DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: **Muresan Mara**
GROUP: **30423**

CONTENTS

1. Assignment Objective

- Main Objective
    - 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 queue, and (2) computing the average waiting time, average service time and peak hour.
- Sub-objectives
    - Analyze the problem and identify the requirements (Section 2)
        - Examining the problem statement to understand the goals and constraints of the project
        - Identifying the specific requirements of the queues management system, such as the supported strategies for inserting the clients in the queues, the input data and the desired user interface features
    - Design the simulation application (Section 3)
        - Designing the architecture of the project and sketching the user interface
        - Creating a plan for how the simulation will function, including the threads mechanisms and algorithms needed to handle the synchronization of the simulation
    - Implement the simulation application (Section 4)
        - Writing the code to implement the simulation based on the design structured at the previous step
        - Implementing the necessary classes, methods and GUI features
    - Test the simulation application (Section 5)
        - Conducting different tests to ensure that the simulation management meets the specified requirements and functions correctly
        - Testing each individual component/method and then testing the project as a whole

2. Problem Analysis, Modeling, Scenarios, Use Cases

- Functional requirements
    - The simulation application should allow users to setup the simulation (insert a number of clients, a number of queues, a simulation interval, a minimum and a maximum arrival time, a minimum and a maximum service time and choose the strategy policy, shortest time/shortest queue, from a comboBox)
    - The simulation application should allow users to start the simulation
    - The simulation application should allow users to intentionally stop the simulation at any time they want
    - The simulation application should display the real-time queues evolution
    - The simulation application should display, in the end, the results of the simulation (the average waiting time, the average service time and the peak hour)
    - The simulation application should display an error/warning message if the user tries to perform a simulation on some inputs wrongly introduced (the inputs should be integers, minimum arrival time <= maximum arrival time and minimum service time <= maximum service time)
    - The simulation application should efficiently manage memory usage, ensuring that it can handle simulations with large data without crashing or becoming unresponsive
- Non-functional requirements
    - The graphical interface should be visually appealing and user-friendly, with intuitive controls and clear feedback to the user (example: when writing the inputs, the application clearly specifies in each textBox what type of input it expects)
    - The simulation application should respond to user inputs quickly and efficiently, with minimal delay in performing the computations and writing the log of events
    - The simulation application should be reliable and dependable, functioning correctly without crashing or producing errors under normal usage conditions

Use Cases

Description

Use Case: Start the simulation

Primary Actor: User

Success Scenario Steps:

1. The user inserts the values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, minimum and maximum service time, and chooses the strategy policy: shortest time/shortest queue
2. The user clicks on the “START” button
3. The application checks if the inputs respect the correct structure (each value should be an integer, minimum arrival time <= maximum arrival time and minimum service time <= maximum service time)
4. The application performs the simulation displaying the log of events (the real-time queue evolution), and giving, in the end, the simulation results (the average waiting time, the average service time and the peak hour)

Alternative Sequence: Invalid values for the setup parameters

- The user inserts invalid values for the application’s setup parameters
- The application displays a messageDialog which contains a clear error message (“Enter valid time intervals!”/”Enter valid numbers in all fields!”) and requests the user to insert valid values
- The scenario returns to step 1

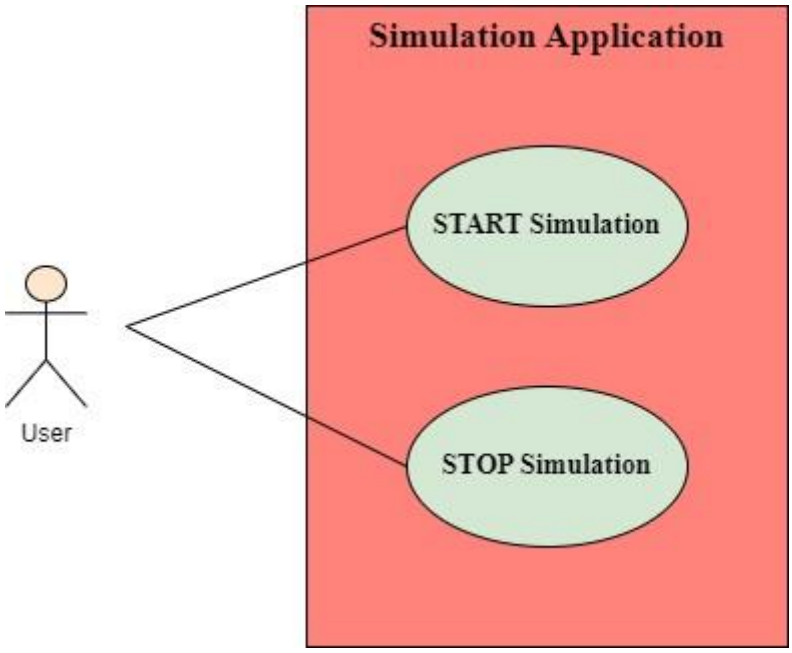
Use Case: Stop the simulation

Primary Actor: User

Success Scenario Steps: (it doesn’t exist an “Alternative Sequence”)

1. The user clicks on the “STOP” button
2. The simulation is stopped and a message is displayed (“Simulation Was Stopped.”)

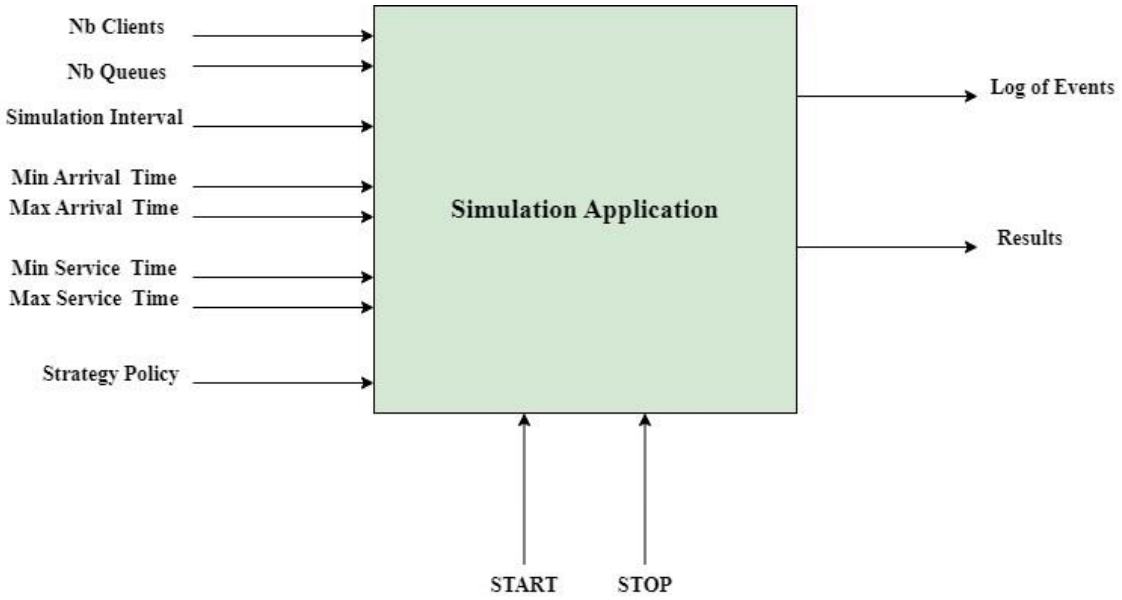
Use Case Diagram



Reference: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/

3. Design

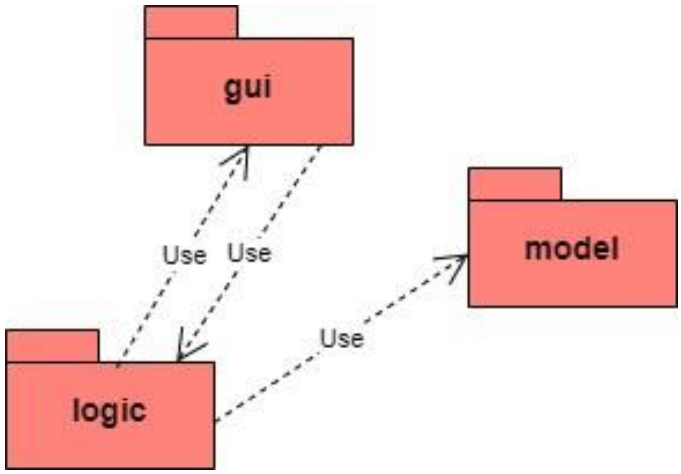
The System's "Black Box"



I organized the application into 3 packages:

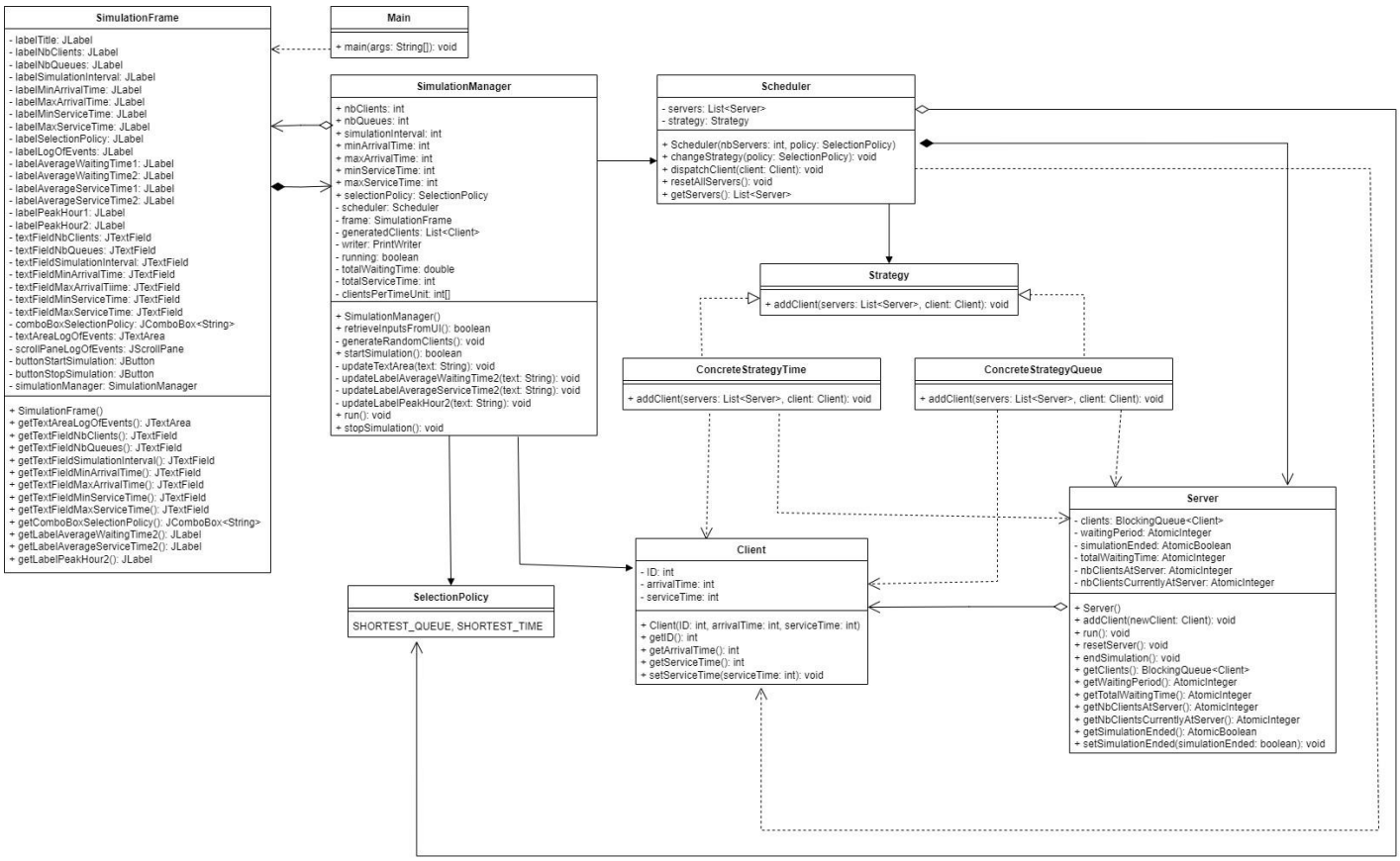
- model
    - contains the classes that models the application data: "***Server***", "***Client***",
- gui
    - contains the class that builds the user interface, responds to user interactions and triggers appropriate actions within the simulator functionalities: "***SimulationFrame***"
- logic
    - contains the "***Strategy***" interface (for defining the 2 possible strategies)
    - contains the classes used to build the 2 strategies for inserting clients in the queues: "***ConcreteStrategyQueue***" (the client goes to the shortest queue), "***ConcreteStrategyTime***" (the client goes to the queue with the shortest time); these classes use the "***SelectionPolicy***" enum
    - contains the class used to change between strategies and to physically add the clients in the queues: "***Scheduler***"; it also creates the threads for each server
    - contains the class that connects all the components and creates the main thread: "***SimulationManager***"; it also starts the simulation and writes the outputs

The UML Package Diagram



The UML Class Diagram

Reference: https://www.gleek.io/blog/uml-relationships



Used Data Structures (Reference: https://dsrl.eu/courses/pt/materials/PT_2024_A2_S2.pdf)

The queue simulation management system uses several advanced data structures and concurrency mechanisms to ensure thread safety, efficiency, and data integrity in a multi-threaded context.

- *Threads and the "Runnable" Interface*

The system utilizes Java threads to manage multiple service queues concurrently, allowing the simulation to process multiple clients simultaneously. Each server runs on its own thread, handling client operations independently (besides that, the main thread is given by the SimulationManager class). Each thread is created by implementing the "Runnable" interface.

This design choice allows each server to define its own thread execution behavior, encapsulating the logic needed to process clients over time. The "run" method is the heart of this operation, where client servicing logic is executed periodically.

- *BlockingQueue* (Reference: https://www.baeldung.com/java-blocking-queue)

To manage the clients efficiently, each server uses a "BlockingQueue" to hold incoming client requests. This approach is particularly suited for this task due to its thread-safe properties, ensuring that data is correctly handled across multiple threads without requiring additional synchronization. It also handles all internal concurrency control, making it ideal for scenarios where items are produces by one thread and consumed by another. (The waiting clients list is generated in the main thread SimulationManager and these clients are used in the servers threads)

- *Atomic Variables* (Reference: https://www.baeldung.com/java-atomic-variables)

Atomic variables ("AtomicInteger" and "AtomicBoolean") play a crucial role in safely managing counters and flags across multiple threads

AtomicInteger: Used for counting clients and waiting periods, ensure that increments and decrements are atomic operations. This is critical in a multi-threaded environment to prevent race conditions and ensure the accuracy of operation counts and time calculations.

AtomicBoolean: The "simulationEnded" flag safely signals when the server should stop processing, providing a thread-safe way to manage the lifecycle of each server thread.

Defined Interfaces

The primary benefit of using the "Strategy" interface is that it allows the specific algorithms for adding clients to servers to be encapsulated within different strategy classes. This encapsulation makes the client distribution algorithms modular and interchangeable without altering the clients that use them.

- "ConcreteStrategyQueue"
    - This strategy implements a client distribution method based on the number of clients. It always assigns a new client to the server with the fewest clients, potentially balancing the load by the number of clients in each queue.
- "ConcreteStrategyTime"

- This strategy considers the total waiting time of each server to decide where to add the new client. By choosing the server with the shortest waiting period, it aims to minimize the average waiting time for clients, which can lead to improved overall service time performance.

Used Algorithms

The most challenging algorithm was the algorithm for computing the average waiting time of the system. The average waiting time metric is a critical performance indicator in queue management systems. It reflects the average amount of time clients spend waiting in queues before being serviced. I compute this value by calculating the average waiting time for each individual queue and then averaging these averages to obtain an overall system average. This method provides a comprehensive view of the waiting time performance across all queues.

4. Implementation

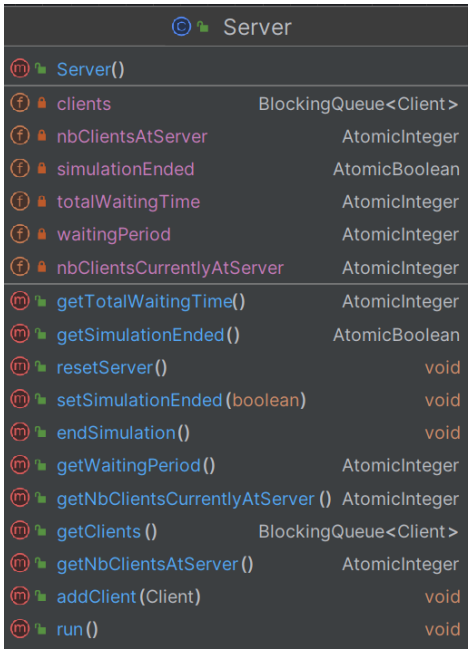
The application contains 10 classes.

1. **Client**

The **constructor** builds a new client with the specified **ID**, **arival time** and **service time**.

The **getters** and **setters** are used to return the client's ID, arivalTime and serviceTime and to set the client's serviceTime (this method allows for dynamic adjustments to the serviceTime).

2. Server



The **constructor** initializes a new instance of the Server with empty client queues and all counters set to zero. The simulation flag is initially set to false, indicating that the simulation is not complete.

addClient(Client newClient) adds a new client to the queue and updates the relevant statistics.

run() implements the server's processing logic that runs in a separate thread; handles client servicing by decrementing waitingPeriod and serviceTime, removing clients from the queue when their service completes, and sleeps for a fixed interval (1 second) to simulate time passage.

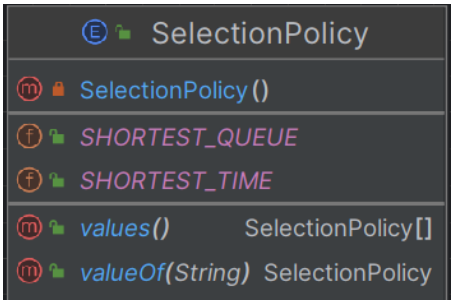
```
@Override
public void run() {

    while(!simulationEnded.get()) {
        try {
            if(!clients.isEmpty()) {
                Client client = clients.peek();
                if(client != null) {
                    if (client.getServiceTime() == 1) {
                        clients.take();
                        if(clients.isEmpty()) endSimulation();
                        nbClientsCurrentlyAtServer.addAndGet( delta: -1);
                    }
                    client.setServiceTime(client.getServiceTime() - 1);
                    waitingPeriod.addAndGet( delta: -1);
                }
            }
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println("Server thread was interrupted.");
        }
    }
}
```

resetServer() clears all clients from the queue and resets all counters to zero.

endSimulation() sets the simulation flag to true, indicating the end of the simulation.

The getters and the setters provide controlled access to the server's state and allow for adjustments to the simulation's settings.

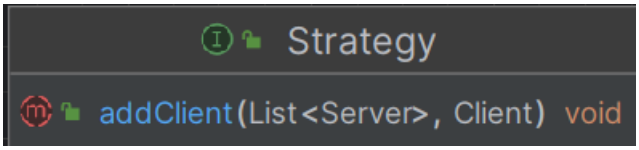
3. **SelectionPolicy** (enum)



Defines the strategies available for selecting queues in the system managing multiple queues.

- **SHORTEST_QUEUE**
    - This policy indicates that the system should select the queue with the fewest number of clients.
    - This approach aims to balance the load by minimizing the queue length.
- **SHORTEST_TIME**
    - This policy directs the system to choose the queue with the shortest waitingPeriod for new clients.
    - This approach aims to minimize the waiting period for all the clients.

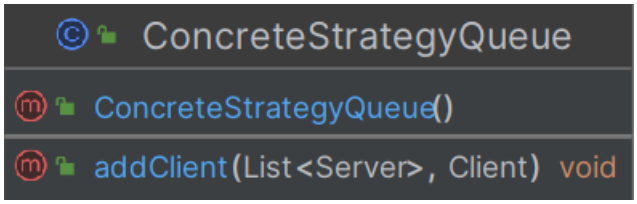
4. **Strategy** (interface)



This interface is crucial for the modular design of the system, allowing for different strategies that can be interchanged easily depending on the system requirements.

addClient(List<Server> servers, Client client) defines a method to add a client to one of the available servers provided by the list of servers.

5. **ConcreteStrategyQueue**

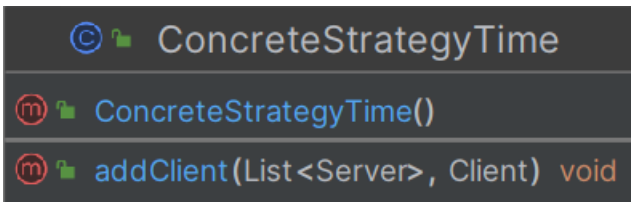


Implements the “Strategy” interface, focusing on a queue selection strategy that assigns a client to the server with the fewest current clients.

The **addClient (List<Server> servers, Client client)** method iterates through the list of servers to find the one with the smallest number of clients. Once identified, the client is added to this server. If the server's simulation has ended, it is restarted by launching a new thread.

```java
@Override
public void addClient(List<Server> servers, Client client) {
    Server fewClientsServer = servers.get(0);
    for(Server server : servers) {
        if(server.getClients().size() < fewClientsServer.getClients().size()) {
            fewClientsServer = server;
        }
    }
    fewClientsServer.addClient(client);
    if(fewClientsServer.getSimulationEnded().get()) {
        fewClientsServer.setSimulationEnded(false);
        Thread t = new Thread(fewClientsServer);
        t.start();
    }
}
```

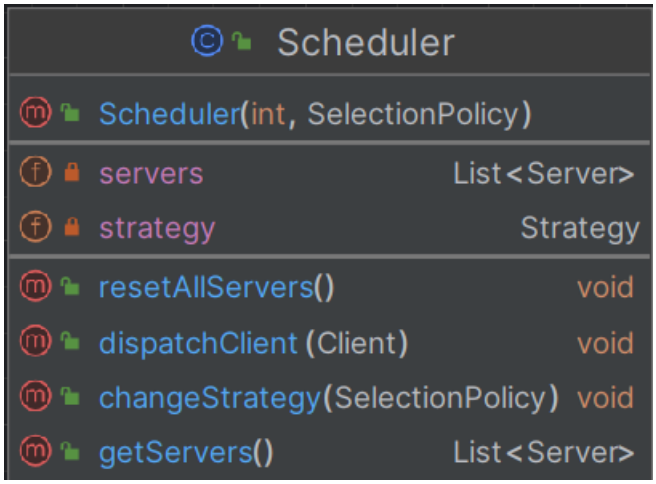
6. **ConcreteStrategyTime**



Implements the “Strategy” interface, focusing on assigning clients to the server with the lowest cumulative waiting time.

The **addClient (List<Server> servers, Client client)** method iterates through the list of servers to find the one with the least total waitingPeriod. Once identified, the client is added to this server. If the server's simulation has ended, it is restarted by launching a new thread.

7. **Scheduler**



The **constructor** builds server objects, starts each in its thread, and sets the distribution strategy according to the specified policy.

changeStrategy(SelectionPolicy policy) changes the strategy for adding clients to servers based on the specified policy. Supports dynamic strategy adjustments during runtime.

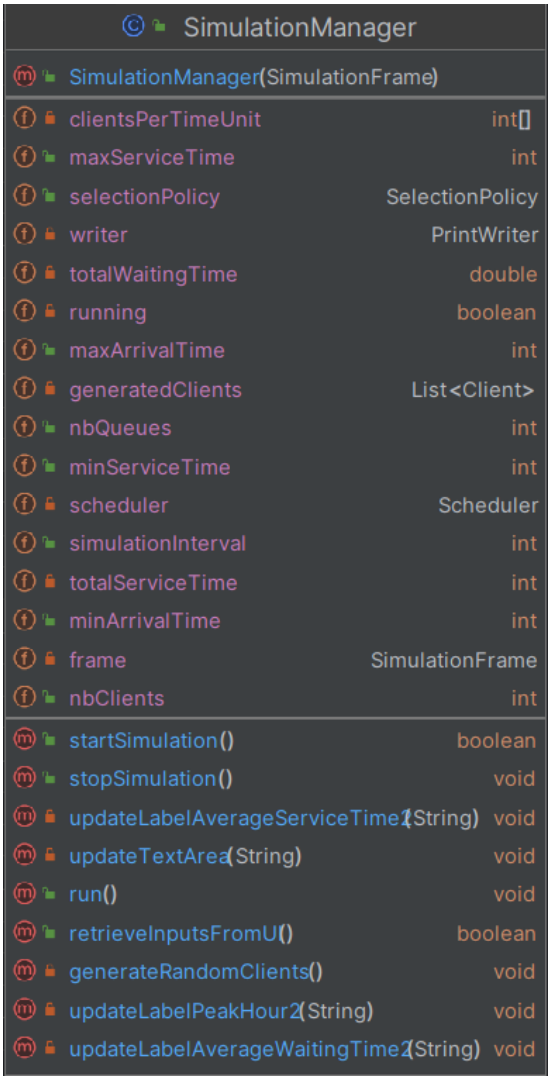
dispatchClient(Client client) delegates a client to the appropriate server based on the current strategy. This method is central to the scheduler's function, utilizing the strategy pattern to adapt the client distribution logic.

```
public void changeStrategy(SelectionPolicy policy) {
    if(policy == SelectionPolicy.SHORTEST_QUEUE) {
        strategy = new ConcreteStrategyQueue();
    }
    if(policy == SelectionPolicy.SHORTEST_TIME) {
        strategy = new ConcreteStrategyTime();
    }
}
```

resetAllServers() resets all managed servers to their initial state. This method is useful for restarting the server system without stopping the application.

The **getter** method returns the list of servers currently managed by the scheduler.

8. SimulationManager



This class is responsible for managing the simulation of client-server interactions in a dynamic environment. It integrates UI components for user interaction and controls the flow of the simulation, including client generation, server allocation, and simulation progress tracking. This class also handles file operations for logging the simulation events and results.

The **constructor** sets up the GUI frame and prepares the file writer for outputting simulation results and events.

retrieveInputsFromUI() retrieves simulation parameters from the user interface, validating and parsing them as necessary. It configures the simulation environment based on user input.

generateRandomClients() generates clients with random arrival and service times within the specified intervals. Sorts clients by arrival time for efficient processing.

```
private void generateRandomClients() {
    Random random = new Random();
    for(int i = 1; i <= nbClients; i++) {
        int ID = i;
        int arrivalTime = random.nextInt( bound: maxArrivalTime - minArrivalTime + 1) + minArrivalTime;
        int serviceTime = random.nextInt( bound: maxServiceTime - minServiceTime + 1) + minServiceTime;
        generatedClients.add(new Client(ID, arrivalTime, serviceTime));
        totalServiceTime = totalServiceTime + serviceTime;
    }

    generatedClients.sort((c1, c2) -> Integer.compare(c1.getArrivalTime(), c2.getArrivalTime()));
}
```

startSimulation() starts the simulation if the inputs are valid and the simulation is not already running. It prepares the environment and creates a new thread for the simulation process.

```
public boolean startSimulation() {
    if (retrieveInputsFromUI()) {
        if (running) {
            return true;
        }
        scheduler.resetAllServers();
        generatedClients = new ArrayList<>();
        clientsPerTimeUnit = new int[simulationInterval];
        totalWaitingTime = 0;
        totalServiceTime = 0;

        generateRandomClients();

        running = true;
        Thread simulationThread = new Thread( task: this);
        simulationThread.start();
        return true;
    }
    return false;
}
```

GUI update methods, SwingUtilities.invokeLater calls, update the GUI components to reflect the current state and results of the simulation:

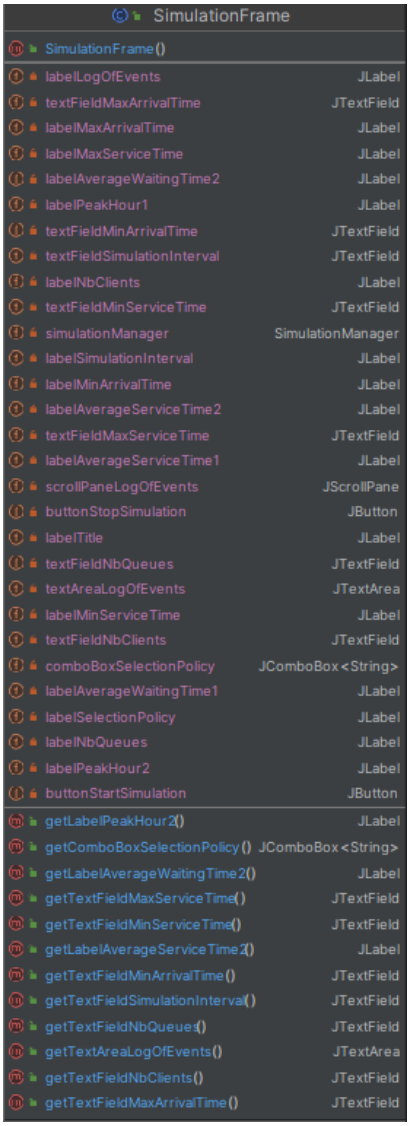
- **updateTextArea()**
- **updateLabelAverageWaitingTime2()**
- **updateLabelAverageServiceTime2()**
- **updatePeakHour2()**

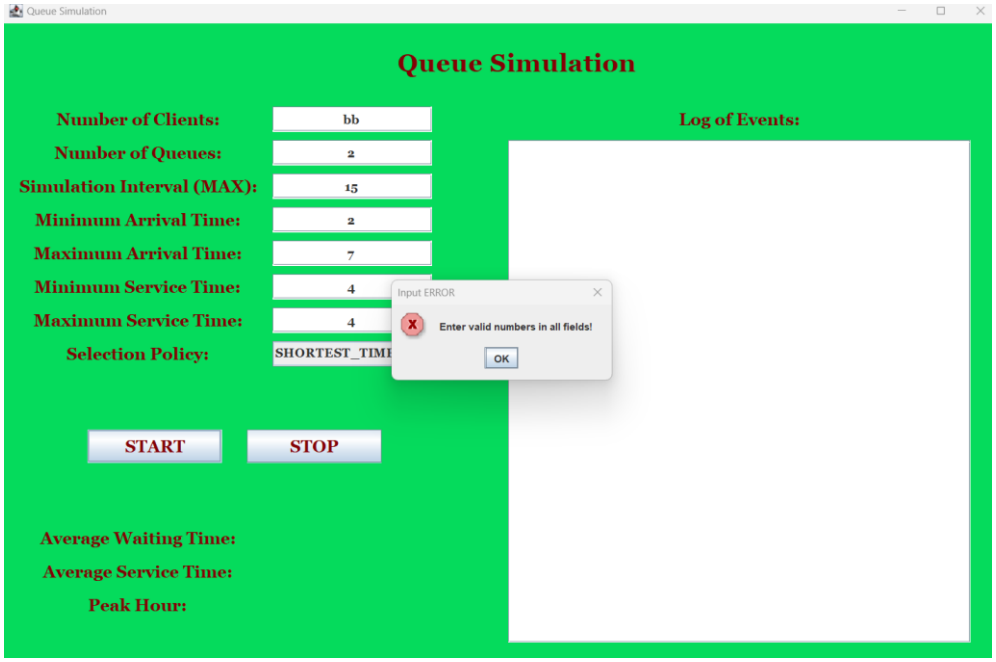
```
private void updateTextArea(String text) {
    SwingUtilities.invokeLater(() -> frame.getTextAreaLogOfEvents().append(text));
}
```

run() is the core method that runs in a separate thread to manage the timing and processing of the simulation. It handles client distribution to servers, updates the GUI with simulation progress, and logs the details to a file.

stopSimulation() stops the simulation and signals all servers to end their operations.

9. SimulationFrame





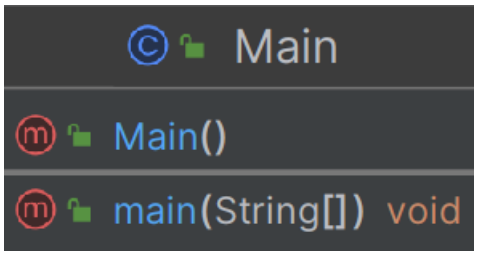
This class represents the **graphical user interface (GUI)** of the simulation application. I utilized **Swing** components to create a window with input textFields and comboBox, buttons for starting and stopping the simulation, and labels and textArea to display the overall results and log of events.

It contains a **constructor** which initializes the GUI components, sets their properties (font, color, bounds), and adds action listeners to the buttons to handle the user interactions.

When a button is clicked, corresponding actions are performed, such as invoking methods from the “SimulationManager” class (startSimulation() and stopSimulation()).

```
buttonStartSimulation = new JButton( text: "START");
buttonStartSimulation.setFont(new Font( name: "Georgia", Font.BOLD,  size: 20));
buttonStartSimulation.setForeground(new Color( r: 132,  g: 1,  b: 6));
buttonStartSimulation.setBounds( x: 100,  y: 485,  width: 160,  height: 40);
 Mara *
buttonStartSimulation.addActionListener(new ActionListener() {
    Mara *
    @Override
    public void actionPerformed(ActionEvent e) {
        textAreaLogOfEvents.setText("");
        labelAverageWaitingTime2.setText("");
        labelAverageServiceTime2.setText("");
        labelPeakHour2.setText("");
        simulationManager.startSimulation();
    }
});
add(buttonStartSimulation);
```

10. Main



This class serves as an entry point of the simulation application. It sets up the main JFrame window, and displays it to the user.

The application will exit when the user closes the window.

5. Results

The testing approach involved manually inputting scenarios into the system and closely observing the outputs (I didn’t use JUnit). These scenarios were derived directly from examples specified within the project requirements, ensuring that the tests were both relevant and comprehensive.

The outputs from each test scenario were systematically recorded in three text files: test1.txt, test2.txt, and test3.txt. These files contain detailed results including the log of events, the average waiting time, the average service time, and the peak hour, facilitating an easy review of the system's performance under various conditions.

The average waiting time and average service time numbers are very close in values across different test scenarios. This consistency suggests that the system's internal scheduling and client handling mechanisms are functioning as expected, maintaining a balanced load among the available service queues.

The proximity of average waiting time to average service time indicates that most clients are being processed efficiently, with minimal delay added by queue management overhead.

6. Conclusions

In conclusion, this project was a success, because I solved all the requirements for the simulation management application.

Through this assignment, I understood better how to represent, manipulate and perform computations and simulations with threads, all by using Java programming techniques. Also, I clarified the OOP principles such as aggregation, composition, encapsulation, inheritance , realization, and polymorphism. I used Swing for creating the graphical user interface and I can say that I gained a little more experience using it. The process was faster and clearer than the last time I used Swing, but I still think that it would have been easier to use Scene Builder, where we could place the GUI components with drag and drop on a panel to make an idea about how the GUI will look (also the JavaFX code would have been generated automatically).

As future developments, I could modify the interface to be more dynamic and more visually appealing. I could use images with little humans that are placed at the queues in real time using motion. I could also extend the functionalities to support additional criteria for placing the clients at the queues, for example I can introduce a maximum possible number of clients present at the queues at a certain time. Also, a good idea would be to implement memory management features to allow users to store and recall previously entered simulations.

7. Bibliography

In addition to the links added during the documentation in the specific section, I studied the Lectures and the Assignment Guides posted on the *Fundamental Programming Techniques* website: https://dsrl.eu/courses/pt/

I drew the diagrams using this website: https://app.diagrams.net/

I made an overall idea of the project using this website: https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html