DOCUMENTATION

ASSIGNMENT 2

STUDENT NAME: COMAN ALECSIA-COSTINA

GROUP: 30423

CONTENTS

1. Assignment Objective

Create and develop a simulation tool that evaluates queuing systems to identify and reduce customer wait times. Queues are a familiar sight in both actual scenarios and theoretical models. The primary goal of a queue is to serve as a holding area for customers awaiting service. Queue system managers aim to decrease the duration customers spend in line.

Introducing additional service points, or queues, each with its own server, is one strategy to cut down on wait times. However, this can lead to higher operational costs. When a new service point is added, the waiting customers are redistributed evenly across all available queues.

The software will mimic a sequence of customers arriving for services, joining queues, waiting, receiving services, and then exiting the queue. It will monitor and report the average time customers wait in line.

To determine wait times, it’s essential to know the arrival, service completion, and service duration times. These times vary based on individual customer arrival and their specific service requirements. The completion time is influenced by the queue count, the number of customers already in line, and their service demands.

Input parameters include:

- The duration of the simulation;
- The number of queues;
- The range for customer arrivals;
- The range for service duration;
- The strategy policy: based on time or queue;

The output should include:

- The average waiting time, service duration and peak hour;
- A log detailing how tasks are processed inside the serves over time;
- An illustration of queue dynamics;

2. Problem Analysis, Modeling, Scenarios, Use Cases

- PROBLEM ANALYSYS

The application is designed to simulate the experience of customers queuing for services, such as those found in supermarkets or banks. In these real-world settings, customers must wait their turn in lines, with each queue handling multiple clients at the same time. The core concept of the application is to assess the service capacity within a given timeframe. This is achieved by inputting various parameters into a straightforward and accessible graphical user interface, which allows for easy interaction and manipulation of the simulation variables. The goal is to provide insights into the operational efficiency and to explore different scenarios that could enhance the customer service process.

- MODELLING

In this simulation, customers are created using a random generator, with each assigned a unique service duration and time of arrival, based on predefined input ranges. The simulation allows the user to configure:

- The **maximum number of queues** that can be active for customer processing.
- The **arrival interval range**, specifying the minimum and maximum time between the arrival of customers seeking services, measured in seconds.
- The **service time range**, indicating the minimum and maximum duration, in seconds, that a customer requires to be serviced, selected at random.
- The **simulation interval**, marking the start and end times of the simulation period.

The user can access the following data:

- The **average wait time** for customers in each queue, also in seconds.
- The **busiest period**, denoted as the ‘peak hour’, during which the highest number of customers are served.
- The **aggregate number of customers** who have been serviced throughout the simulation.
- The **mean duration of service** provided, in seconds.
- The **count of customers** and the **mean wait time** during a user-specified interval.

- SCENARIOS

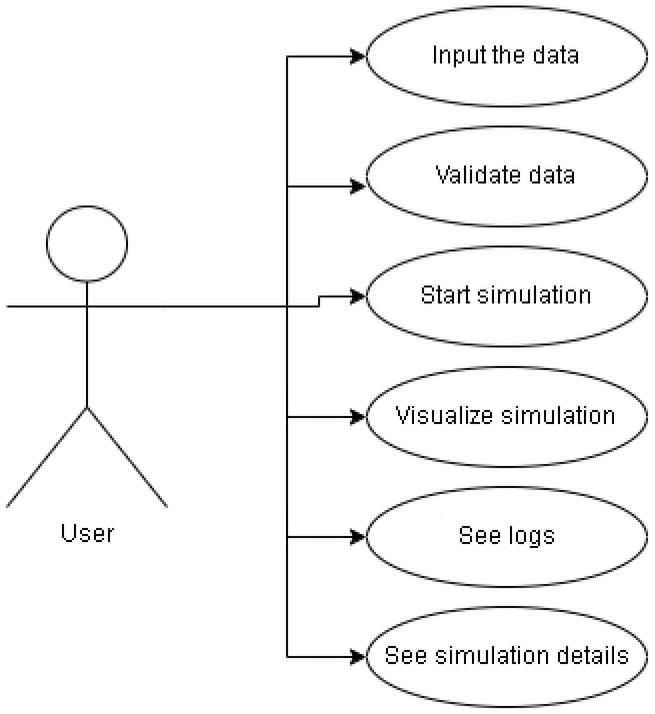
Use Case: simulation setup
Primary Actor: user
Primary Success Path:

1. The user inputs the required figures for the client count, queue count, simulation duration, and the range for both arrival and service times.
2. The user selects the 'validate input data' option.
3. The system checks the data and, if correct, prompts the user to commence the simulation.

Alternate Flow: Incorrect Configuration Data

- The user enters incorrect figures for the setup parameters of the application.
- The system flags an error and instructs the user to provide correct values.
- The process reverts to the initial step.

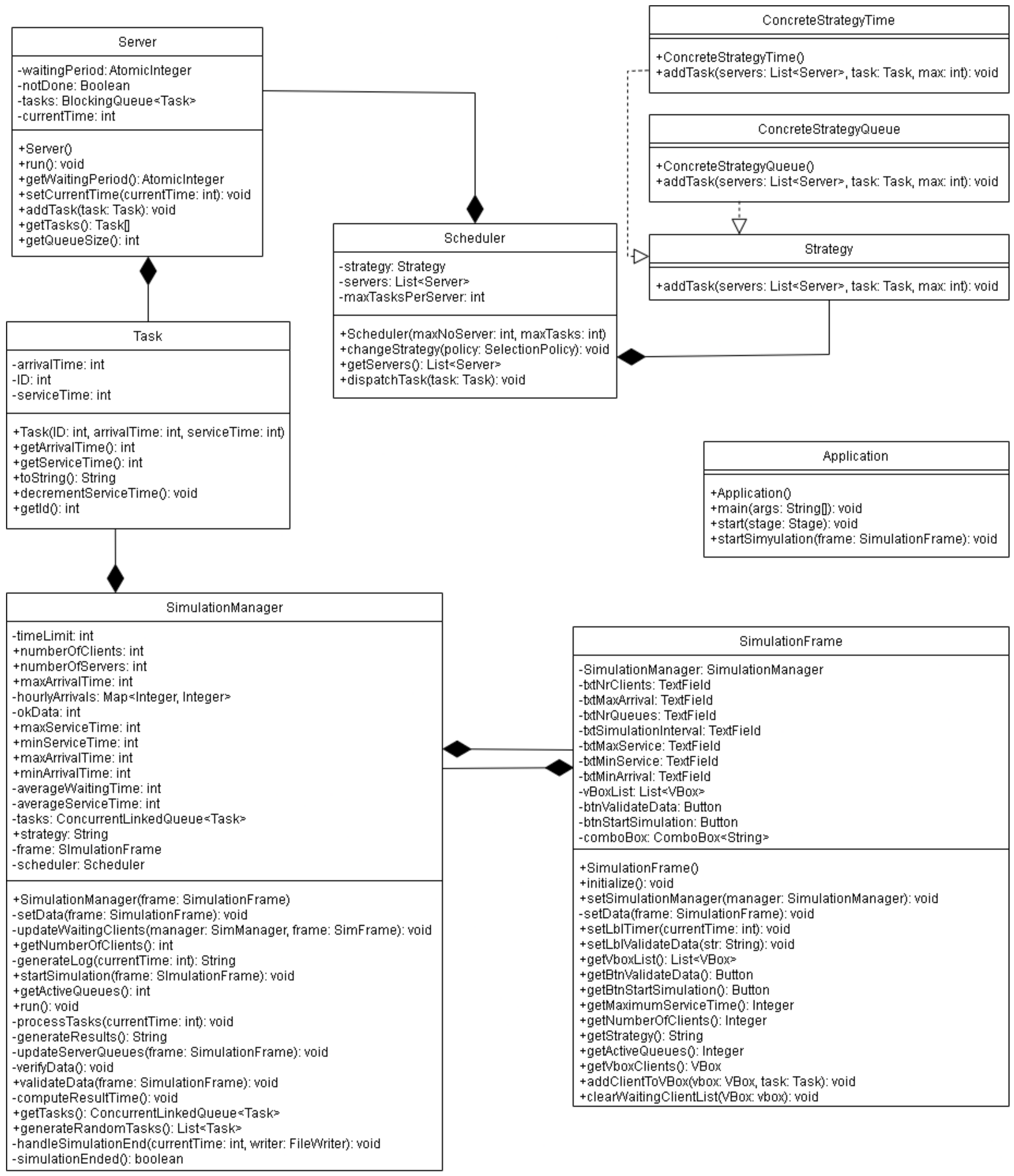
- USE CASE DIAGRAM



The stickman depicted in the diagram symbolizes the actor, which is the user interacting with the application. Connected to the stickman are six ovals representing its potential actions: inputting the data for the simulation, validate it, start the simulation if introduced data is correct, visualize how the queues work, see logs and simulation details.

3. Design

- <u>CLASS DIAGRAM</u>



Server
-waitingPeriod: AtomicInteger
-notDone: Boolean
-tasks: BlockingQueue<Task>
-currentTime: int
+Server()
+run(): void
+getWaitingPeriod(): AtomicInteger
+setCurrentTime(currentTime: int): void
+addTask(task: Task): void
+getTasks(): Task[]
+getQueueSize(): int
ConcreteStrategyTime
+ConcreteStrategyTime()
+addTask(servers: List<Server>, task: Task, max: int): void
ConcreteStrategyQueue
+ConcreteStrategyQueue()
+addTask(servers: List<Server>, task: Task, max: int): void
Strategy
+addTask(servers: List<Server>, task: Task, max: int): void
Scheduler
-strategy: Strategy
-servers: List<Server>
-maxTasksPerServer: int
+Scheduler(maxNoServer: int, maxTasks: int)
+changeStrategy(policy: SelectionPolicy): void
+getServers(): List<Server>
+dispatchTask(task: Task): void
Task
-arrivalTime: int
-ID: int
-serviceTime: int
+Task(ID: int, arrivalTime: int, serviceTime: int)
+getArrivalTime(): int
+getServiceTime(): int
+toString(): String
+decrementServiceTime(): void
+getId(): int
Application
+Application()
+main(args: String[]): void
+start(stage: Stage): void
+startSimyulation(frame: SimulationFrame): void
SimulationManager
-timeLimit: int
+numberOfClients: int
+numberOfServers: int
+maxArrivalTime: int
-hourlyArrivals: Map<Integer, Integer>
-okData: int
+maxServiceTime: int
+minServiceTime: int
+maxArrivalTime: int
+minArrivalTime: int
-averageWaitingTime: int
-averageServiceTime: int
-tasks: ConcurrentLinkedQueue<Task>
+strategy: String
-frame: SImulationFrame
-scheduler: Scheduler
+SimulationManager(frame: SimulationFrame)
-setData(frame: SimulationFrame): void
-updateWaitingClients(manager: SimManager, frame: SimFrame): void
+getNumberOfClients(): int
-generateLog(currentTime: int): String
+startSimulation(frame: SImulationFrame): void
+getActiveQueues(): int
+run(): void
-processTasks(currentTime: int): void
-generateResults(): String
-updateServerQueues(frame: SimulationFrame): void
-verifyData(): void
+validateData(frame: SimulationFrame): void
-computeResultTime(): void
+getTasks(): ConcurrentLinkedQueue<Task>
+generateRandomTasks(): List<Task>
-handleSimulationEnd(currentTime: int, writer: FileWriter): void
-simulationEnded(): boolean
SimulationFrame
-SimulationManager: SimulationManager
-txtNrClients: TextField
-txtMaxArrival: TextField
-txtNrQueues: TextField
-txtSimulationInterval: TextField
-txtMaxService: TextField
-txtMinService: TextField
-txtMinArrival: TextField
-vBoxList: List<VBox>
-btnValidateData: Button
-btnStartSimulation: Button
-comboBox: ComboBox<String>
+SimulationFrame()
+initialize(): void
+setSimulationManager(manager: SimulationManager): void
-setData(frame: SimulationFrame): void
+setLblTimer(currentTime: int): void
+setLblValidateData(str: String): void
+getVboxList(): List<VBox>
+getBtnValidateData(): Button
+getBtnStartSimulation(): Button
+getMaximumServiceTime(): Integer
+getNumberOfClients(): Integer
+getStrategy(): String
+getActiveQueues(): Integer
+getVboxClients(): VBox
+addClientToVBox(vbox: VBox, task: Task): void
+clearWaitingClientList(VBox: vbox): void

- OOP DESIGN

The object-oriented programming (OOP) design of the application follows key principles and concepts that are fundamental to software engineering:

1. **Encapsulation**: Classes encapsulate related data (fields) and behavior (methods) together. For example, in the **Task** class, fields such as **ID**, **arrivalTime**, and **serviceTime** are encapsulated along with methods to access and modify them.
2. **Abstraction**: Abstraction allows hiding the complex implementation details and showing only the necessary features of an object. For instance, the **SimulationManager** class abstracts the details of managing the simulation process from the user interface by providing methods like **validateData()** and **startSimulation()**.
3. **Inheritance**: Inheritance allows creating a new class (subclass) based on an existing class (superclass), inheriting its attributes and methods. While explicit inheritance is not visible in the provided code, it might be used in the wider project structure.
4. **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. In the provided code, polymorphism may be implemented through interfaces like **Runnable**, which is implemented by the **Server** class, enabling different classes to be executed in a threaded manner.
5. **Composition**: Composition involves creating complex objects by combining simpler objects. For example, the **Scheduler** class contains a list of **Server** objects, composing a scheduling system that manages tasks.
6. **Association**: Association represents relationships between classes. In the provided code, associations exist between classes such as **SimulationFrame** and **SimulationManager**, indicating a collaboration or dependency between them.
7. **Dependency Injection**: Dependency Injection is a design pattern where objects are passed their dependencies rather than creating them internally. In the provided code, dependencies like **SimulationFrame** are injected into classes like **SimulationManager** via constructor injection.

These principles collectively facilitate the creation of modular, maintainable, and reusable code by promoting concepts such as code reusability, modularity, and separation of concerns.

- <u>DATA STRUCTURES</u>

These data structures are chosen based on their suitability for concurrent access, thread safety, and performance characteristics required by different parts of the simulation system. They facilitate efficient storage, retrieval, and manipulation of data throughout the project.

Lists:

List<Task>: Used to store tasks generated for the simulation. Implemented in the **generateRandomTasks()** method of the **SimulationManager** class.

List<VBox>: Used to store multiple VBox elements representing queues in the user interface. Initialized and managed in the **SimulationFrame** class.

Queues:

BlockingQueue<Task>: Used in the **Server** class to manage tasks in a thread-safe manner. Specifically, a **LinkedBlockingQueue** implementation is used, ensuring that tasks are handled in a first-in-first-out manner.

ConcurrentLinkedQueue<Task>: Another queue implementation used in the **SimulationManager** class to store tasks. This queue is not thread-safe but is suitable for concurrent access.

Maps:

Map<Integer, Integer>: Used to store hourly arrival data in the **SimulationManager** class. It maps the hour to the number of arrivals during that hour.

Arrays:

Task[]: Used to represent an array of tasks in the **Server** class. The array holds tasks currently in the queue of a server.

Atomic Variables:

AtomicInteger: Used in the **Server** class to manage the waiting period of tasks. It provides atomic operations ensuring thread safety when updating the waiting period.

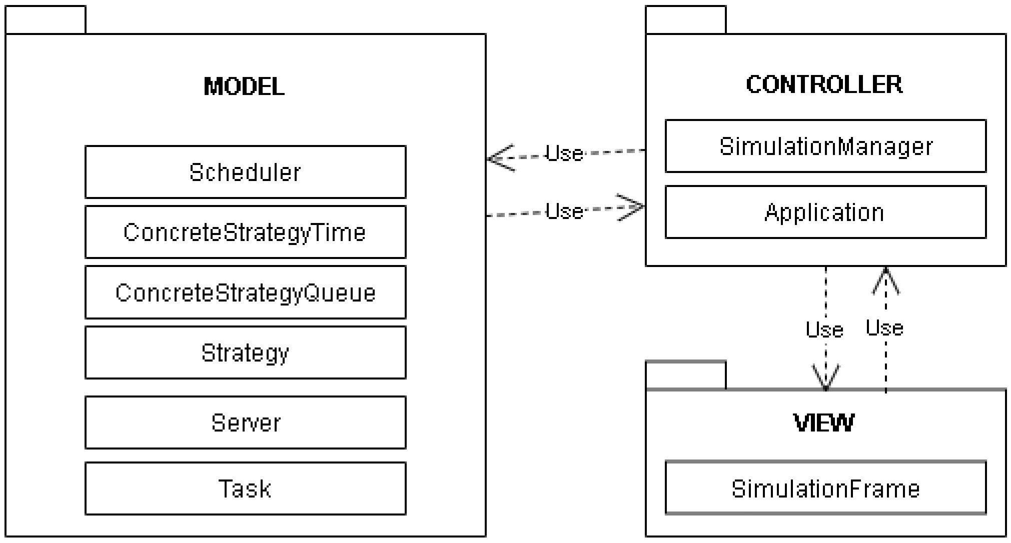
Thread-safe Lists:

CopyOnWriteArrayList<Server>: Used to store the list of servers in the **Scheduler** class. This list implementation allows for concurrent access without the need for explicit synchronization.

- PAKAGES

MVC stands for Model-View-Controller, which is a software design pattern commonly used for developing user interfaces. The methodology aims to separate the concerns of an application into three interconnected components.

Model classes (**Task**, **Server**, **Scheduler**) encapsulate the data and business logic of the simulation. They handle the simulation mechanics and data processing independently of the user interface.
View class (**SimulationFrame**) is responsible for displaying the simulation to the user and capturing user input. It interacts with the user, presenting information and controls.
Controller class (**SimulationManager**) acts as an intermediary between the model and view. It interprets user actions from the view, updates the model accordingly, and updates the view with changes in the model. It also manages the simulation process, coordinating interactions between the model and view components.



4. Implementation

Task:

- Contains data about tasks in the simulation, such as task ID, arrival time, and service time.
- Constructor with parameters to initialize task attributes.
- Usage: Represents individual tasks that need to be processed in the simulation.

Server:

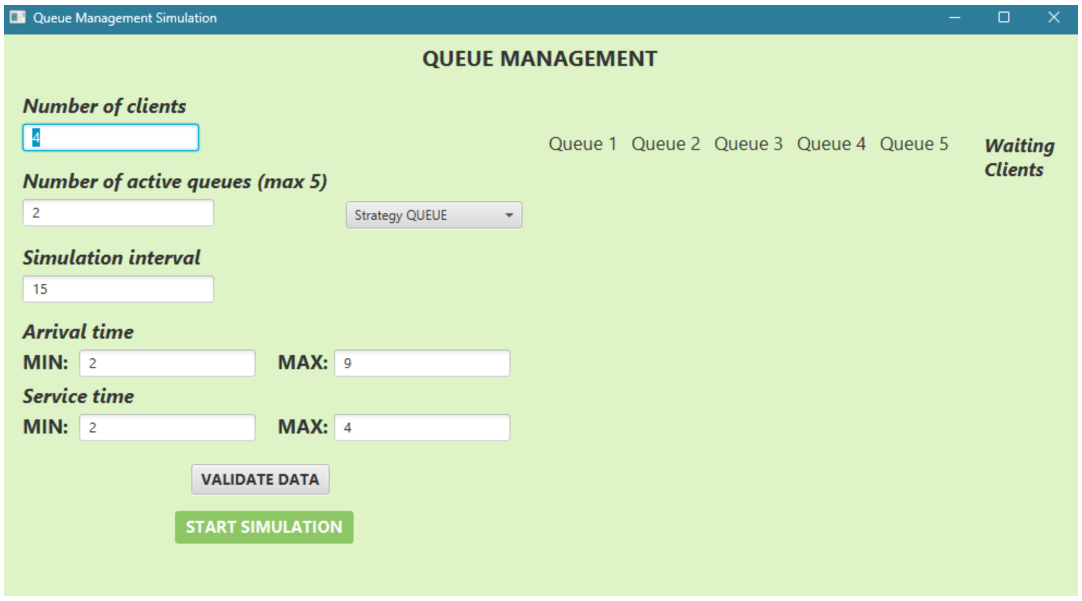
- Implements the **Runnable** interface to represent a server (or queue) in the simulation.
- Attributes include a blocking queue of tasks and an atomic integer for tracking waiting period.
- Methods for adding tasks to the queue, processing tasks, and retrieving queue size.
- Contains a run() method as it acts as a thread itself.
- Usage: Manages tasks within a server queue and simulates task processing.

Scheduler:

- Manages the scheduling of tasks among servers in the simulation.
- Attributes include a list of servers and a strategy for task allocation.
- Methods for changing the scheduling strategy and dispatching tasks to servers.
- Usage: Allocates tasks to servers based on scheduling policies and coordinates task distribution.

SimulationFrame:

- Represents the graphical user interface (GUI) for the simulation.
- Attributes include text fields, labels, buttons, and a combo box for configuring simulation parameters.
- Methods for initializing the GUI components, capturing user input, and updating the interface with simulation data.
- Usage: Interacts directly with the user, providing a visual representation of the simulation and allowing users to control simulation parameter



Number of clients: 4
Number of active queues (max 5): 2
Strategy QUEUE
Simulation interval: 15
Arrival time MIN: 2 MAX: 9
Service time MIN: 2 MAX: 4

SimulationManager:

- Acts as the main controller for the simulation, coordinating interactions between the model (Task, Server, Scheduler) and the view (SimulationFrame).
- Attributes include simulation parameters, a concurrent linked queue of tasks, and a map for tracking hourly arrivals.
- Methods for validating user input, starting and managing the simulation process, updating the view with simulation data, and handling simulation logic.
- Constructor with parameters to initialize simulation parameters and set up the scheduler.
- Usage: Orchestrates the entire simulation process, including initializing parameters, running the simulation loop, and updating the GUI with results.

ConcreteStrategyTime and **ConcreteStrategyQueue**:

- Assigns tasks to servers based on the shortest processing time/shortest queue length/
- Adds tasks to the server with the shortest processing time/shortest queue length and updates total waiting time.

SelectionPolicy:

- Enumerates selection policies for task allocation.
- Includes options for shortest queue and shortest time.

Strategy:

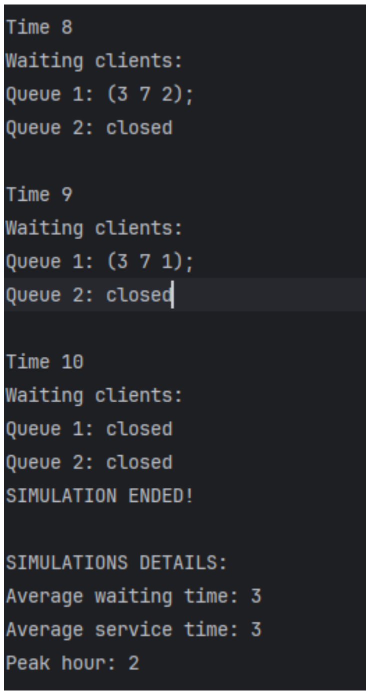
- Interface defining task allocation strategies.
- Contains a method for adding tasks to servers.
- Implemented by concrete strategy classes.

Application:

- Extends **javafx.application.Application**.
- **main()** launches the application.
- **start()** initializes the application:
    - Loads FXML for the simulation view.
    - Sets **SimulationFrame** as the controller.
    - Creates a scene and sets it on the stage.
    - Initializes and sets **SimulationManager** for the controller.
- Provides a method **startSimulation()** to start the simulation in a new thread.

5. Results

Once the user inputs the data and initiates the simulation, they can observe the assignment of each client to a queue and their service progression. Additionally, they can access simulation logs and details such as average waiting time, average service time and peak hour all saved in a text file.



```
Time 8
Waiting clients:
Queue 1: (3 7 2);
Queue 2: closed

Time 9
Waiting clients:
Queue 1: (3 7 1);
Queue 2: closed

Time 10
Waiting clients:
Queue 1: closed
Queue 2: closed
SIMULATION ENDED!

SIMULATIONS DETAILS:
Average waiting time: 3
Average service time: 3
Peak hour: 2
```

6. Conclusions

This project served as a valuable exercise in reinforcing and expanding my understanding of OOP concepts acquired during the first semester. It presented both familiar and new challenges, fostering growth in various aspects.

Firstly, I gained a deeper appreciation for the importance of time management. Effective organization allowed for a gradual progression through tasks, significantly aiding in project completion. Secondly, I learned the significance of accurately modeling the problem from the outset. By establishing a clear and comprehensive model, the implementation process was expedited. Thirdly, I discovered the value of independently tackling coding challenges and troubleshooting issues. This approach not only facilitated problem-solving but also facilitated the acquisition of new concepts and a deeper understanding of existing ones.

Through this project, I also enhanced my understanding of queue processing and its implementation within systems. Additionally, I acquired knowledge about threads and synchronization, further broadening my skill set and understanding of concurrent programming principles.

7. Bibliography

Programming Techniques – Lectures of prof. Cristina POP

Model-View-Controller: https://www.techtarget.com/whatis/definition/model-view-controller-MVC

Diagram: https://www.draw.io/