DOCUMENTATION

ASSIGNMENT *2*

NAME: STOICA IRINA

GRUPA: 30422

**TABLE OF CONTENTS**

1. Assignment Objective……………………………………………………………………………………3
2. Problem analysis, modelling, scenarios, use cases………………………………………………………3
3. Design……………………………………………………………………………………………………4
4. Implementation…………………………………………………………………………………………..6
5. Results……………………………………………………………………………………………………7
6. Conclusions………………………………………………………………………………………………7
7. **Assignment Objective**

The purpose of the assignment is to create a Java Application which simulates and analyses queuing based systems for determining and optimizing the client waiting time. Queues are often used and realised in the real world, where their pur pose is to provide an organized place for the clients to wait before they receive the service they want. The application should simulate a number of clients arriving at a certain time in order to obtain a service. When clients enter the queue, they will wait for the clients who arrived previously to finish their own service time and after a period of time, the time needed for their service, they will leave. The application is also computing the peak hour, average waiting time and average service time for the clients. The peak hour is the moment in the simulation with the most clients getting their service and waiting in the queues, the average service time is the average between the service periods of all clients and the average waiting time is the average time spend by the clients waiting in the queue before they get to start their service time.

In order to achieve our goal, we can define some secondary objectives. We will need to generate random clients that will be the subject of our simulation. They will be defined by an ID, the time at which they arrive, and the time needed for the service to be performed. When the clients arrive, they will enter either in the shortest queue available or in the queue that has the shortest total waiting time. The queues will run parallel with each other and will be managed by the simulation manager that also creates the clients. The data regarding the simulation will be entered by the user of the application. The user will enter at the beginning the number of clients, the number of queues, time interval for the simulation, the minimum and the maximum arrival time of the clients and the minimum and maximum service time of allocated to the clients. The queues and the waiting clients will be observed in the user interface in real time.

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

Requirements

1. The simulation should interact with the user and receive the data parameters needed for the simulation to run.
2. The simulation should create a thread for each queue and run all the threads simultaneously in order to mimic the real-life situation.
3. It should provide the statistics (peak hour, average waiting time, average processing time) at the end of the simulation.
4. It should offer real time feedback of the state of the simulation, the clients that are about to arrive, the clients that are in the queue at a given moment in time.
5. It should also record the steps of the simulation and the computed statistics in a file.

Use-cases

There are two use cases of the application and one of them includes another 3. The user can set up the simulation by entering the data in the user interface and choosing the strategy. The user can start the simulation after entering the needed data. This use case includes the actions of generating the clients with the random client generator, creating the queues and sending the generated clients to the queues.

Diagram

Description automatically generated

1. **Design**

The application contains 2 main packages: model and business logic. Besides, there is an App class which loads the fxml file view.fxml which is the user interface. This user interface communicates with the logic part of the application through the Controller class which initiates and calls the methods needed after getting the necessary data from the user interface. The business logic package contains the SimulationManager and Scheduler classes, the Strategy interface which is implemented by the ConcreteStrategyTime and ConcreteStrategyQueue classes and the SelectionPolicy enum. The model package is made from the Server and Task classes.

Interface

The interface Strategy has one method which adds a client to one of the queues. In this way we can create multiple classes which will add the client to the queue depending on different criteria such as length of the queue or the waiting time of the queue.

Enum

The enum SelectionPolicy is used in order to limit the confusion when it comes to the chosen strategy.

Class Design and relationships

The class App launches the application with the user interface. The class Controller is created in order to make the connection between the user interface and the business logic. It initializes a SimulationManager, sends the data entered by the user in the UI which will be checked in the business logic part, displays the error if the data is not valid, and holds the declarations of all the elements of the GUI (buttons, text fields, labels etc.).

Two classes, ConcredeStrategyQueue and ConcreteStrategyTime implement the Strategy interface. These classes are responsible for adding the clients to a queue depending on the criteria chosen, the least time or the shortest queue.

The class SimulationManager is the main class with the most functionality. It implements the interface Runnable, which means that the instance is executed by a thread and that it has to implement the method run (). This class forms an aggregation with the Scheduler class that is responsible for sending the clients to the queues. The simulation manager generates clients with IDs from 0 to the number desired by the user and arrival and processing time within the limits also chosen by the user. The queue is represented by the class Server, which, like the simulation manager, implements the interface Runnable and forms an aggregation with the class Task, which is a class that mainly holds the variables we need for our clients, we do not have a lot of functionality in this class, its purpose is rather to keep the variables organized and to access them easily.

The Runnable interface provides the means for a class to be active while not subclassing it. In our case it is better to implement the Runnable interface rather than extending the Thread class, because we only want to override the run () method, we do not need any of the other methods of the Thread class.

Diagram

Description automatically generated

User interface

In order to create the user interface, I used the JavaFX Scene Builder in order to arrange the elemets on the screen. The Objects are declared in the Controller class, and the arrangement and aspect of them are settled in the View.fxml file. The user interface contains seven text fields and a choice box for the user to enter the data, in the left of the text field is the description of the content that should be entered. If the user attempts to write letters or symbols which do not form a number, the simulation will not start. It will display the message “Invalid input data. Please enter valid input” and wait for the user to enter valid data. When the user enters the desired data, he can click on the “Start Simulation” button and wait for the simulation to finish in order to see the statistics. In the lower part of the screen, the queues and their clients are displayed in real time.

Graphical user interface

Description automatically generated

Data Structures

For the implementation of the queues, we need a data structure that is thread safe and has the queue properties. I chose the ArrayBlockingQueue which implements the BlockingQueue interface. This is a bounded blocking queue backed by an array. The element which has been in the queue for the longest will be the first extracted. It has a fixed capacity that once created will not be changed. Other data structures can be used too if they implement the BlockingQueue interface. I also have to use an AtomicInteger and AtomicBoolean, which provide operations on underlying int and boolean value that can be read and written atomically, and also contain advanced atomic operations. They are used in multithreaded context in order to perform atomic operations without the synchronized key. For storing the random generated clients, I use the Collections class which implements synchronization wrappers which create synchronized views of collections such as synchronizedList. The good thing is that the synchronized collections are thread safe which is why I use them, but the fact that the user has to manually synchronize the collection when iterating over it is inconvenient.

1. **Implementation**

The class ConcreteStrategyQueue implements the Strategy interface, to be more specific, the method addTask (List<Server> servers, Task t). We go through all the servers, and we count the number of tasks they have in order to find the one with the least number of tasks at the current moment and add the new task to it. We do similarly with the ConcreteStrategyTime.

The Controller class contains the declarations of the labels and text boxes of the user interface as well as the check box. Besides, there is the method that initializes the simulation when pressing the start button. The controller class interacts with the view.fxml file

The simulation manager has in its constructor the controller as parameter, in order to be able to access all the information it needs (number of clients, number of servers, minimum arrival time, maximum arrival time, minimum processing period, maximum processing period, time limit). This information needs to be checked. The numbers entered need to be positive and the minimal values need to always be smaller than the maximal values. In addition, the maximal arrival time always be smaller than the time of the whole simulation. We also create an instance of a scheduler in the simulation manager and pass the number of servers, the simulation time, and the minimal arrival time, so that the scheduler can guide the clients to the queues according to the specifications. In addition, we can already compute the average processing time of the tasks. A first important method of the simulation manager is the random client generator called generateNRandomTasks (). We generate the desired number of tasks and after that we sort them ascendingly depending on their arrival time. For sorting the list, we define it as a collection and use the arrival time as a comparator in the method sort ().

The in the run () method we initialise the current time of the simulation to 0 and increment it after waiting one second until the current time is equal with the duration of the simulation. At each step in time, we check to see if there are any generated tasks that have their arrival time equal to the current time. If so, the scheduler takes the task from the waiting list and sends it to the queue with the shortest queue/with the shortest waiting time. We also log all the information at a given time in the log file and check the number of tasks present in the queues to see if it is a peak hour. After this we increment the current time and wait for one second. The log (int currentTime) function opens and writes the current waiting list of clients, the queues, and the clients in the queues in the file log.txt. At the end of the simulation, we also log the obtained statistics in the file with the logStatistics() function.

I also created a runtime exception for when the user enters invalid time intervals or negative numbers and a function to check for this situation.

The Scheduler class contains a list of the servers and the strategy as criteria for distributing the tasks in the queues. The scheduler also has a constructor which receives the number of servers, the duration of the simulation and the current time. The function dispatchTask (Task t) is used to add a task depending on the strategy chosen in the scheduler. A function updateNrOfClients(int currentTime) will decrement at each moment in time in order to have the remaining time of the task in the queue and the function updateWaitingTime(float waitingTime) will keep track of the total average waiting time.

The Server class implements the Runnable interface and has a BlockingQueue. The constructor has as parameters the duration of the simulation and the current time. The method addTasks (Task newTask) adds a task to the queue. In the run () method, the thread initially sleeps for one second, and after that enters a loop in which it is checked if the queue is empty. If it is, the current time is incremented, and the thread sleeps for one second. If there is at least one element in the queue, it is taken out and the thread sleeps the 1 second for each second of processing time of the task that was previously taken out. The function getTasks () returns an ArrayList of the tasks in available in the BlockingQueue at the given moment. If there are no tasks at the moment, the function returns an empty list. If there are elements in the queue, they will be moved to the ArrayList and then copied back to the blocking queue.

1. **Results**

For the testing part of the assignment, I ran the application with the parameters given in the requirements, logged the results in files and saved them. The test results can be found in the folder of the project.

1. **Conclusions**

Unfortunately, I encountered some difficulties regarding the real time display of the contents of the queues in the user interface and I only displayed the parameters of the simulation that the user has to enter. There can definitely be improvement in this area. This assignment gave me the opportunity to work with threads, which I have not had much experience with until now, and I think it was some good practice, even though I am aware that the subject is extensive and that programs with threads can easily become very complex and difficult, I think I am a little more confident in working with threads.