DOCUMENTATION

ASSIGNMENT *ASSIGNMENT\_2*

STUDENT NAME: Tandea Sorin Darius

GROUP: 30422

Contents

[1. Main Objective: 3](#_Toc133478187)

[2. Analysis: 3](#_Toc133478189)

[Requirements: 3](#_Toc133478190)

[Functional Requirements: 3](#_Toc133478191)

[Non-Functional Requirements: 3](#_Toc133478192)

[Use Case Diagram: 3](#_Toc133478193)

[Use Case: 4](#_Toc133478194)

[3. Design 5](#_Toc133478195)

[3.1. OOP Design of the Application: 5](#_Toc133478196)

[3.2. UML Package and Class Diagrams: 5](#_Toc133478197)

[4. Implementation 7](#_Toc133478198)

[5. Results 10](#_Toc133478199)

[6. Conclusions 10](#_Toc133478200)

[6.1. Lesson learned: 10](#_Toc133478201)

[6.2. Further improvements: 10](#_Toc133478202)

[7. Bibliography 10](#_Toc133478203)

# Main Objective:

Design and implement an application aiming to analyze queuing-based systems by simulating a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues.

## Sub-objectives:

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

# Analysis:

## Requirements:

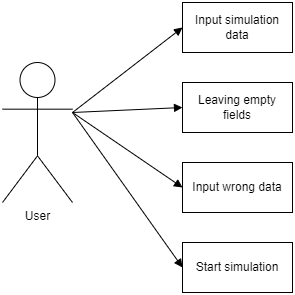
### Functional Requirements:

* The application allows the User to setup the simulation
* The application allows the User to start the simulation
* The evolution of the application should be displayed in real time.

### Non-Functional Requirements:

* Intuitive graphic interface
* Easy to use

## Use Case Diagram:



## Use Case:

Use Case: input simulation data

Primary Actor: User

Precondition: the app is running

Post condition: the simulation is ready to go

Steps:

* The user selects each field he wants to set a value to
* Input a value

Use Case: Empty Fields

Primary Actor: User

Precondition: -

Post condition: The Simulation will start with some predefined conditions

Steps:

* Start the application
* No input is given by the user
* Press Begin Simulation

Use Case: Start Simulation

Primary Actor: User

Post condition: The input fields are filled with valid values

Post condition: The simulation begins

Steps:

* Press the Begin Simulation Button

Use Case: Wrong input data

Precondition: -

Post condition: A Error warning will be shown

Steps:

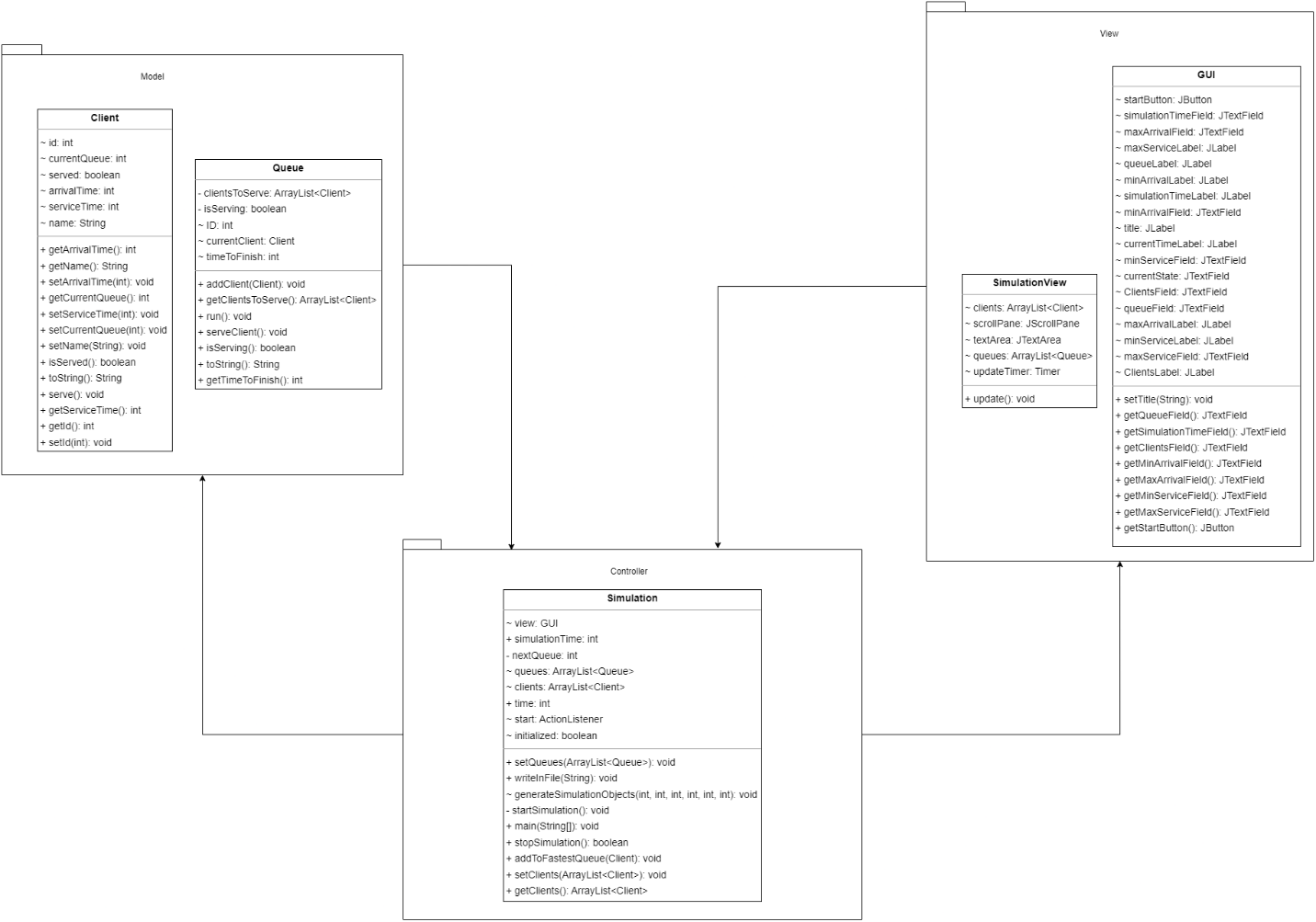
* Start the application
* Input alphabetic characters instead of numbers
* Press Begin Simulation
* A warning pops off

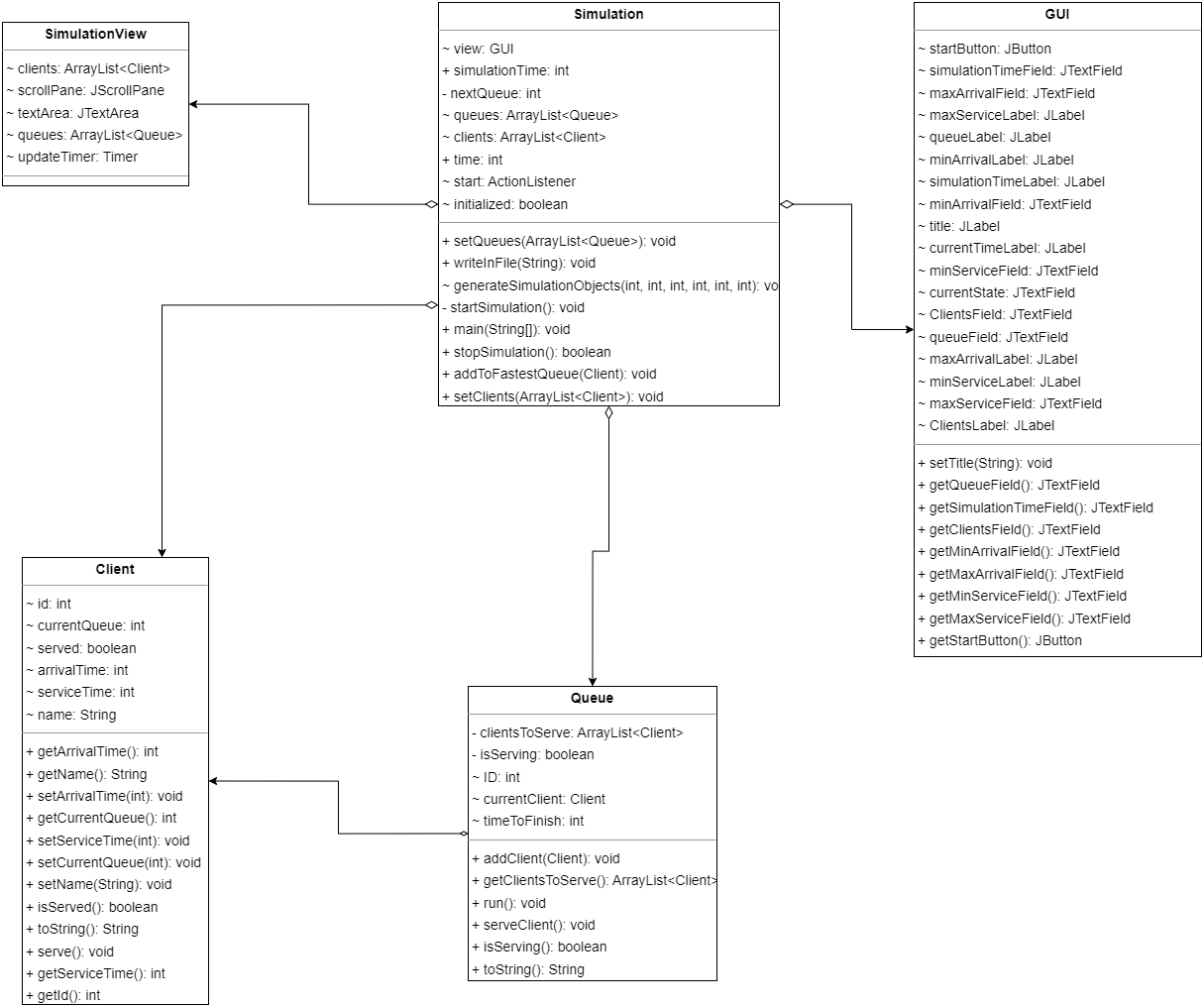
# Design

## OOP Design of the Application:

The application is designed using Object-Oriented Programming (OOP) principles, which include encapsulation, inheritance, and polymorphism. The application is divided into different classes, each focusing on a specific responsibility, making it modular and easy to maintain.

## UML Package and Class Diagrams:





The application consists of three main packages: App.Model, App.Controller, and App.View.

App.Model package:

* Client class
* Queue class

App.Controller package:

* Simulation class

App.View package:

* GUI class
* SimulationView class

Used Data Structures:

ArrayLists: Used to store and manage multiple instances of Client and Queue objects in the Simulation class, and it’s acting as a FIFO data structure.

Used Algorithms:

In the Simulation class, the addToFastestQueue method implements a round-robin algorithm to distribute clients among the available queues.

The generateSimulationObjects method in the Simulation class generates random client arrival and serving times using the provided min and max values.

# Implementation

In this section I will present some details about the implementation of each class

* 1. Queue

This class represents a queue of clients in a simulation. It extends the Thread class to handle multiple queues concurrently.

**Fields:**

isServing: A boolean that indicates whether the queue is currently serving a client or not.

clientsToServe: An ArrayList of Client objects representing clients waiting to be served in the queue.

ID: An integer representing the ID of the queue.

currentClient: A Client object representing the client currently being served.

timeToFinish: An integer representing the time left to finish serving the current client.

**Important methods:**

getTimeToFinish(): Returns the time left to finish serving the current client.

getClientsToServe(): Returns the list of clients waiting to be served in the queue.

isServing(): Returns whether the queue is currently serving a client or not.

addClient(Client client): Adds a client to the queue.

serveClient(): Starts serving the next client in the queue if the queue is not currently serving any client and there are clients waiting.

run(): Overrides the Thread's run() method, handling the serving process of clients in the queue.

toString(): Returns a string representation of the queue, showing its ID and clients waiting to be served.

* 1. Client

This class represents a client in the simulation.

Fields:

id: An integer representing the ID of the client.

arrivalTime: An integer representing the time the client arrives at the queue.

serviceTime: An integer representing the time it takes to serve the client.

name: A String representing the name of the client.

currentQueue: An integer representing the current queue the client is in.

served: A boolean indicating whether the client has been served or not.

Important methods:

Getters and setters for the fields mentioned above.

serve(): Marks the client as served.

toString(): Returns a string representation of the client, showing its ID, arrival time, and service time.

* 1. SimulationView

This class represents the graphical interface for the simulation. It extends JFrame to create a custom window.

Fields:

clients: An ArrayList of Client objects representing the clients in the simulation.

queues: An ArrayList of Queue objects representing the queues in the simulation.

scrollPane: A JScrollPane object to allow scrolling in the text area.

textArea: A JTextArea object to display the simulation state.

updateTimer: A Timer object to update the simulation view every second.

Important methods:

SimulationView(ArrayList<Client> clients, ArrayList<Queue> queues): constructor that initializes the view with the given list of clients and queues.

update(): Updates the text area with the current state of the simulation, including the time, waiting clients, and the state of each queue.

* 1. Simulation

The Simulation class is responsible for controlling the entire simulation process of the queuing system. This class belongs to the package App.Controller. The main components of this class are:

Fields:

view (GUI): an instance of the GUI class for displaying the user interface.

clients (ArrayList<Client>): a list of clients to be served.

queues (ArrayList<Queue>): a list of queues in the system.

time (int): the current time of the simulation.

simulationTime (int): the total time for the simulation.

nextQueue (int): an index used to determine the next queue to add clients to.

Important methods:

Simulation(): constructor that initializes the view and adds a start button ActionListener.

generateSimulationObjects(): generates clients and queues for the simulation.

addToFastestQueue(): adds a client to the next queue in a round-robin fashion.

writeInFile(): writes simulation data to a file.

startSimulation(): starts the simulation by running a new thread and updating the SimulationView.

stopSimulation(): checks if the simulation can be stopped.

* 1. GUI

The GUI class is responsible for creating and displaying the graphical user interface of the application. This class belongs to the package App.View. The main components of this class are:

Fields:

JLabels and JTextFields for various inputs and labels in the user interface.

startButton (JButton): a button to start the simulation.

currentState (JTextField): a field to display the current state of the simulation.

Methods:

GUI(): constructor that initializes the graphical user interface.

Getter and setter methods for the fields.

The GUI class sets up the layout and input fields for the user to provide the necessary information for the simulation. It also displays the current time of the simulation.

# Results

Based on the manual testing done on the software, I observe the application run in 95% of the cases, the majority of the problems are encountered when the number of clients that are arriving at the same time is higher than the number of queues.

# Conclusions

The java application successfully implements the queues and the client generation. In rare occasions It might throw an error regarding synchronization.

## Lesson learned:

* How to synchronize threads
* Why it’s important to use synchronized data structures in your application if you are multithreading

## Further improvements:

* Improve the synchronization, so no error could happen
* Implement a better user interface, but for the moment I think it is ok
* More simulation parameters
* Improve serving time optimization

# Bibliography

<http://inf.ucv.ro/documents/tudori/laborator8_53.pdf>

<https://www.geeksforgeeks.org/java-util-timer-class-java/>

<https://www.baeldung.com/java-synchronized>

<https://www.geeksforgeeks.org/synchronization-in-java/>

<https://stackoverflow.com/questions/2442599/how-to-set-jframe-to-appear-centered-regardless-of-monitor-resolution>