DOCUMENTATION

ASSIGMENT NUMBER 2

**-Programming Techniques-**

Queue Management

STUDENT NAME: Tat Teodora

GROUP: 30423

# CONTENT

1. Assignment Objectives
2. Problem analysis, modelling, scenarios, use cases
3. Design
4. Implementation
5. Results and testing
6. Conclusions
7. Bibliography
8. **Assigment Objectives**
   1. **Main objective**

The main objective of this theme is to implement an application that simulates the assignment of customers to a certain number of queues, so that the waiting time is minimized as much as possible.

* 1. **Secondary objective**

The implementation of the application was aimed at meeting some secondary objectives, which will be presented in the following chapters, as follows:

**Chapter II** - Correct implementation of as many different use cases as possible, starting from the main objective: A use case is a methodology used in the analysis of a problem to identify, clarify and organize the requirements to be implemented by the developer. The use-case consists of a set of possible sequences of interactions between applications and users in a specific environment and related to a specific purpose.

**Chapter III** - Organize the code into different classes and packages so that it is as easy to understand as possible: An M-V-C (Model - View - Controller) structure was used to create the GUI graphical interface. Each class contains specific methods, generally implemented in less than 30 lines, to ensure code readability. Classes have less than 300 lines, allowing for better content organization.

**Chapter III** - Use of appropriate data structures so that the security of the threads is ensured: The data structures used within each class and how they were chosen to facilitate threading will be described.

**Chapter III** - Use appropriate algorithms to implement the required functionalities: The algorithms used in the implementation of the methods will be described.

**Chapter IV** - Solution implementation: Each class will be presented separately, with the fields and methods specifically used.

**Chapter V** - Testing the solution in different scenarios: The scenarios that were addressed during the testing will be described.

1. **Problem analysis, modelling, scenarios, use cases**

In the following lines, the formalized functional requirements framework and use cases will be presented, as well as diagrams and descriptions of use-cases. The descriptions of the use-cases will be made in the form of a flow-chart.

In order for the application to assign customers to different queues, it is necessary for the user to enter valid data for the simulation from the simulation window that will be displayed on the screen. The data entered must be integers, with minArrivalTime less than maxArrivalTime and minServiceTime less than maxServiceTime. The user also has the option to choose the strategy by which customers are added to the queue, either based on the waiting time or the number of existing customers already queued.

After pressing the start button called "Simulate", the correctness of the numbers provided by the user is checked. If the data entered is invalid, a frame with the message "Invalid input" will appear on the screen, and the user will have to re-enter the data. If the values entered are correct, a new window will open, called "Simulation Frame", which will simulate in real time the arrival of customers, queuing them, reducing the serving time and finally eliminating customers from the queue.

Graphical user interface, application

Description automatically generated

In the second figure, you can see the “Simulation Frame” window, in which, every second, the list of customers waiting at that moment is displayed, as well as the existing customers at different queues. If a queue is free, then "<closed>" appears next to it. Clients are represented in the form (ID, remaining service time and the arrival time), where the ID and ArrivalTime will remain unchanged from the beginning to the end of the simulation, while ServiceTime changes as the client is processed. It is also shown the statistics below the end of the simulation.

Graphical user interface, text

Description automatically generated

The functionality of the interface is described in a very simple way by the flow chart below:

Shape

Description automatically generated

1. **Design**

In this chapter, we will present the OOP design of the application, the UML diagram, as well as the data structures and algorithms used.

To create the graphical interface I chose to use the JavaFX tool, and the test results are written in a file.

The application is organized in several different packages, which respect the structure M-V-C (Model - View - Controller). The "Application" package contains the "App" class, which contains the main method, through which the execution of the program starts.

"Model" package is also the most important package of the program, because it contains the basic classes: "Server" which prefigures a queue and "Task" used to represent customers.

Diagram

Description automatically generated

The "GUI" package contains the "QueueManagementSystemController" and "SimulationFrameController" classes that link Model and View (controlls the interface). The Views are also represented by the FXML tabs "queue" and "simulationFrame" through which the two windows in the graphical interface are created.

Chart, box and whisker chart

Description automatically generated

The BussinessLogic package also contains most classes and implements the operations performed on the queues.

As data structures, we chose to use collections that provide the concept of thread security such as BlockingQueue, Array BlockingQueue or CopyOnWriteArrayList. The AtomicInteger data type is also used for the same reasons mentioned above. In the Server class, the list of tasks on that server at a given time is stored in a BlockingQueue queue.

In terms of customer deployment strategies, there are two types: ConcreteStrategyQueue, which distributes tasks to the freest queue, and ConcreteStrategyTime, which sends customers to the queue with the shortest waiting time at that time. These 2 classes implement the Strategy interface, with the addTask method.

Diagram

Description automatically generated

1. **Implementation**

This chapter will describe each class with important fields and methods, as well as the implementation of the user interface.

**Task Class**

Stores customer data. such as ID, arrival time, but serving time (how long it must be first in line until it is processed).

**Server Class**

Model data about a queue containing multiple clients (tasks). It has as fields a structure of type BlockingQueue <Task>, but also a variable of type AtomicInteger that retains the waiting period corresponding to the respective queue. Each queue also has a unique ID associated with it to make it easier to identify. At the same time, each queue corresponds to a thread that has the behavior described by the run () method, specific to the Runnable interface, implemented by this class.

Thus, in the run method, the tasks in the queue are processed one by one, and every second, the serving time for the first customer is decremented, until it reaches zero. We can see that, in total, the thread corresponding to the current Server is inactive (sleep) a time interval equal to the serviceTime of the first task in the queue. After this interval, the customer is removed from the queue, which means that it has been served.

The addTask () method deals with adding a client to the task list corresponding to the queue, which also helps to update the waitingTime variable.

The diagram below illustrates the organization of tasks in a BlockingQueue structure, as well as the order of their processing within the queue (we notice that the First In First Out principle is respected).

Diagram

Description automatically generated

**Scheduler class**

This class sends the tasks to the Server, depending on the strategy chosen by the user. It contains data such as the list of existing servers, the number of servers, but also the number of tasks on a server.

The most important method of the class is dispatchTask, which calls the addTask method according to the chosen strategy. It should also be noted that the changeStrategy method sets how customers are queued.

**ConcreteStrategyQueue and ConcreteStrategyTime classes**

These classes define the strategy used to send tasks to the Server. The addTask () method calculates the shortest queue time, or the smallest number of clients in a queue. Depending on these values, the server to which the received task will be added as a parameter is chosen. Also, the waitingTime corresponding to the chosen queue is updated in the same method.

**Statistics class**

This class calculates and shows the statistics for average waiting time, peak time (the time when there are the most tasks in servers), average service time.

**SimulationManager class**

It is the most complex class of the application, as it actually performs the simulation. This class defines all the data needed for the simulation, as well as a list of tasks such as CopyOnWriteArrayList.

The sortTasks () method is a method that sorts a list of tasks according to their arrival time.

The generatedRandomTasks () method generates an appropriate number of random tasks that meet user-specified specifications. At the same time, this method deals with sorting the generated tasks according to their arrival time, calling the sortTasks () function.

The run () method specifies the behavior of the thread running the simulation. It scrolls through the generated task list, while the currentTime value is less than the simulation time. If the task has the arrivalTime equal to the current time, it is sent to a queue, invoking the dispatchTask method. The client is then removed from the task list because it is no longer waiting. All this method deals with the transmission of the simulation data both to the .txt file and to the graphical interface, by calling the static display method from SimulationFrameController.

**QueueManagementSystemController class**

This class makes the connection between the window in which the simulation data is entered and the actual implementation. Labels, textFields, buttons, etc. in View are defined here. The onStartClick () method starts the thread for simulation. It describes what will happen after you press the Simulation button, handles the InvalidInputException exception, and, if the data you enter is correct, opens a new window in which the simulation results will be displayed.

**SimulationFrameController** class

This class controls the Simulation Frame window. It contains the display method, through which the simulation results will be displayed in TextArea in real time.

**Graphic interface**

It was made using JavaFX. Buttons, textFields and labels have been declared in the two FXML tabs called queue.fxml and simulationFrame.fxml, respectively, which have the role of Views for our application. The simulation results are displayed in the Simulation Frame window using the Platform.runLater () method which calls the display method in the SimulationFrameController.

1. **Results and testing**

To test the functionality of the application, we introduced the following tests below, running them with both the ShortestQueue strategy and the ShortestTime strategy. The results of each simulation were displayed, both in the Simulation Frame window and in an automatically generated file, the name of which also contains the type of strategy for which the simulation was performed.

Comparing the obtained results, we notice that by using the Shortest Time strategy we obtained a shorter average waiting time, than by using the Shortest Queue technique, on the same set of input data. However, we can conclude that the values ​​for averageWaitingTime are similar in both cases, if the input entered by the user in the graphical interface is the same.

At the same time, it is observed that pickTime, but also the average serving time, respectively waiting in line, differs from one simulation to another. This is justified both by the fact that the list of customers is randomly generated at each simulation, and by the fact that the simulation data we performed on the test differ.

1. **Conclusions**

The application develops a queue management system, which assigns each customer to a queue so that the waiting time is minimized as much as possible. Depending on the data entered by the user, the evolution of each queue will appear in real time on the screen. In terms of further development possibilities, the graphical interface can be improved so as to simulate in a more realistic way the arrival of customers, adding them to the queues, respectively removing them from the queue, as well as better synchronization of threads. At the same time, another later implementation could be to display more statistics in the GUI, such as the average number of customers waiting in a queue at a given time, or the least crowded time slot. accumulated a lot of knowledge related to object-oriented programming, but especially to the use of Threads and the creation of graphical interfaces. I also learned how to better organize my code, following the Model-View-Controller structure, so that it is as readable as possible.

1. **Bibliography**

* <https://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html>
* <https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm>
* <http://tutorials.jenkov.com/javafx/concurrency.html>
* <https://dzone.com/articles/java-thread-tutorial-creating-threads-and-multithr>
* <https://www.javatpoint.com/how-to-create-a-thread-in-java>
* <https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html>
* <https://www.geeksforgeeks.org/filewriter-class-in-java/>