DOCUMENTATION

ASSIGNMENT *2*

QUEUES MANAGEMENT APPLICATION USING

THREADS AND SYNCHRONIZATION MECHANISMS

STUDENT NAME: Vlad Darius Stefan
GROUP: 30226

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 queues, and (2) computing the average waiting time, average service time and peak hour .

Sub-objectives:

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

2. Problem Analysis, Modeling, Scenarios, Use Cases

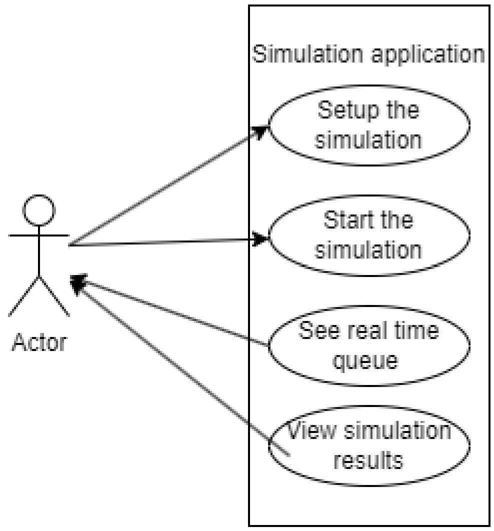
Functional requirements:

- 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
- The simulation application should display average waiting time , average service time and peak hour at the end of the simulation
- The simulation application should verify if the user inputs are valid

Non-functional requirements:

- The simulation application should be intuitive and easy to use by the user
- The simulation should be visually appealing and easy for the user to follow
- The simulation should carefully treat exception cases and not interrupt the flow

Use case diagram:



***Use case:** start simulation

Primary actor: user

Main success scenario:

1. The 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. The users clicks on “start simulation”;
3. The application validates the data and starts the simulation;

Alternative sequence:

1. The user inserts invalid values for the application’s setup parameters;
2. The application displays an error message;
3. The scenario returns to step 1;

***Use case:** see real time queue

Primary actor: user

Main success scenario:

1. The times goes from 1 to simulation interval and for every second it shows the status of the queue;

Alternative sequence:

1.There is no alternative sequence because the data is verified in start simulation;

***Use case:** view simulation results

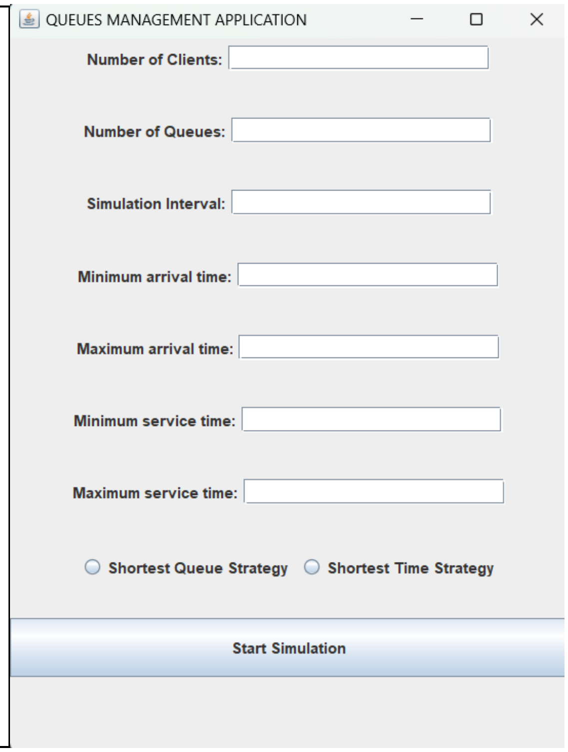
Primary actor : user

Main success scenario:

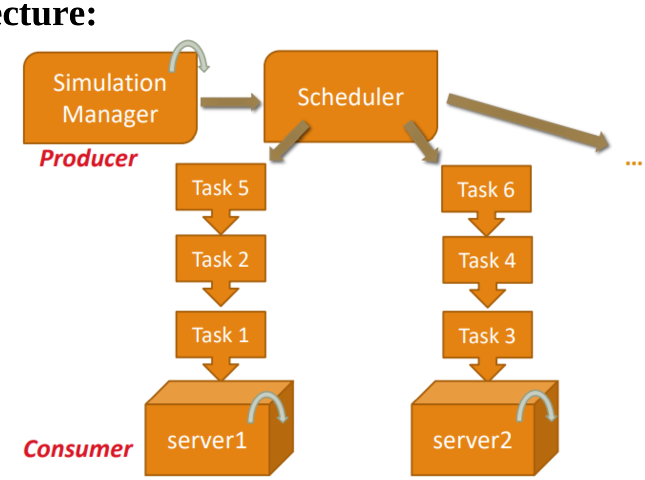
1.After the simulation is finished the user can view different statistics such as average waiting time, peak hour and average service time;

3. Design

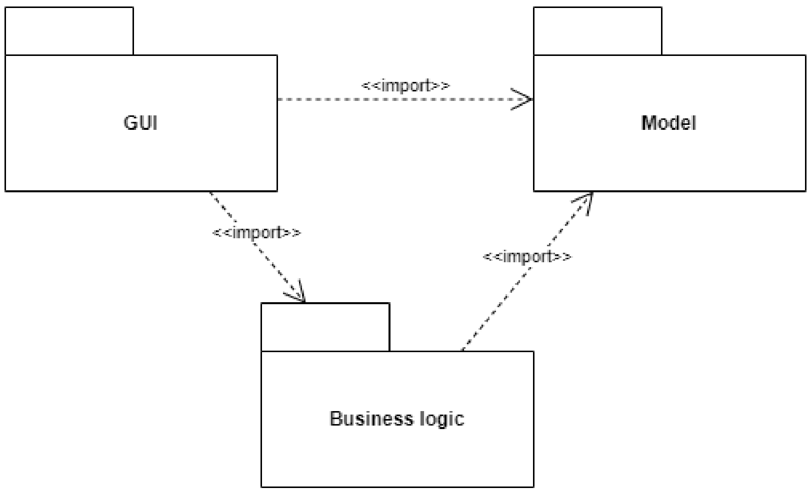
Mockup: **Final Design:**



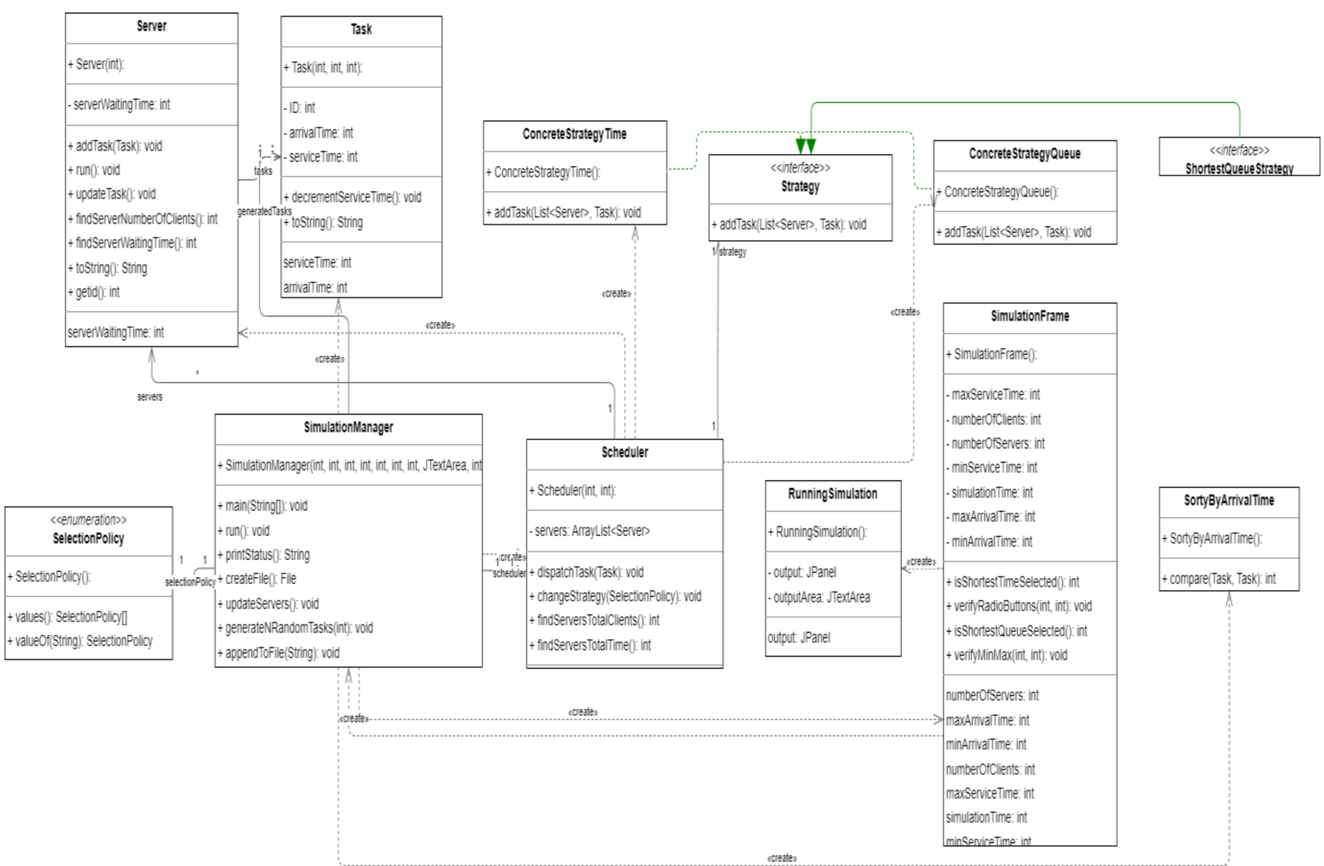
Conceptual architecture:



Package diagram:



UML Class Diagram:



The **GUI** package contains two classes **: SimulationFrame** where the user makes the simulation setup and starts the simulation and **Running Simulation** where the user can see the real time queue simulation.

The **Model** package contains two main classes : **Task** which is the equivalent of a client , having its own ID , arrival time and service time , and **Server** which is the equivalent of a queue , where the clients are processed.

The **Business Logic** package contains 5 main classes which help the simulation run smoothly. We have **ConcreteStrategyQueue** and **ConcreteStrategyTime** , both implementing the **Strategy** interface. In both of these classes , based on the **selection policy** , we decide to which queue should the task be added. We also have the **Scheduler** where we inititate the threads for every queue and dispatch the tasks. The last class is the **SimulationManager** , the main class for this application , where we create the entire simulation and run it.

Used Data Structures:

- ArrayList : to retain the servers in the scheduler;
- BlockingQueue : to retain the tasks in the server in a synchronized way

4. Implementation

Task Class:

-Fields:

- private int id – a unique identifier for every task;
- private int serviceTime – the processing time for every task;
- private int arrivalTime – the arrival time for every task;

-**Methods**:

- public void decrementServiceTime() – we decrement the service time for the task;

Server Class:

-Fields:

- private int id – a unique identifier for every queue;
- BlockingQueue<Task> task – a blocking queue where we put the tasks for every queue , it is a synchronized queue , a thread safe one;
- public AtomicInteger waitingPeriod – the waiting period for each queue;
- private int server waiting time – total waiting time for the server;

-Methods:

- public void addTask(Task newTask) – adds a task to this server and also increments the waiting time for the server;
- public void updateTask() – it checks if the service time for the task at the front of the queue is zero , and if it is zero it removes the task from the queue;

- public void run() – we put the thread to sleep for 1 second and also decrement the waiting time;
- public int findServerWaitingTime() – find the total server waiting time;

ConcreteStrategyQueue Class:

-Fields:

- No fields;

-Methods:

- public void addTask(List<Server> servers , Task t) – it goes through every queue and gets its size and then it puts the task in the server with the smallest size;

ConcreteStrategyTime Class:

-Fields:

- No fields;

-Methods:

- public void addTask(List<Server> servers , Task t) – it goes through every queue and gets the total service time for every queue and then it puts the task in the server with the lowest service time;

Scheduler Class:

-Fields:

- ArrayList<Server> servers – an array where every server is saved;
- int maxNoServers – the maximum number of servers;
- int maxTasksPerServer – the maximum number of tasks that the server can handle(I decided to put the total task number);

-Methods:

- public Scheduler(int maxNoServers , int maxTasksPerServer) – a constructor where we create maxNoServer servers and initiate a thread for every server
- public void changeStrategy(SelectionPolicy policy) – we choose what strategy we want to use for putting the tasks into the server based on an input from the simulationFrame
- public void dispatchTask(Task t) – we use the addTask method from the strategy;

SortByArrivalTime Class:

-Fields:

- no fields;

-Methods:

- public int compare(Task a , Task b) - returns a number > 0 if a is greater than b , a number equal to zero is task a is equal to task b , and a number < 0 if a is smaller than b

Simulation Manager Class:

-Fields:

- public int timeLimit – simulation time limit;
- public int minProcessingTime and public int maxProcessingTime – the boundaries for the service time;
- public int minArrivalTime and public int maxArrivalTime – the boundaries for the arrival time;
- public int numberOfClients – total number of tasks;
- public int numberOfServers – total number of servers;
- public SelectionPolicy selectionPolicy – how we select the queue for the task;
- private Scheduler scheduler – a scheduler that contains the servers;
- private JTextArea output – where the whole simulation will be shown;
- private List<Task> generatedTask – a list where we save the random generated tasks;

-Methods:

- public SimulationManager(...) – a constructor for our simulation manager where we instantiate every field and also generate numberOfClients random clients;
- public void generateNRandomTasks(int n) – we generate random arrival times and service times for n tasks and we also at them to the generatedTask list;
- public void run() – we use currentTime to go through the simulation , second by second and we print the status for the simulation for every step and also update the queues;
- public File createFile() – we use this to create a file where we show the output;
- public void appendToFile(String message) – we append to the end of the file;

SimulationFrame Class:

-Fields:

- A JLabel and a JTextField for every necessary input for the simulation setup;
- A JButon to start the simulation;
- Two JRadioButtons for the selection policy;

-Methods:

- getters for every input;
- public void verifyRadioButtons() – we verify if one radio button is pressed;
- public void verifyMinMax() – we verify if the min inputs are smaller than the max inputs;

RunningSimulation Class:

-Fields:

- private JPanel output
- private JTextArea outputArea – where we show the simulation;

-Methods:

- getters for the panels;

Time Strategy Implementation:

```java
1 usage
public class ConcreteStrategyTime implements Strategy {

    1 usage
    @Override
    public void addTask(List<Server> servers, Task t) {

        int minServerTime = Integer.MAX_VALUE-1;
        int time;

        for (Server server : servers) {
            if (server == null)
                time = 0;
            else
                time = server.findServerWaitingTime();
            if (time < minServerTime) {
                minServerTime = time;

            }
        }

        for (Server server : servers) {
            if (server.findServerWaitingTime() == minServerTime) {
                server.addTask(t);
                break;
            }
        }
    }
```

Queue Strategy Implementation:

```java
public class ConcreteStrategyQueue implements Strategy {

    1 usage
    public void addTask(List<Server> servers, Task t) {
        int numberOfClients;
        int minNumberOfClients = Integer.MAX_VALUE-1;
        for (Server server : servers) {
            if(server == null)
                numberOfClients = 0;
            else
             numberOfClients = server.findServerNumberOfClients();
            if (numberOfClients < minNumberOfClients) {
                minNumberOfClients = numberOfClients;
            }
        }

        for (Server server : s
            if (server.findServer                                  ) {
                System.out.println("Clientul " + t.getID() + " a intrat in coada " + server.getid());
                server.addTask(t);
                break;

            }
        }
    }
```

```
Reassigned local variable
int minNumberOfClients = Integer.MAX_VALUE-1
PT2024_30226_Vlad_Darius_Assignment_2
```

5. Results

Tested on 3 different cases with simulation interval from 60 to 200 seconds and number of clients from 4 to 1000. The log files for each simulation will be included in the git repository.

6. Conclusions

The conclusion of this assignment is the creation of a fully functional **queues management system using threads and synchronization mechanisms**.

From this homework I have learned how to properly work with threads and how to use **synchronization mechanisms**. It was also the first time working with **BlockingQueue** or **AtomicInteger** , but with a little research I understood them very easy. It was also the first time working with **Files** in java.

For the **future developments** , the queues management system could be made even more efficient , could print out more statistics and maybe find different selection patterns to make the queues as efficient as possible.

7. Bibliography

1. https://dsrl.eu/courses/pt/
2. https://users.utcluj.ro/~igiosan/teaching_poo.html
3. https://www.programiz.com/c-programming/c-file-input-output#google_vignette
4. https://www.geeksforgeeks.org/java-swing-popup-and-popupfactory-with-examples/
5. https://www.geeksforgeeks.org/user-defined-custom-exception-in-java/
6. https://www.baeldung.com/java-streams-peek-api