DOCUMENTATION

ASSIGNMENT *2*

STUDENT NAME: Urdea Mara-Cristina
GROUP: 30422

CONTENTS

1. Assignment Objective

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

Queues are commonly used to model real world domains. The main objective of a queue is to provide a place for a "client" to wait before receiving a "service". The management of queue-based systems is interested in minimizing the time amount their "clients" are waiting in queues before they are served. One way to minimize the waiting time is to add more servers, i.e., more queues in the system (each queue is considered as having an associated processor) but this approach increases the costs of the service supplier. The queues management application should simulate (by defining a simulation time $tsimulation$) a series of N clients arriving for service, entering Q queues, waiting, being served and finally leaving the queues. All clients are generated when the simulation is started, and are characterized by three parameters: ID (a number between 1 and N), $t_{arrival}$ (simulation time when they are ready to enter the queue) and $t_{service}$ (time interval or duration needed to serve the client; i.e. waiting time when the client is in front of the queue). The application tracks the total time spent by every client in the queues and computes the average waiting time. Each client is added to the queue with the minimum waiting time when its $t_{arrival}$ time is greater than or equal to the simulation time ($t_{arrival} \geq t_{simulation}$).

The following data should be considered as input data for the application that should be inserted by the user in the application's user interface:
- Number of clients (N);
- Number of queues (Q);
- Simulation interval ($t_{simulation}{}^{MAX}$);
- Minimum and maximum arrival time ($t_{arrival}{}^{MIN} \leq t_{arrival} \leq t_{arrival}{}^{MAX}$);
- Minimum and maximum service time ($t_{service}{}^{MIN} \leq t_{service} \leq t_{service}{}^{MAX}$);

2. Problem Analysis, Modeling, Scenarios, Use Cases

Problem Analysis

The problem addressed is to efficiently manage a system of queues that assign clients to minimize their waiting time while also minimizing the costs of the service supplier. The application simulates a series of N clients arriving for service, entering Q queues, waiting, being served, and leaving the queues. The clients are characterized by three parameters: ID, arrival time, and service time. The application tracks the total time spent by every client in the queues and computes the average waiting time. The input data for the application includes the number of clients, number of queues, simulation interval, and minimum and maximum arrival and service times. The solution to this problem requires the implementation of a simulation that effectively manages the arrival, queuing, and servicing of clients while minimizing their wait time. To achieve this, the application will need to use threads and synchronization mechanisms to manage the flow of clients through the system of queues.

Modelling the problem

To model the Queues Management Application, we need to define the entities, events, and processes involved in the system. This includes generating clients with unique IDs, arrival times, and service times, assigning them to the shortest queue, processing them through the queue, and tracking their total time spent and average waiting time. We'll use threads and synchronization mechanisms to ensure an accurate simulation and efficient queue management.

Use cases and scenario

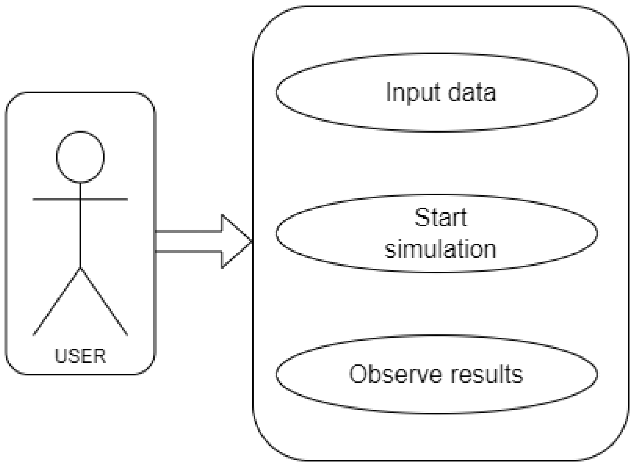
The main use case for the Queues Management Application is to help service providers manage client queues in a way that minimizes waiting time. The scenario for the application involves a series of clients arriving for service, entering queues, waiting, being served, and then leaving. The application tracks the total time spent by each client in the queue and computes the average waiting time. The user inputs the necessary data, such as the number of clients, queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time. The application then simulates the queue management process and displays the results.

3. Design

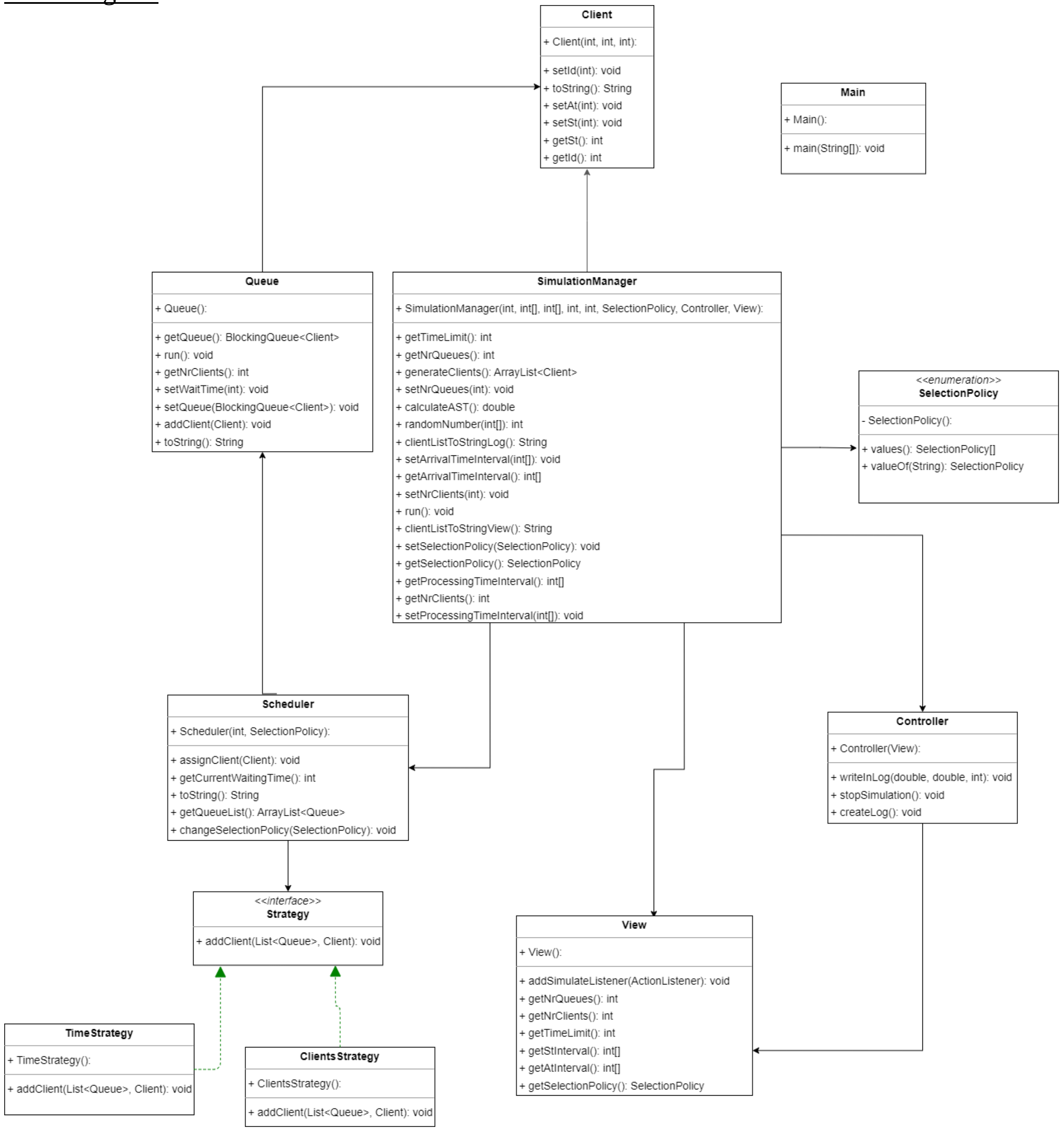
The Queues Management Application is designed using object-oriented programming (OOP) principles. The application is divided into several packages, each representing a different aspect of the application's functionality, such as user interface, client management, and queue management. The class diagram shows the relationships between the different classes, such as the Client class, Queue class etc.

The application's user interface is designed to be intuitive and easy to use. The user can input the necessary data, such as the number of clients, queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time. The application then simulates the queue management process and displays the results, such as the average waiting time.

User diagram



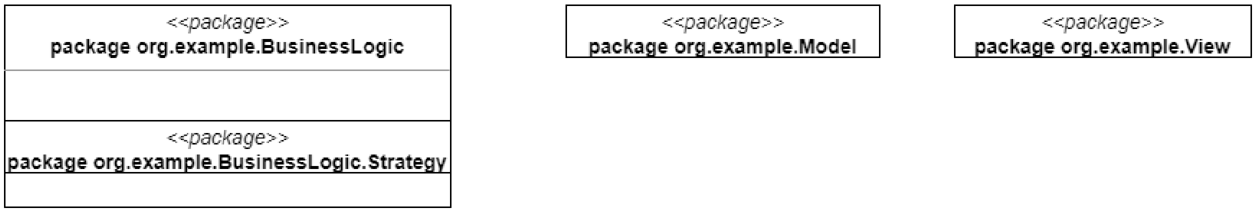
Class diagram


Client
+ Client(int, int, int):
+ setId(int): void
+ toString(): String
+ setAt(int): void
+ setSt(int): void
+ getSt(): int
+ getId(): int
Main
+ Main():
+ main(String[]): void
Queue
+ Queue():
+ getQueue(): BlockingQueue<Client>
+ run(): void
+ getNrClients(): int
+ setWaitTime(int): void
+ setQueue(BlockingQueue<Client>): void
+ addClient(Client): void
+ toString(): String
SimulationManager
+ SimulationManager(int, int[], int[], int, int, SelectionPolicy, Controller, View):
+ getTimeLimit(): int
+ getNrQueues(): int
+ generateClients(): ArrayList<Client>
+ setNrQueues(int): void
+ calculateAST(): double
+ randomNumber(int[]): int
+ clientListToStringLog(): String
+ setArrivalTimeInterval(int[]): void
+ getArrivalTimeInterval(): int[]
+ setNrClients(int): void
+ run(): void
+ clientListToStringView(): String
+ setSelectionPolicy(SelectionPolicy): void
+ getSelectionPolicy(): SelectionPolicy
+ getProcessingTimeInterval(): int[]
+ getNrClients(): int
+ setProcessingTimeInterval(int[]): void
<<enumeration>>
SelectionPolicy
- SelectionPolicy():
+ values(): SelectionPolicy[]
+ valueOf(String): SelectionPolicy
Scheduler
+ Scheduler(int, SelectionPolicy):
+ assignClient(Client): void
+ getCurrentWaitingTime(): int
+ toString(): String
+ getQueueList(): ArrayList<Queue>
+ changeSelectionPolicy(SelectionPolicy): void
Controller
+ Controller(View):
+ writeInLog(double, double, int): void
+ stopSimulation(): void
+ createLog(): void
<<interface>>
Strategy
+ addClient(List<Queue>, Client): void
View
+ View():
+ addSimulateListener(ActionListener): void
+ getNrQueues(): int
+ getNrClients(): int
+ getTimeLimit(): int
+ getStInterval(): int[]
+ getAtInterval(): int[]
+ getSelectionPolicy(): SelectionPolicy
TimeStrategy
+ TimeStrategy():
+ addClient(List<Queue>, Client): void
ClientsStrategy
+ ClientsStrategy():
+ addClient(List<Queue>, Client): void

The UML, Unified Modelling Language, diagram presented above is a visual representation used to illustrate the software system. It includes the Java classes used to implement the project and their relationships and dependencies.

Package diagram

The purpose of Java packages is to group related classes together for easy organization. In Object-Oriented Programming (OOP), the Model-View-Controller (MVC) pattern is a common approach to designing projects for optimal efficiency. This pattern is widely used in most of the programming languages. In my project, a modified version of the MVC pattern was utilized, with the model and view components remaining the same, but the controller section being altered into a package named BusinessLogic containing a package named Strategy, for a better organization.


<<package>>
package org.example.BusinessLogic
<<package>>
package org.example.BusinessLogic.Strategy
<<package>>
package org.example.Model
<<package>>
package org.example.View

The project is made up of the Model, View, and BusinessLogic as its core parts. The logical organization of the project and the high-level classes connected to it are under the control of the Model. All user interactions are handled by the View and all the algorithms used to carry out the operations and determine the results are included in the BusinessLogic package.

- The Main is not technically a package, but it can be thought of as one. The main component of the application is located here and serves as the starting point.
- The Client and Queue classes can be found in the Model package.
- The View package contains the Interface, which includes an interactive class where users can enter the time components, number of queues, number of clients and view the process.
- The BusinessLogic package only has the package Strategy and the Controller, Scheduler, SelectionPolicy and SimulationManager classes; it includes all of the methods required to perform the operations.

4. Implementation

Class 'Client':
- Has attributes such as the client's id (unique identifier), arrival time, and service time.
- Has a basic constructor, getters and setters.
- toString() function used for displaying in logs/interface.

Class 'Queue':
- This class represents an actual queue used in the project.
- Since the project uses a separate thread for each queue, our class needs to implement the Runnable interface.

- To store clients in memory, we use a synchronized/concurrent structure called BlockingQueue, which functions like a normal queue.
- The class also has an attribute "waitingTime" which represents the total waiting time of the queue at any given moment.
- run() function is called at the start of the program. It decrements the service time of the client at the head of the queue by 1 each second until it reaches 0, then extracts the client from the queue.
- The addClient() function is called in the scheduler to add clients to the queues. It adds the client to the BlockingQueue and increments the total waiting time by the service time of the added client.
- getters and setters.
- toString() function displays a "I" for each client in the queue, used only for the aesthetics of the graphical interface.

Class "Scheduler":

- This class is used for assigning clients to queues based on selection policy.
- It contains a list of all queues used in the program.
- The constructor takes the total number of queues and initializes each queue from 0 to the total number of queues -1. It also starts a thread for each queue and the selection strategy.
- The function changeSelectionPolicy() changes the selection policy.
- The function calculateAWT() is used in SimulationManager to calculate the average wait time of queues.
- The function addClient() assigns clients to queues based on the selection policy.
- The toString() function forms an elegant string for displaying it in the event log.

Package 'Strategy':

- It has an interface called "Strategy" with the method "addClient", which adds clients to queues.
- The "ClientsStrategy" class represents the strategy of adding clients to queues based on the smallest number of clients. The client is always added to the first queue, and then the queues are sorted according to the number of clients, so that the queue with the smallest number of clients is in the first position in the queue list.
- The "TimeStrategy" class represents the strategy of adding clients to queues based on the shortest waiting time of queues. The client is always added to the first queue, and then the queues are sorted according to the waiting time, so that the queue with the shortest waiting time is in the first position in the queue list.

Class 'SimulationManager':

- It has a scheduler object that controls the dispersion of clients in queues, a randomly generated list of clients, a controller, and a view that communicates with each other to make things work smoothly
- I generate random numbers from the required intervals using Math.random()
- The run() function is quite complex: we have a loop that goes from 0 to the time limit we set. At each moment, we check the client list for clients with an arrival time equal to the current time, add them to the queues, and remove them from the client list. Of course, at

each moment, we update the interface, write the information to the EventLog, and calculate the new averageWaitingTime and peakHour, if applicable.

Class 'View':

- Includes all the elements that make up the JFrame displayed on the screen (buttons, labels, text fields).
- I created a JPanel in which I arranged all the elements, and then add it to the JFrame.
- The buttons and labels use a different font size for aesthetics
- It has all the methods that add ActionListeners for each button
- The updateView() function retrieves all current simulation information and displays it in the interface (current time, clients still outside queues, clients in queues)
- Clients in queues are represented by a "(I)", which should be a person
- We also have a list of all generated clients that empties as their arrival time equals the current time

Class "Controller":

- Connects the buttons and their functionalities
- Includes all classes corresponding to the ActionListener of the buttons
- Retrieves all the necessary information from the interface to start the simulation, such as simulation time, client arrival time interval, client service time interval, total number of queues, total number of clients, and selection strategy

Class "Main":

- Test of the functionality of queues.
- The main objects 'view' and 'controller'.

5. Results

I completed the task where I was asked to create an average waiting time, an average service time, and a peak hour for each simulation. The average waiting time is calculated in the scheduler. The list of queues is iterated, and the waiting time from each queue is taken. Then, the waiting time is divided by the total number of queues. The average waiting time is calculated at each time step, so it is important to print the result obtained from the scheduler at the current time.

The average service time is calculated by iterating through the sorted list of clients and taking the arithmetic mean of the service time attribute of each client.

The peak hour is the peak time of the simulation and is calculated at each time step. We keep track of the maximum number of clients in the queues (or the maximum waiting time), and at each time step, we compare the current state with the maximum state kept in mind. If the result is greater, then 'peakHour' receives the current time.

6. Conclusions

This assignment has allowed me to gain a solid understanding of the usage of threads, which I previously struggled to comprehend. In addition, through the development of this application, I have honed my Java programming skills and gained experience in building a robust application.

As a result of this experience, I have come to the realization that encountering and troubleshooting issues with code on your own, while conducting research to resolve them, leads to an improved understanding of both new and existing concepts.

7. Bibliography

https://stackoverflow.com/
https://www.youtube.com/
https://www.geeksforgeeks.org/
https://dsrl.eu/courses/pt/