class encapsulates its data and behavior, exposing only necessary methods for interaction.
   - For example, the 'Task' class encapsulates data related to task ID, arrival time, and service time, along with methods to manipulate this data.
   - Encapsulation ensures data integrity and hides implementation details, promoting modularity and code maintainability.

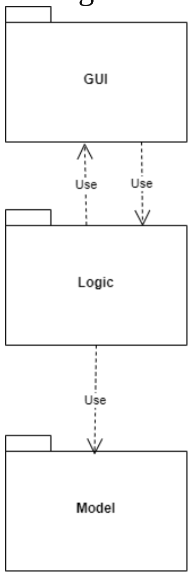
3. Abstraction:
   - Abstraction is used to represent real-world entities and behaviors in a simplified manner.
   - Classes like 'Task', 'Server', 'Scheduler', and 'SimulationManager' abstract different aspects of the simulation process, such as tasks, servers, scheduling logic, and simulation management.
   - Abstraction enables modeling complex systems in a comprehensible and manageable way.

4. Inheritance:
   - Inheritance is utilized to establish relationships between classes based on common characteristics and behaviors.
   - For example, the 'Server' class may inherit common attributes and methods from a superclass representing generic server behavior.
   - Inheritance promotes code reuse and facilitates the organization of related classes in a hierarchy.

5. Polymorphism:
   - Polymorphism allows objects of different classes to be treated uniformly through a common interface.
   - For example, the 'Scheduler' class can use polymorphism to interact with different strategies (e.g., 'ConcreteStrategyTime' and 'ConcreteStrategyQueue') through the 'Strategy' interface.
   - Polymorphism enhances flexibility and extensibility by enabling interchangeable components and dynamic behavior at runtime.

b. *UML package*

The UML package diagram provides a high-level overview of the organizational structure of the queue management application. It shows how classes are grouped into packages based on their functionalities and dependencies. This modular organization helps in maintaining the codebase, promoting code reusability, and managing dependencies effectively. Each package encapsulates related classes, promoting encapsulation and separation of concerns. Moreover, the dependency relationships between packages ensure that classes within different packages can interact as needed, facilitating modular development and testing:

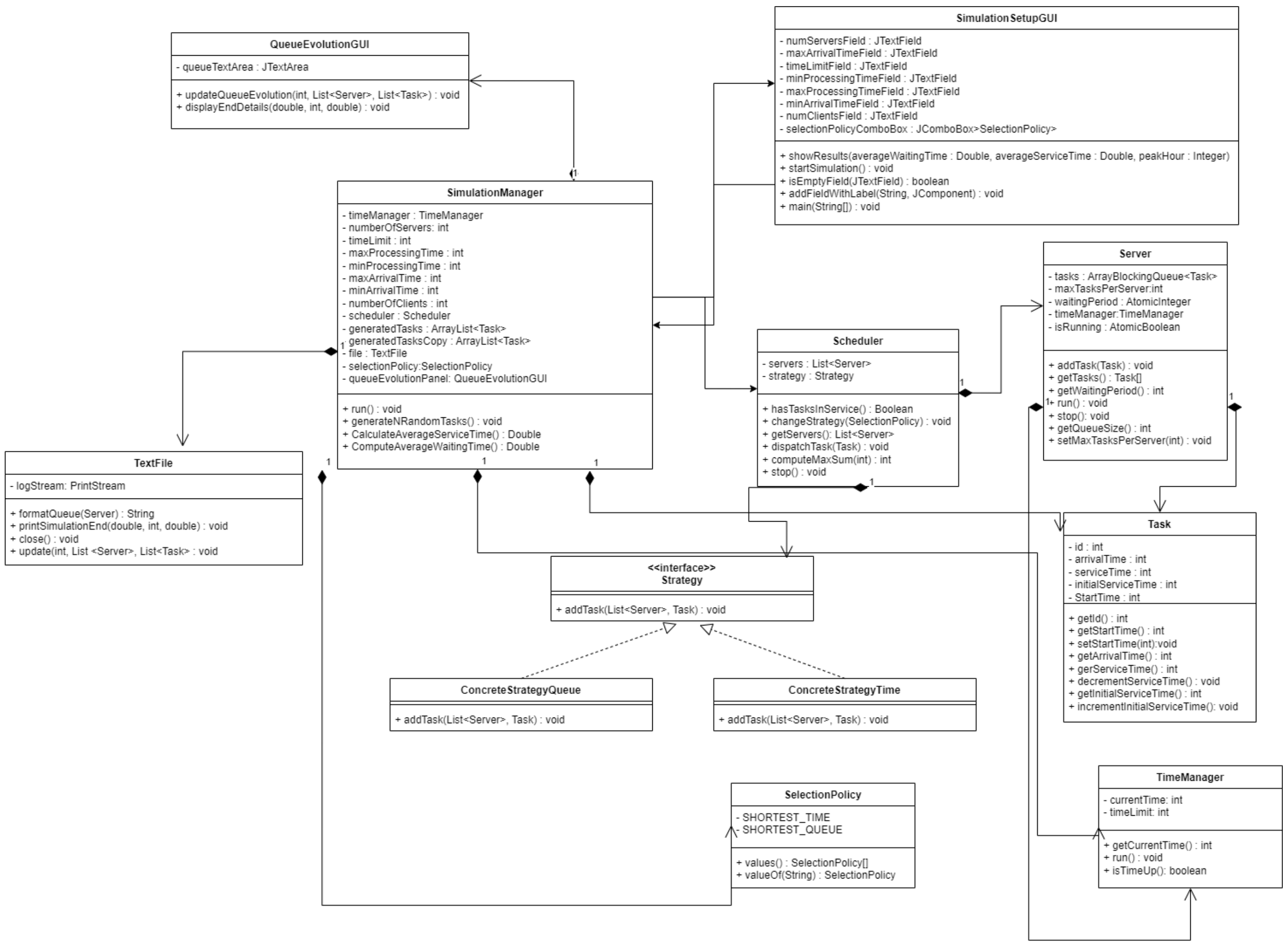


Packages:

- <u>GUI:</u> This package contains classes responsible for handling the graphical user interface (GUI) aspects of the application. It facilitates user interaction and visualization of the simulation process.
- <u>Logic:</u> This package contains classes responsible for implementing the logic and algorithms required for simulation management, queue assignment strategies, and time management.
- <u>Model:</u> This package contains classes that represent the underlying data model of the simulation. These classes encapsulate the data entities and their behaviors, providing the necessary functionality for simulating the queue management system.

c. *Class Diagram :*

A class diagram is a type of static structure diagram in UML that represents the structure of a system by showing the classes of the system, their attributes, methods, relationships, and constraints. It typically consists of class boxes connected by lines indicating relationships such as associations, inheritance, and dependencies.:


QueueEvolutionGUI
- queueTextArea : JTextArea
+ updateQueueEvolution(int, List<Server>, List<Task>) : void
+ displayEndDetails(double, int, double) : void
SimulationSetupGUI
- numServersField : JTextField
- maxArrivalTimeField : JTextField
- timeLimitField : JTextField
- minProcessingTimeField : JTextField
- maxProcessingTimeField : JTextField
- minArrivalTimeField : JTextField
- numClientsField : JTextField
- selectionPolicyComboBox : JComboBox>SelectionPolicy>
+ showResults(averageWaitingTime : Double, averageServiceTime : Double, peakHour : Integer)
+ startSimulation() : void
+ isEmptyField(JTextField) : boolean
+ addFieldWithLabel(String, JComponent) : void
+ main(String[]) : void
SimulationManager
- timeManager : TimeManager
- numberOfServers: int
- timeLimit : int
- maxProcessingTime : int
- minProcessingTime : int
- maxArrivalTime : int
- minArrivalTime : int
- numberOfClients : int
- scheduler : Scheduler
- generatedTasks : ArrayList<Task>
- generatedTasksCopy : ArrayList<Task>
- file : TextFile
- selectionPolicy:SelectionPolicy
- queueEvolutionPanel: QueueEvolutionGUI
+ run() : void
+ generateNRandomTasks() : void
+ CalculateAverageServiceTime() : Double
+ ComputeAverageWaitingTime() : Double
Server
- tasks : ArrayBlockingQueue<Task>
- maxTasksPerServer:int
- waitingPeriod : AtomicInteger
- timeManager:TimeManager
- isRunning : AtomicBoolean
+ addTask(Task) : void
+ getTasks() : Task[]
+ getWaitingPeriod() : int
+ run() : void
+ stop(): void
+ getQueueSize() : int
+ setMaxTasksPerServer(int) : void
Scheduler
- servers : List<Server>
- strategy : Strategy
+ hasTasksInService() : Boolean
+ changeStrategy(SelectionPolicy) : void
+ getServers(): List<Server>
+ dispatchTask(Task) : void
+ computeMaxSum(int) : int
+ stop() : void
TextFile
- logStream: PrintStream
+ formatQueue(Server) : String
+ printSimulationEnd(double, int, double) : void
+ close() : void
+ update(int, List <Server>, List<Task> : void
Task
- id : int
- arrivalTime : int
- serviceTime : int
- initialServiceTime : int
- StartTime : int
+ getId() : int
+ getStartTime() : int
+ setStartTime(int):void
+ getArrivalTime() : int
+ gerServiceTime() : int
+ decrementServiceTime() : void
+ getInitialServiceTime() : int
+ incrementInitialServiceTime(): void
<<interface>>
Strategy
+ addTask(List<Server>, Task) : void
ConcreteStrategyQueue
+ addTask(List<Server>, Task) : void
ConcreteStrategyTime
+ addTask(List<Server>, Task) : void
SelectionPolicy
- SHORTEST_TIME
- SHORTEST_QUEUE
+ values() : SelectionPolicy[]
+ valueOf(String) : SelectionPolicy
TimeManager
- currentTime: int
- timeLimit: int
+ getCurrentTime() : int
+ run() : void
+ isTimeUp(): boolean
1
1
1
1
1
1
1
1
1

d. *Defined Interfaces:*

Strategy Interface:

- The 'Strategy' interface defines a contract for implementing different queue assignment strategies.
- It provides a common method 'addTask' for adding tasks to servers based on specific strategies.
- Concrete strategy classes ('ConcreteStrategyTime' and 'ConcreteStrategyQueue') implement this interface to define strategy-specific behavior.

e. Used data structures

1. Lists and Arrays:

- Lists and arrays are used to store collections of tasks, servers, and other entities.
- For example, the 'List<Task>' and 'List<Server>' data structures are used to manage tasks and servers, respectively.
- Arrays are used to represent queues of tasks within servers ('Task[]').

2. Queue:

- The 'BlockingQueue<Task>' implementation is used for managing the queue of tasks waiting to be processed by servers.
- Queues facilitate first-in-first-out (FIFO) processing of tasks and ensure thread safety in concurrent environments.

f. Used Algorithms:

1. Queue Assignment Strategies:

- The application employs two queue assignment strategies: shortest time and shortest queue.
- The ConcreteStrategyTime class implements the shortest time strategy, assigning tasks to the server with the shortest remaining service time.
- The ConcreteStrategyQueue class implements the shortest queue strategy, assigning tasks to the server with the fewest tasks in its queue.
- These strategies utilize algorithms for selecting appropriate servers based on specific criteria, such as queue length or remaining service time.

2. Simulation Process Execution:

- The application simulates the arrival, queueing, service, and departure processes using algorithms executed within the Server class.
- Upon receiving tasks, servers execute a processing loop that decrements task service times and updates queue statuses until tasks are completed.
- Algorithms for task processing and queue management ensure the accurate simulation of real-world queueing systems.

The application's OOP design emphasizes modularity, encapsulation, and abstraction, utilizing appropriate data structures, interfaces, and algorithms to facilitate efficient simulation management and execution.

4. Implementation

In this chapter, a detailed description of each class in the queue management system application will be provided. This includes an overview of their fields, constructors, and methods, as well as their roles and interactions within the application. Understanding each class is essential for comprehending how the application functions and how different components work together to perform simulation. Additionally, the implementation of the graphical user interface (GUI) and how it facilitates user interaction with the application will be explored.

I. Task Class

Fields:

1. 'id' (int): Represents the unique identifier of the task.
2. 'arrivalTime' (int): Indicates the time at which the task arrives.
3. 'serviceTime' (int): Represents the remaining service time needed for the task to be completed.
4. 'startTime' (int): Represents the time at which the task starts being serviced. Initialized to -1.
5. 'initialServiceTime' (int): Represents the initial service time required for the task.

Constructor:

- 'Task(int id, int arrivalTime, int serviceTime)': Initializes a Task object with the provided 'id', 'arrivalTime', and 'serviceTime'. The 'startTime' is set to -1, indicating that the task has not started yet, and 'initialServiceTime' is set to 0.

Methods:

1. 'getId(): int'
   - Returns the unique identifier of the task ('id').
2. 'getArrivalTime(): int'
   - Returns the arrival time of the task ('arrivalTime').
3. 'getServiceTime(): int'
   - Returns the remaining service time needed for the task ('serviceTime').
4. 'decrementServiceTime(): void'
   - Decrements the remaining service time by 1 if it's greater than 0. This method simulates the progress of servicing the task.
5. 'getStartTime(): int'
   - Returns the start time of the task ('startTime').
6. 'setStartTime(int time): void'
   - Sets the start time of the task to the provided value ('time'). This method is used to mark the beginning of task service.
7. 'incrementInitialServiceTime(): void'
   - Increments the initial service time by 1. This method is used to track the total service time of the task.
8. 'getInitialServiceTime(): int'
   - Returns the initial service time required for the task ('initialServiceTime').

II. Server Class

Fields:

1. 'tasks' (BlockingQueue<Task>): Represents the queue of tasks to be processed by the server.

2. 'waitingPeriod' (AtomicInteger): Represents the total waiting time of tasks in the server's queue.
3. 'maxTasksPerServer' (int): Represents the maximum number of tasks allowed to be processed concurrently by the server.
4. 'timeManager' (TimeManager): Manages the current time for the server.
5. 'isRunning' (AtomicBoolean): Indicates whether the server is running or not.

<u>Constructor:</u>
- 'Server(TimeManager timeManager)': Initializes a Server object with a new 'LinkedBlockingQueue' for tasks, sets the 'waitingPeriod' to 0, assigns the provided 'timeManager', and sets 'isRunning' to true.

<u>Methods:</u>
1. 'stop(): void'
   - Stops the server by setting `isRunning` to false.
2. 'setMaxTasksPerServer(int maxTasksPerServer): void'
   - Sets the maximum number of tasks allowed to be processed concurrently by the server.
3. 'addTask(Task task): void'
   - Adds a task to the server's queue.
   - Updates the `waitingPeriod` by adding the service time of the task if the task is successfully added and the queue size is less than or equal to the maximum allowed tasks per server.
4. 'run(): void'
   - Implements the logic for the server's processing of tasks.
   - Continuously checks for tasks in the queue and processes them until the server is stopped.
   - For each task in the queue, sets its start time, decrements its service time in intervals of 1 second, updates the waiting period, and removes the task from the queue when its service time becomes 0.
5. 'getTasks(): Task[]'
   - Returns an array containing all tasks currently in the server's queue.
6. 'getQueueSize(): int'
   - Returns the size of the server's queue, indicating the number of tasks awaiting processing.
7. 'getWaitingPeriod(): int'
   - Returns the total waiting period of tasks in the server's queue.

III. Strategy Interface
<u>Method:</u>
1. 'addTask(List<Server> servers, Task t): void'
   - Responsible for adding a task t to a list of servers based on a specific strategy. Implementations of this interface define the strategy for adding tasks to servers.

IV. ConcreteStrategyTime Class
<u>Method</u>:
1. 'addTask(List<Server> servers, Task t): void'
   - Overrides the addTask method defined in the Strategy interface.
   - Implements the strategy of adding a task t to the server with the shortest waiting time.

  - Iterates over the list of servers to find the server with the shortest waiting time (getWaitingPeriod()), then adds the task to that server.

V. ConcreteStrategyQueue Class

Method:

1. 'addTask(List<Server> servers, Task t): void'
   - Overrides the addTask method defined in the Strategy interface.
   - Implements the strategy of adding a task t to the server with the shortest queue size.
   - Iterates over the list of servers to find the server with the shortest queue size (getQueueSize()), then adds the task to that server.

VI. SelectionPolicy Enum

- Enumeration representing different selection policies or strategies.
- Defines two constants: SHORTEST_QUEUE and SHORTEST_TIME.
  - SHORTEST_QUEUE: Represents the selection policy of choosing the server with the shortest queue length.
  - SHORTEST_TIME: Represents the selection policy of choosing the server with the shortest waiting time.

VII. TimeManager Class

Fields:

1. 'currentTime' (int): Represents the current time managed by the TimeManager.
2. 'timeLimit' (int): Represents the time limit or maximum allowed time managed by the TimeManager.

Constructor:

- 'TimeManager(int timeLimit)': Initializes a TimeManager object with the provided 'timeLimit'. Sets the 'currentTime' to -1 initially.

Methods:

1. 'run(): void'
   - Overrides the 'run' method from the 'Runnable' interface.
   - Implements the logic for time management.
   - Increments the 'currentTime' every second until it reaches the 'timeLimit', simulating the passage of time.
   - Sleeps the thread for 1000 milliseconds (1 second) between increments.
2. 'getCurrentTime(): int'
   - Returns the current time managed by the TimeManager ('currentTime').
3. 'isTimeUp(): boolean'
   - Checks if the current time has exceeded the time limit.
   - Returns true' if the current time is greater than the time limit, indicating that time is up; otherwise, returns false.

VIII. Scheduler Class

Fields:

1. 'servers' (List<Server>): Represents a list of servers managed by the scheduler.

2. 'strategy' (Strategy): Represents the strategy used for dispatching tasks to servers.

Constructor:
- 'Scheduler(int maxNoServers, int maxTasksPerServer, TimeManager timeManager)': Initializes a Scheduler object with the specified maximum number of servers ('maxNoServers'), maximum tasks per server ('maxTasksPerServer'), and a 'TimeManager' object for managing time.
  - Creates a list of servers and starts a thread for each server.
  - Sets the maximum tasks per server for each server.

Methods:
1. 'computeMaxSum(int maxQueueSizeSum): int'
   - Computes the maximum sum of queue sizes across all servers.
   - Calculates the sum of queue sizes for all servers and compares it with the provided 'maxQueueSizeSum'.
   - Returns the maximum of the current sum and 'maxQueueSizeSum'.
2. 'changeStrategy(SelectionPolicy policy): void'
   - Changes the strategy used for dispatching tasks based on the specified 'SelectionPolicy'.
   - Instantiates a new strategy object (either 'ConcreteStrategyQueue' or 'ConcreteStrategyTime') based on the provided policy.
3. 'dispatchTask(Task t): void'
   - Dispatches a task 't' to the servers using the current strategy.
   - Calls the 'addTask' method of the strategy, passing in the list of servers and the task.
4. 'getServers(): List<Server>'
   - Returns the list of servers managed by the scheduler.
5. 'hasTasksInService(): boolean'
   - Checks if any of the servers have tasks currently in service.
   - Iterates through all servers and checks if any server has tasks in its queue or if any task is being serviced.
   - Returns true if any server has tasks in service; otherwise, returns false.
6. 'stop(): void''
   - Stops all servers managed by the scheduler.
   - Calls the `stop` method for each server, ensuring that all server threads are terminated.

IX. SimulationManager Class

Fields:
1. 'timeLimit' (int): Represents the time limit for the simulation.
2. 'numberOfServers' (int): Represents the number of servers in the simulation.
3. 'numberOfClients' (int): Represents the number of clients/tasks in the simulation.
4. 'minArrivalTime' (int): Represents the minimum arrival time for tasks.
5. 'maxArrivalTime' (int): Represents the maximum arrival time for tasks.
6. 'maxProcessingTime' (int): Represents the maximum processing time for tasks.
7. 'minProcessingTime' (int): Represents the minimum processing time for tasks.
8. 'selectionPolicy' (SelectionPolicy): Represents the selection policy/strategy for task dispatching.
9. 'scheduler' (Scheduler): Manages the scheduling and dispatching of tasks to servers.
10. 'file' (TextFile): Manages the simulation log file.

11. 'generatedTasks' (List<Task>): Represents the list of generated tasks.
12. 'timeManager' (TimeManager): Manages the simulation time.
13. 'generatedTasksCopy' (List<Task>): Represents a copy of the generated tasks list.
14. 'queueEvolutionPane' (QueueEvolutionGUI): GUI component for visualizing queue evolution.

Constructor:
- 'SimulationManager(int timeLimit, int numberOfServers, int numberOfClients, int minArrivalTime, int maxArrivalTime, int maxProcessingTime, int minProcessingTime, SelectionPolicy selectionPolicy)': Initializes a SimulationManager object with the provided parameters.
  - Sets up the simulation parameters.
  - Instantiates the time manager, scheduler, log file manager, and queue evolution GUI.
  - Generates random tasks and sorts them based on arrival time.
  - Starts the time manager thread.
  - Configures the scheduler with the specified number of servers and selection policy.

Methods:
1. 'generateNRandomTasks(): void'
   - Generates a specified number of random tasks with arrival times and processing times within the specified ranges.
   - Sorts the generated tasks based on arrival time.
2. 'calculateAverageServiceTime(): double'
   - Calculates the average service time of tasks that have been serviced.
   - Iterates over the generated tasks copy and computes the total service time.
   - Returns the average service time.
3. 'computeAverageWaitingTime(): double'
   - Computes the average waiting time of tasks.
   - Iterates over the generated tasks copy and computes the total waiting time.
   - Returns the average waiting time.

4. 'run(): void'
   - Overrides the 'run' method from the 'Runnable' interface.
   - Implements the logic for running the simulation.
   - Continuously dispatches tasks to servers based on arrival times and updates the queue evolution and log file.
   - Tracks the maximum sum of queue sizes and determines the peak second.
   - Calculates and prints the average waiting time and service time at the end of the simulation.
   - Stops the scheduler and closes the log file at the end of the simulation.

X. TextFile Class
Fields:
1. 'logStream' (PrintStream): Represents the stream for writing log data to a text file.

Constructor:
- 'TextFile(String logFileName)': Initializes a TextFile object with the specified log file name.

- Creates a 'PrintStream' for writing to the log file specified by 'logFileName'.

Methods:
1. 'update(int time, List<Server> servers, List<Task> tasks): void'
   - Updates the log file with information about the current simulation state.
   - Writes the current time, waiting clients' details, and queue status of each server to the log file.
2. 'formatQueue(Server server): String'
   - Formats the queue status of a server as a string.
   - Returns a formatted string representation of the queue status, including task details if the queue is open.
3. 'close(): void'
   - Closes the log file stream.
   - Checks if the log stream is not null and closes it to release resources.
4. 'printSimulationEnd(double averageWaitingTime, int peakHour, double averageServiceTime): void'
   - Prints the summary of simulation results at the end of the simulation.
   - Writes the average waiting time, average service time, and peak hour to the log file.

XI. SimulationSetupGUI Class

Fields:
1. 'timeLimitField' (JTextField): Represents the text field for entering the simulation time limit.
2. 'numServersField' (JTextField): Represents the text field for entering the number of servers in the simulation.
3. 'numClientsField' (JTextField): Represents the text field for entering the number of clients in the simulation.
4. 'minArrivalTimeField' (JTextField): Represents the text field for entering the minimum arrival time for clients.
5. 'maxArrivalTimeField' (JTextField): Represents the text field for entering the maximum arrival time for clients.
6. 'minProcessingTimeField' (JTextField): Represents the text field for entering the minimum processing time for tasks.
7. 'maxProcessingTimeField' (JTextField): Represents the text field for entering the maximum processing time for tasks.
8. 'selectionPolicyComboBox' (JComboBox<SelectionPolicy>): Represents the combo box for selecting the dispatching policy/strategy.

Constructor:
- 'SimulationSetupGUI()': Initializes the SimulationSetupGUI frame and its components.
  - Sets the look and feel to FlatLaf.
  - Sets the title and default close operation for the frame.
  - Sets the layout to a grid layout with two columns.
  - Creates and configures text fields and combo box for entering simulation parameters.
  - Adds a "Start Simulation" button with an action listener to start the simulation.

Methods:
1. 'addFieldWithLabel(String labelText, JComponent field): void'

   - Adds a label and a corresponding input field/component to the GUI.
   - Takes a label text and a Swing component as input and adds them to the GUI.
2. 'startSimulation(): void'
   - Validates the input fields and starts the simulation.
   - Checks if any input field is empty, displays an error message if so, and returns.
   - Parses the input values from text fields.
   - Constructs a SimulationManager object with the entered parameters and selection policy.
   - Starts a new thread for the simulation manager.
   - Disposes of the setup GUI frame.
3. 'isEmptyField(JTextField field): boolean'
   - Checks if the specified text field is empty.
   - Returns true if the text field is empty or contains only whitespace; otherwise, returns false.
4. 'main(String[] args): void'
   - Entry point of the application.
   - Invokes the creation of the SimulationSetupGUI frame within the event dispatch thread.

XII. QueueEvolutionGUI Class

Fields:
1. 'queueTextArea' (JTextArea): Represents the text area for displaying the queue evolution and simulation end details.

Constructor:
- 'QueueEvolutionGUI()': Initializes the QueueEvolutionGUI frame and its components.
  - Sets the title and default close operation for the frame.
  - Sets the layout to BorderLayout.
  - Creates a JTextArea for displaying queue evolution and simulation end details.
  - Configures the JTextArea to be non-editable and sets its font.
  - Adds the JTextArea to a scroll pane and adds the scroll pane to the frame.
  - Sets the size, location, and visibility of the frame.

Methods:
1. 'updateQueueEvolution(int currentTime, List<Server> servers, List<Task> remainingTasks): void'
   - Updates the queue evolution displayed in the JTextArea.
   - Clears the text area and displays the current time, waiting clients, and queue status of each server.
   - Formats the display to show tasks in the waiting list and the queue of each server.
2. 'displayEndDetails(double avgWaitingTime, int peakHour, double avgServiceTime): void'
   - Displays the simulation end details in the JTextArea.
   - Appends the average waiting time, average service time, and peak hour to the JTextArea.
   - Separates the end details with a horizontal line for clarity.

5. Conclusions

The simulation application for queue management systems provides a comprehensive solution for modeling and analyzing the performance of queueing systems. By implementing various functionalities such as task generation, server management, simulation execution, and result tracking, the application offers valuable insights into queue dynamics, waiting times, and service efficiency. Through the graphical user interface, users can easily configure simulation parameters, visualize queue evolution, and interpret simulation results.

What I Have Learned from the Assignment:

1. Queue Management Concepts: This assignment deepened my understanding of queue management concepts, including arrival processes, service times, queue dynamics, and performance metrics like waiting times and service rates.

2. Object-Oriented Design: Designing and implementing the application's object-oriented architecture helped me reinforce concepts such as encapsulation, inheritance, polymorphism, and abstraction. It emphasized the importance of modular design and code organization for building scalable and maintainable software systems.

3. Algorithm Implementation: Implementing queue assignment strategies and simulation execution algorithms provided hands-on experience in algorithm design and optimization. It involved considering factors like task prioritization, server selection, and real-time processing.

4. Graphical User Interface Development: Developing the GUI components enhanced my skills in designing user-friendly interfaces and handling user interactions. It involved creating intuitive layouts, implementing event-driven behavior, and visualizing simulation data effectively.

Future Developments:

1. Enhanced Visualization: Future developments could focus on enhancing the visualization capabilities of the application. This could include adding more graphical elements, interactive charts, and real-time analytics to provide deeper insights into simulation results.

2. Advanced Queueing Models: Introducing support for more complex queueing models and simulation scenarios could expand the application's utility. This could involve incorporating features like priority queues, multiple server types, service interruptions, and custom arrival distributions.

3. Performance Optimization: Optimizing the application for better performance and scalability would be beneficial for handling larger simulation scenarios and datasets. This could involve refining algorithms, implementing parallel processing techniques, and leveraging optimization frameworks.

4. User Customization: Providing users with more customization options for simulation parameters, queue assignment strategies, and result presentation would enhance the application's flexibility and usability. This could include configurable simulation settings, strategy plugins, and customizable dashboards.

Overall, the assignment provided valuable insights into queue management systems, software development principles, and graphical user interface design. By incorporating feedback, iterating on design choices, and embracing continuous improvement, future developments can further enhance the application's functionality and usability to meet evolving user needs and requirements.

6. Bibliography

The following references were consulted during the implementation of the queue management application:

1. http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
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.tutorialspoint.com/java/pdf/java_multithreading.pdf
5. https://dsrl.eu/courses/pt/materials/PT_2024_A2_S1.pdf
6. https://dsrl.eu/courses/pt/materials/PT_2024_A2_S2.pdf