**DOCUMENTATION**

**ASSIGNMENT #2**

STUDENT NAME: Bărăian Adrian-Călin

GROUP: 30422

**CONTENTS**

[1. Assignment Objective 3](#_Toc128043139)

[2. Problem Analysis, Modeling, Scenarios, Use Cases 4](#_Toc128043140)

[3. Design 5](#_Toc128043141)

[4. Implementation 11](#_Toc128043142)

[5. Results 18](#_Toc128043143)

[6. Conclusions 18](#_Toc128043144)

[7. Bibliography 19](#_Toc128043145)

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 a 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.

1. **Problem Analysis, Modeling, Scenarios, Use Cases**

**Use Cases**

A diagram of a system

Description automatically generated

Figure 1. Use Case Diagram

1. **Use Case**: Setup Simulation

**Primary Actor**: User

**Main Success Scenario**:

1. The user inserts the values for the: number of clients, number of queues, simulation interval, minimum and maximum arrival time, and minimum and maximum service time
2. The user clicks on the validate input data button
3. The application validates the data and displays a message informing the user to start the simulation

**Alternative Sequence**: Invalid values for the setup parameters

* The user inserts invalid values for the application’s setup parameters
* The application displays an error message and requests the user to insert valid values
* The scenario returns to step 1

1. **Use Case**: Start Simulation

**Primary Actor**: User

**Main Success Scenario**:

1. The user has correctly introduced the simulation data.
2. The user clicks on the execute simulation button
3. The simulation starts
4. The user is able to see the simulation steps being displayed on the screen

**Alternative Sequence**:

* The user inserts the data but does not press the execute button
* The scenarios stops at step 1

**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 compute the average waiting time, average service time and peak hour of the simulation

**Non-Functional Requirements**

* Intuitiveness:
  + The GUI should be intuitive and easy to use by the user;
* Stability:
  + The simulation application should be stable and reliable;
* Input validation:
  + The simulation application should validate user inputs to prevent erroneous computations.

1. **Design**

**Level 1**: Overall System Design

A diagram of a software system

Description automatically generated with medium confidenceFigure 2. Conceptual Architecture of the Application

**Level 2**: Division into sub-systems/packages

A screenshot of a computer

Description automatically generated

Figure 3.Classes of GUI Package

**Level 3**: Division into classes

A screenshot of a computer

Description automatically generated

Figure 4. Classes Of Model Package and Utils Package

A screenshot of a computer

Description automatically generated

Figure 5. Classes Of BusinessLogic Package

A diagram of a software system

Description automatically generatedFigure 6. Class Diagram of the Application

**Data Structures and Types used**:

* **CopyOnWriteArrayList**: Enhanced version of ArrayList in which all modifications(add, set, remove etc.) are implemented by making a fresh copy. It is a data structure created to be used in a concurrent environment
* **AtomicInteger**: Class that can be used in simple applications like concurrent counting and building simple readable code without the complexity of using a lock. Can sometimes be more efficient than a regular integer with a lock as protection.
* **BlockingQueue**: interface that supports flow control (in addition to queue) by introducing blocking if either BlockingQueue is full or empty. A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more elements or clearing the queue completely. Similarly, it blocks a thread trying to delete from an empty queue until some other threads insert an item.

1. **Implementation**

* Task Class:

Fields:

* - ID: int (each task / client has a unique id);
* - arrivalTime: int (the time a client arrives at the store).
* - serviceTime: int (the time needed for the task / client’s job to be finished)

Public Methods:

* Getters for the ID, arrivalTime and serviceTime fields (used for obtaining the data stored in the private fields ID, arrivalTime and serviceTime);
* decrementServiceTime(Server): Decrements the serviceTime of the current task to be processed in the queue, and also updates the waiting period of the whole queue

public void decrementServiceTime(Server server) {  
 this.serviceTime--;  
 server.setWaitingPeriod(new AtomicInteger(server.getWaitingPeriod().decrementAndGet()));  
}

* Server Class:

Fields:

* - tasks: BlockingQueue<Task> tasks: Private field used to implement the queue based system, with multiple task type objects;
* - waitingPeriod: AtomicInteger (field used to keep track of the waiting period needed for the last person in line until service, meaning until the last person reaches first in line)

Public Methods:

* + addTask(Task): void (Method for adding a Task to the tasks field of the Server class)
* Getters for the tasks and waitingPeriod fields (used to get the reference to the private tasks BlockingQueue and the data from the private waitingPeriod field)
* Setter for waitingPeriod (used to set data to the private field waitingPeriod)
* Overriding of the run method from the Runnable interface for Thread implementation of the Server (Queue). The first task in queue is being processed, the thread sleeps for the serviceTime of the task seconds, then the task is removed from the queue.

public void run() {  
 while(true) {  
 if(!tasks.isEmpty()) {  
 Task currentTask = tasks.peek();  
 try {  
 Thread.sleep(currentTask.getServiceTime() \* 1000L);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 this.waitingPeriod.addAndGet(-currentTask.getServiceTime());  
 tasks.remove();  
 }  
 }  
}

* Scheduler Class:

Fields:

* - servers: CopyOnWriteArrayList<Server> (list of queues / servers of the application)
* - maxNoServers: int (the maximum number of servers)
* - maxTasksPerServer: int (the maximum number of tasks per server)
* -strategy: Strategy (the strategy used by the current program simulation, either Time Strategy or the Shortest Queue Strategy)

Public methods:

* ChangeStrategy(SelectionPolicy): void (Method used to change the strategy of the application based on the input provided by the user)

public void changeStrategy(SelectionPolicy selectionPolicy) {  
 if(selectionPolicy == SelectionPolicy.SHORTEST\_QUEUE) {  
 this.strategy = new ShortestQueueStrategy();  
 }  
 if(selectionPolicy == SelectionPolicy.SHORTEST\_TIME) {  
 this.strategy = new TimeStrategy();  
 }  
}

* dispatchTask(Task task): void (Method that calls the method addTask of the strategy of the scheduler).

public void dispatchTask(Task task) {  
 strategy.addTask(this.servers, task);  
}

* Getter method for the servers private field.
* SimulationManager Class:

Fields:

* - timeLimit: int
* - maxProcessingTime: int
* - minProcessingTime: int
* - maxArrivalTime: int
* - minArrivalTime: int
* - numberOfClients: int
* - numberOfServers: int
* - selectionPolicy: SelectionPolicy
* - scheduler: Scheduler
* - view: View (Field used to manage the view of the simulation so that to access the fields in which the results are logged)
* - tasks: CopyOnWriteArrayList<Task> (Field that holds the randomly generated tasks / clients)
* + peakHourNrClients: int (Field used to keep track of the clients to help determine the peak hour of the simulation)
* - static averageWaitingTime: int (Field used to determine the average waiting time for a client to reach the first in line postion)
* - static averageServiceTime: int (Field used to determine the average service time of all the clients generated in the current simulation)
* - static peakHour: int (Field used to determine the peak hour of the simulation, meaning the hour in which there are the most clients in all queues)

Public methods:

* generateNRandomTasks(): void (Method used to generate numberOfClients different tasks, each with unique id. The processing time and the arrival time are computed as random numbers between the minProcessingTime/maxProcessingTime and minArrivalTime/maxArrivalTime)

public void generateNRandomTasks() {  
 int currentID = 1;  
 for(int i = 1; i <= numberOfClients; i++) {  
 int processingTime = (int)(Math.random() \* (maxProcessingTime - minProcessingTime) + minProcessingTime);  
 int arrivalTime = (int)(Math.random() \* (maxArrivalTime - minArrivalTime) + minArrivalTime);  
 Task task = new Task(currentID++, arrivalTime, processingTime);  
 this.tasks.add(task);  
 }  
  
 this.tasks.sort(Comparator.comparingInt(Task::getArrivalTime));  
  
 }  
}

* Overriding of the run method from the Runnable interface. This method manages the main simulation thread. At each step (hour) add a task / client form the task / client list to a queue (server) based on the strategy selected by the user, decrement the service time of the first in line client, and the waiting period of the server the client belongs to. Also in this method, we create the output strings that are logged in the log.txt file and also displayed in the view through the updateView method and the peakHour of the simulation.
* Getters and setters for the private fields averageWaitingTime, averageServiceTime, getter for scheduler and selectionPolicy fields, and setters for selectionPolicy and view fields.
* SelectionPolicy Enumeration:

public enum SelectionPolicy {  
 SHORTEST\_QUEUE, SHORTEST\_TIME  
}

* Strategy Interface:
* void addTask(CopyOnWriteArrayList<Server> servers, Task task);

Method that will be overridden by the ShortestQueueStrategy Class and TimeStrategy Class

* TimeStrategy Class:

No Fields.

Public methods:

* @Override  
  public void addTask(CopyOnWriteArrayList<Server> servers, Task task) {  
   if(servers.isEmpty()) {  
   return;  
   }  
    
   Server minServiceTimeServer = null;  
    
   int minServiceTime = 0x7FFFFFFF;  
    
   for(Server server : servers) {  
   if(server.getTasks().isEmpty()) {  
   minServiceTimeServer = server;  
   break;  
   }  
   if(server.getWaitingPeriod().get() < minServiceTime) {  
   minServiceTime = server.getWaitingPeriod().get();  
   minServiceTimeServer = server;  
   }  
   }  
    
   if(minServiceTimeServer != null) {  
   minServiceTimeServer.addTask(task);  
   }  
    
  }

Method used to add a task / client to a queue based on the least waiting time needed for it to reach the first in line

* ShortestQueueStrategy Class:

No Fields.

Public Methods:

* @Override  
  public void addTask(CopyOnWriteArrayList<Server> servers, Task task) {  
   if(servers.isEmpty()) {  
   return;  
   }  
     
   Server currentShortestServer = servers.getFirst();  
    
   for(Server server : servers) {  
   if(server.getTasks().size() < currentShortestServer.getTasks().size()) {  
   currentShortestServer = server;  
   }  
   }  
   if (currentShortestServer != null) {  
   currentShortestServer.addTask(task);  
   }  
  }

Method used to add a task / client to the shortest queue in terms of count of tasks / clients.

* Utils Classes:

Utility classes that contain static methods which work with the main data types of the application but are not directly related to the functional part of the application.

InputValidator Class:

* + isNumeric(String str): boolean (Method used for checking if a given string is only formed of digits, meaning is a number):

public static boolean isNumeric(String str) {  
 if(str == null) {  
 return false;  
 }  
 try {  
 Integer d = Integer.parseInt(str);  
 } catch(NumberFormatException nfe) {  
 return false;  
 }  
  
 return true;  
}

* + isValidInput(int, int, int, int, int, int, int): boolean (Method used to check if the input provided in the GUI is valid, meaning minServiceTime < maxServiceTime, all the values provided are positive integers etc)

public static boolean isValidInput(int minServiceTime, int maxServiceTime, int numberOfClients, int numberOfQueues, int simulationTime, int minArrivalTime, int maxArrivalTime) {  
 if(minServiceTime < 0 || maxServiceTime < 0 || numberOfClients < 0 || numberOfQueues < 0 || simulationTime < 0 || minArrivalTime < 0 || maxArrivalTime < 0) {  
 return false;  
 }  
 if(maxServiceTime < minServiceTime || maxArrivalTime < minArrivalTime) {  
 return false;  
 }  
  
 return simulationTime >= maxArrivalTime;  
}

* GUI related classes:
* View Class:

The class that implements the visual part of the application, the way the content is displayed to the user. The entire application content is placed in the conentPane which is actually a grid of 1 row by 2 columns. Each column is in turn a panel made up of different components to achieve the final design. On the right column we have the logging area, made up of 2 text areas, one which displays the queue content and the other which displays the remaining clients / tasks. In the left column, we have the input part of the user interface, where the user can insert data and start the simulation.

Fields:

* All the fields representing JSwing components (e.g. labels, textfields, buttons, panels);
* The controller field for which we set as view the current view
* Controller controller = new Controller(this);

Methods:

* + prepareGUI(): void (Method that “prepares” the base of the window, setting the dimensions, the exit\_on\_close operation and calls the methods that prepare the actual content of the GUI)
* + prepareInputPanel(): void (Method that “prepares” the left part of the window, where the input happens <<the input of the data, the choice of strategy, the press of execute button>>)
* + prepareLogPanel(): void (Method that “prepares” the right part of the window, where the output happens <<the queue log area, the remaining tasks / clients area)
* + updateView(String loggerQueues, String loggerWaiting): void (Method used to update the text in the text areas where the simulation is displayed)
* + createResultsFrame(float, float, int): void (Method used to create a new window at the end of the simulation where the average waiting time, average service time and the peak hour are displayed).
* Getters and setters for the input and label objects of the view.

If the wrong input is provided, the user gets a message specifying that there is an error witht the input provided.



* Controller Class:

Class which implements all the actions triggered by the interaction of the user with the GUI. In this case, the application has a button, which when pressed starts the simulation using the strategy provided in the combo box.

Fields:

* - view: View (private field of type View representing the view being “controlled” by the controller)

Methods:

* + actionPerformed(actionEvent e): void (When the execute button is pressed, first the input is validated. If the input provided by the user has the right format, then the simulation of the application starts, if not, then an error message is displayed and the user must insert new data in the input fields.)

A screenshot of a computer

Description automatically generated

1. **Results**

A math equations on a white background

Description automatically generated

The tests provided in the laboratory assignment have been simulated and the results have been logged inside 3 .txt file provided in the project files, each with an appropriate name.

1. **Conclusions**

The assignment presents a real-life problem. Using threads we can simulate how the queues at a marketplace work, assigning a thread to each queue. Two strategies have been studied, one which does not consider the time needed to service a task but only the number of tasks / clients in a queue. This is called the shortest time strategy. The second strategy implemented is the time strategy where clients are assigned considering the minimum waiting time of each queue. The time strategy is the more efficient one out of the two, resulting in a smaller average waiting time value as seen in the provided tests.

The assignment has helped me understand how threads work and solve problems related to real-life situations.

1. **Bibliography**
2. [Fundamental Programming Techniques (dsrl.eu)](https://dsrl.eu/courses/pt/) – Laboratory Assignment Guide + Lectures