DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: Boboș Răzvan-Andrei
GROUP: 30425

CONTENTS

1. Assignment Objective

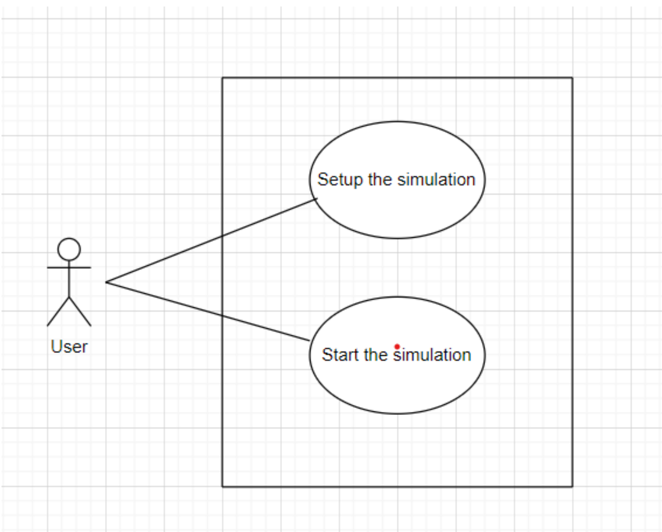
Main Objective

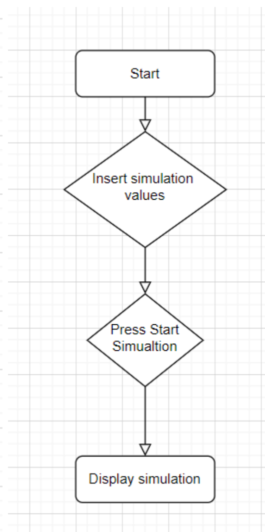
Design and implement a queues management application which assigns clients to queues such that the waiting time is minimized.

Sub-objectives

- Analyze the problem and identify requirements
- Design the simulation application
- Implement the simulation application
- Test the simulation application

2. Problem Analysis, Modeling, Scenarios, Use Cases





Functional requirements:

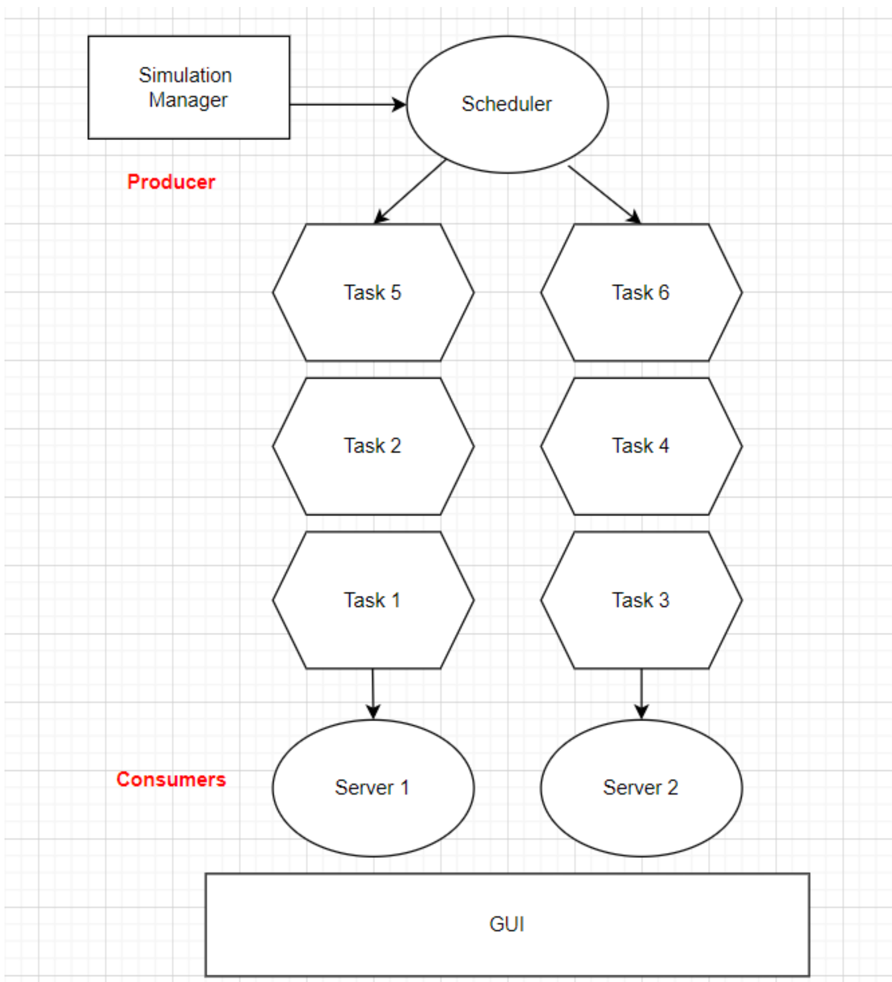
- The simulation application should allow users to setup the simulation
- The simulation application should allow users to start the simulation
- The simulation application should display the real-time queues evolution
- At the end of the simulation, users should be provided with detailed statistics and analytics about the simulation run, including waiting times and service timesand peak hour analysis.

Non-Functional requirements:

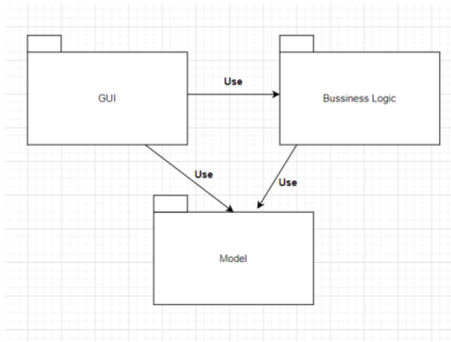
- The simulation application should be intuitive and easy to use by the user
- The simulation application should be capable of handling large-scale simulations efficiently without significant performance degradation.
- The application should include comprehensive documentation, including user manuals, developer guides, and API references, to aid users and developers in understanding and utilizing the software effectively.

3. Design

Overall System Design



Division into packages



Division into classes

To design the object-oriented architecture of the queue managament application, we can break down the functionality into several classes representing different components of the system. Here's a conceptual overview of the OOP design:

Task.java

The **Task** class represents an individual task with attributes such as ID, arrival time, service time, and dispatch status. It facilitates setting and retrieving task details, including whether it has been dispatched, for efficient task management within a system.

Server.java

The **Server** class represents a server entity capable of processing tasks concurrently. It manages a queue of tasks and processes them sequentially, decrementing their service time until completion.

Scheduler.java

The **Scheduler** class orchestrates task dispatching among multiple servers based on predefined strategies. It dynamically allocates tasks to servers, considering factors like queue length or processing time.

Strategy.java (Interface) & ShortestTimeStrategy/ShortestQueueStartegy

The **Strategy** interface defines a contract for implementing different task scheduling strategies. Implementations include **ShortestTimeStrategy**, which assigns tasks to servers with the shortest estimated remaining processing time, and ShortestQueueStrategy, which allocates tasks to servers with the shortest queue length.

The **SelectionPolicy enum** enumerates the available task selection policies, including **SHORTEST_QUEUE** and **SHORTEST_TIME**, providing a clear and concise representation of the strategies employed by the system.

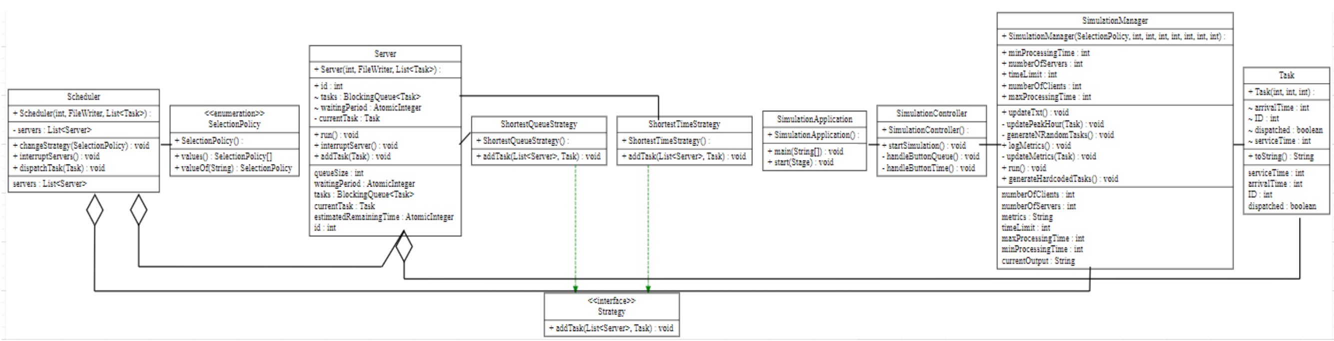
SimulationManager.java

The **SimulationManager** class orchestrates the simulation, handling task generation, scheduling, and metrics calculation. It implements the **Runnable** interface to run as a concurrent thread. The class interacts with the **Scheduler** component to dispatch tasks to servers based on the selected strategy defined by the **SelectionPolicy.**

SimulationController.java

The **SimulationController** class serves as the controller for the graphical user interface (GUI) of the simulation application.

Here's how these classes might relate to each other in a basic UML class diagram:



Used data structures and defined interfaces

The simulation application utilizes several data structures for managing tasks and server queues efficiently:

- **AtomicInteger for Waiting Period (Server class):** Each server maintains an AtomicInteger named waitingPeriod, which represents the estimated remaining time for all tasks in the server's queue. This AtomicInteger allows for thread-safe updates to the waiting period as tasks are added or completed.

- **BlockingQueue for Server Tasks (Server class):** The BlockingQueue<Task> data structure is used to store the tasks currently in each server's queue. This queue ensures thread safety and provides blocking behavior when attempting to add tasks to a full queue or retrieve tasks from an empty queue.

- **Strategy Interface for Task Dispatching (Strategy interface):** The Strategy interface defines a common method addTask for selecting and dispatching tasks to servers based on different strategies. It serves as a flexible abstraction for implementing various task dispatching policies.

These data structures facilitate efficient task management, server queue handling, and task dispatching within the simulation application, ensuring smooth and accurate simulation execution.

4. Implementation

To comprehend the implementation thoroughly, we will delve into each class's implementation in detail, examining how they contribute to the overall functionality of the project.

Task.java

The **Task** class represents a task in the simulation, containing information such as ID, arrival time, service time, and whether it has been dispatched. It provides methods to retrieve and modify these attributes, as well as a **toString()** method for convenient string representation.

```java
public class Task {
    4 usages
    int ID;
    4 usages
    int arrivalTime;//volatile
    5 usages
    int serviceTime;//volatile
    3 usages
    boolean dispatched;//volatile
    7 usages   Razvan
    public Task(int ID, int arrivalTime, int serviceTime) {
        this.ID = ID;
        this.arrivalTime = arrivalTime;
        this.serviceTime = serviceTime;
        this.dispatched=false;

    }
    5 usages   Razvan
    public int getID() { return ID; }

    12 usages   Razvan
    public int getArrivalTime() { return arrivalTime; }

    9 usages   Razvan
    public int getServiceTime() { return serviceTime; }
    Razvan
    @Override
    public String toString(){
        return " (" + ID + "," + arrivalTime + "," + serviceTime + ");";
    }
```

Server.java

The **Server** class represents a server in the simulation, capable of processing tasks concurrently. It maintains a queue of tasks and executes them sequentially.

Notable methods:

- The **run** method of the **Server** class, tasks are processed sequentially while ensuring thread safety through synchronization to prevent race conditions when accessing shared resources.
- The **addTask** method allows for the addition of tasks to the server's queue, synchronizing access to ensure thread safety when modifying the task queue.

```
@Override                                                                                    ⚠14 ✓5
public void run() {
    while (active.get() && !Thread.currentThread().isInterrupted()) {
        try {
            Task task = tasks.peek();//nu scoate din array , faci sa iasa doar cand are service time 0
            if (task != null) {

                System.out.println("Server " + id + "  task: (" + task.getID() + "," + task.getArrivalTime(

                Thread.sleep( millis: 1000);
                waitingPeriod.decrementAndGet();

                task.setServiceTime(task.getServiceTime() - 1);

                if (task.getServiceTime() == 0) {
                    tasks.poll();
                    System.out.println("Server " + id + " finished processing task: (" + task.getID() + ","

                }

            }
        } catch (InterruptedException  e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Scheduler.java

The **Scheduler** class manages a collection of servers and implements task dispatching strategies.

Notable methods:

- Its **changeStrategy** method allows for switching between different task dispatching policies based on the provided selection policy.
- The **dispatchTask** method assigns tasks to servers based on the selected strategy.

- The **interruptServers** method interrupts all server threads, ensuring their termination.

```
1 usage   Razvan
public void changeStrategy(SelectionPolicy policy){

    if(policy==SelectionPolicy.SHORTEST_QUEUE){
        strategy=new ShortestQueueStrategy();
    }
    if(policy==SelectionPolicy.SHORTEST_TIME){
        strategy=new ShortestTimeStrategy();
    }
}
```

Strategy.java

The Strategy interface defines a contract for strategies used in task dispatching within the simulation. Implementations of this interface must provide a method **addTask** that specifies how tasks are added to servers based on the strategy.

ShortestTimeStrategy.java & ShortestQueueStrategy.java

In the **ShortestTimeStrategy** class, the **addTask** method selects the server with the shortest estimated remaining time and adds the task to it. This strategy aims to minimize the overall waiting time for tasks by assigning them to servers that are expected to finish processing sooner.

On the other hand, in the **ShortestQueueStrategy** class, the **addTask** method selects the server with the shortest queue (fewest number of tasks in the queue) and adds the task to it. This strategy prioritizes servers with shorter queues, aiming to balance the workload across servers and prevent bottlenecks.

```
1 usage  Razvan
public class ShortestTimeStrategy implements Strategy{
    1 usage  Razvan
    @Override
    public void addTask(List<Server> servers, Task t) throws InterruptedException {
        Server shortestTime = servers.get(0);
        for (Server server : servers) {
            // System.out.println("Server " + server.id + " has time " + server.getEstimatedRemainingTime());
            if (shortestTime.getEstimatedRemainingTime().get() > server.getEstimatedRemainingTime().get()) {
                shortestTime = server;
            }
        }
        shortestTime.addTask(t);
    }
}
```

SimulationManager.java

The **SimulationManager** class orchestrates the simulation of task processing within a server environment.

Notable methods:

- The run() method in the **SimulationManager** class controls the main simulation loop, managing task generation, dispatching, time progression, and metric updates while ensuring thread safety and proper resource cleanup.
- The **generateNRandomTasks()** and **generateHardcodedTasks(),** which generate tasks either randomly or with predetermined values; **updateTxt(),** which updates the text output with the current simulation state; **updateMetrics().**

```java
@Override
public void run() {
    try {
        generateNRandomTasks();
        //generateHardcodedTasks();
        while (currentTime.get() <= timeLimit && !Thread.currentThread().isInterrupted()) {
            System.out.println("Time: " + currentTime);
            for (Task task : generatedTasks) {
                if (currentTime.get() >= task.getArrivalTime() && !task.isDispatched()) {
                    scheduler.dispatchTask(task);
                    task.setDispatched(true);
                    displayGeneratedTasks.remove(task);
                    updateMetrics(task);
                }
            }
            updateTxt();
            Thread.sleep( millis: 1000);
            currentTime.incrementAndGet();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (IOException e) {
        throw new RuntimeException(e);
    } finally {
        scheduler.interruptServers();
        try {
            //logMetrics(); removed this only displaying results in GUI
            logger.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
```

SimulationController.java

The SimulationController class manages the user interface for configuring and running the simulation.

Notable methods:

- The startSimulation() method parses user inputs, creates a SimulationManager instance, and starts a thread to run the simulation.
- The handleButtonQueue() and handleButtonTime() methods update the selection policy based on user input.

```java
//SimulationManager simulationManager=new SimulationManager(Selection
Thread simulationThread=new Thread(simulationManager);
simulationThread.start();
new Thread(() -> {
    while (simulationThread.isAlive()) {
        textArea.appendText(simulationManager.getCurrentOutput());
        try {
            Thread.sleep( millis: 1000); // Update every second
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    String metrics = simulationManager.getMetrics();
    textAreaMetrics.setText(metrics);
}).start();
```

SimulationApplication.java

The **SimulationApplication** class serves as the entry point for the JavaFX application. It loads the FXML file for the user interface, sets up the stage with a title and an icon, and displays the scene to start the application.

```java
public class SimulationApplication extends Application {
    @Override
    public void start(Stage stage) throws IOException {
        FXMLLoader fxmlLoader = new FXMLLoader(SimulationApplication
        Scene scene = new Scene(fxmlLoader.load(), 600, 605);
        stage.setTitle("Queues Management Application");
        String iconPath = "C:\\Users\\danie\\Desktop\\Razvi\\Altele\
        stage.getIcons().add(new Image(iconPath));
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) { launch(); }
}
```

Simulation.fxml

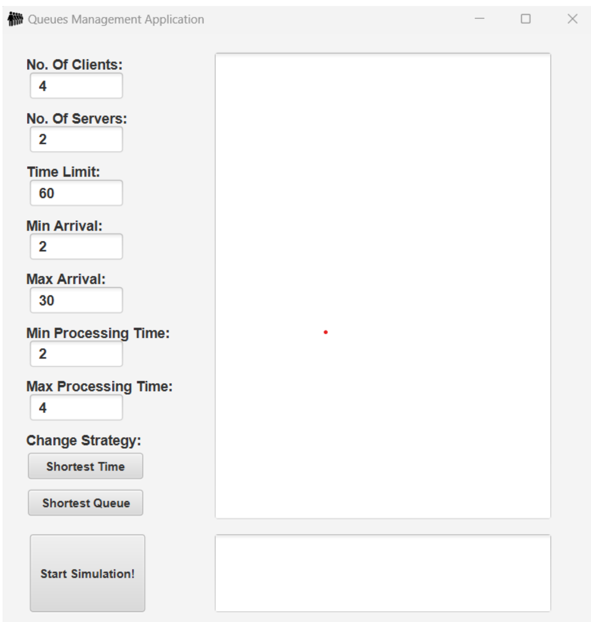
The **FXML** file defines the layout for the user interface of the simulation application. It includes text fields and labels for inputting parameters such as the number of clients, servers, time limit, minimum and maximum arrival times, and minimum and maximum processing times.

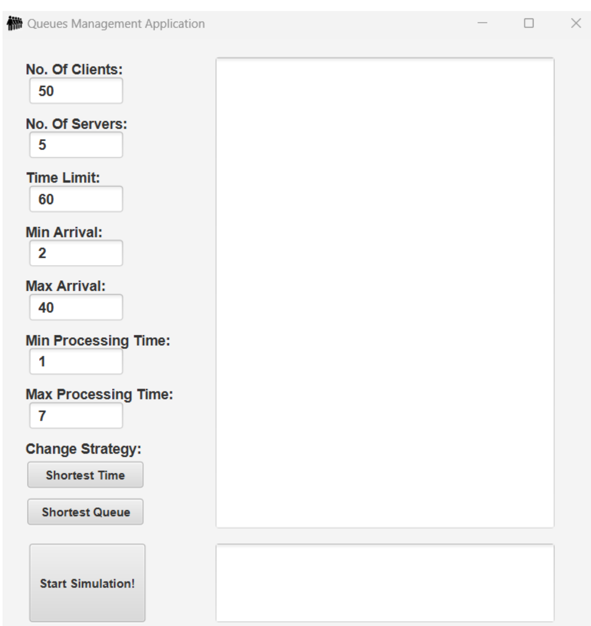
```xml
    </font></Button>
<TextArea fx:id="textArea" layoutX="216.0" layoutY="19.0" prefHeight="469.0" prefWidth="338.0">
    <font>
        <Font name="Arial Bold" size="12.0" />
    </font>
</TextArea>
```

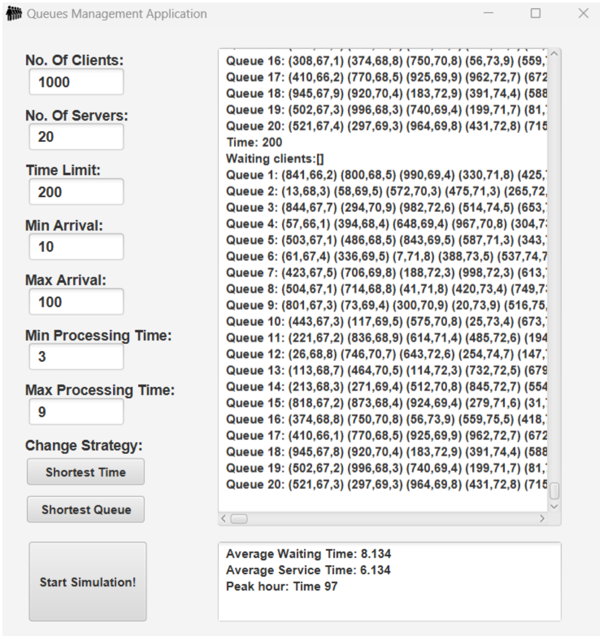
5. Results

I ran the 3 proposed test. The log of events and metrics results are posted on GitLab.

Test Scenarios:







6. Conclusions

Conclusions*:* Designing the simulation revealed synchronization challenges due to coordinating various components. The choice of appropriate data structures, such as **BlockingQueue** and **AtomicInteger**, proved critical for efficient coordination. Modular design, exemplified by the Strategy interface, was very useful for dispatching the clients.

Learnings: Trying threads in Java for the first time presented a significant learning curve, with challenges in understanding and managing concurrent execution. Displaying simulation results posed an additional obstacle, requiring careful synchronization to ensure accurate and real-time updates. These experiences underscored the importance of design and thorough testing to address complexities effectively.

Future Developments: Future plans might involve making the display better by showing queues as blocks and using simple stick figures to show clients waiting and leaving after they're done. Fixing problems with lots of clients and only a few servers will need careful organizing of data and methods to work well. Making sure everything runs smoothly with more clients and fewer servers will be a big focus.

7. Bibliography

- Fundamental Programming Techniques: A2_S1, A2_S2
- OOP – Marius Joldoș
- https://youtu.be/IZCwawKILsk?si=1a1T8uV8iof2JGQE
- https://www.youtube.com/@Randomcode_0
- https://stackoverflow.com/
- https://www.geeksforgeeks.org/