Documentation

Assignment 2. Queues Management Application Using Threads and Synchronization Mechanisms

Student name: Potra Paul-Antonio

Group: 30421

# CONTENTS

[1. Assignment Objective 3](#_Toc132929232)

[2. Problem Analysis, Modeling, Scenarios, Use Cases 3](#_Toc132929233)

[3. Design 4](#_Toc132929234)

[4. Implementation 7](#_Toc132929235)

[*Task* record 7](#_Toc132929236)

[*Constants* class 8](#_Toc132929237)

[*Server* class 8](#_Toc132929238)

[*StartMenu class* 9](#_Toc132929239)

[*Simulation* class 10](#_Toc132929240)

[*Strategy* interface 10](#_Toc132929241)

[*SelectionPolicy* enum 11](#_Toc132929242)

[*Scheduler* class 11](#_Toc132929243)

[*SimulationManager* class 12](#_Toc132929244)

[5. Results 13](#_Toc132929245)

[6. Conclusions 13](#_Toc132929246)

[7. Bibliography 14](#_Toc132929247)

# Assignment Objective

The purpose of this assignment is to implement a Java GUI application representing a queue management system, which will distribute a set of clients to a set of queues to minimize the waiting time.

* Assign clients to queues based on different strategies (smallest waiting time, shortest queue), and display the real time evolution of the queues.
* Provide the user with a simple to use and intuitive GUI, where he can insert all the necessary data needed to perform the simulation.
* Create a log of events as a .txt file where the user will be able to see evolution of the queues in more detail.
* Perform an analysis on the data set to compute the most relevant details about the simulation (peak hour, average waiting time, average service time)

# Problem Analysis, Modeling, Scenarios, Use Cases

The first step is to break down the project, analyze the requirements and figure out the design procedure.

It is a simulator, so it must be able to take user input. The required data is the number of clients and queues, the simulation time, the minimum and maximum arrival and service time. Based on those inputs, the application should open another GUI where the user can see the real-time evolution of the queue (which clients enter which queues, the time they enter and the time they exit). So, the application requires 2 GUIs, one for the user input and one for the actual simulation. The one for the simulation should update itself over time.

The application should also compute details about the simulation: average waiting time, average service time, peak hour for the simulation interval.

Finally, after the simulation, a log of events should be generated (as a .txt file), where the user will see more details about the simulation.

Below is a use case diagram highlighting the above.

Diagram

Description automatically generated

# Design

The application follows a Model-View-Controller design architecture, and it contains three packages: *model*, *GUI*, *businessLogic*.

The *model* package contains 3 classes, *Server* and *Constants*, and a record, *Task*.

A *Task* represents a client, and I used a record to automatically have all instance variables private, automatic constructor and getters.

*Constants* is a class which contains one constant, the TIME\_UNIT with respect to which the simulation is performed. A TIME\_UNIT of 1000 is the equivalent of a second in the application runtime.

*Server* models the queue, a thread where clients come and go at specific times.

The *view* package contains the *StartMenu* and *Simulation* classes, and it presents the GUIs with which the user interacts. The *Simulation* GUI appears only after the user entered the required data in the *StartMenu*, and that data was checked for validity.

The *businessLogic* package contains an interface (*Strategy*), 2 implementations of that interface (*ShortestTimeStrategy* and *ShortestQueueStrategy*), and enumeration (*SelectionPolicy*), and 2 more classes (*Scheduler* and *SimulationManager*).

The interface and its implementations were used so the application can change the policy with respect to which the clients are assigned to the queues. The *ShortestQueueStrategy* assigns the client to the queue with the smallest length, while the *ShortestTimeStrategy* assign the client to the queue with the smallest waiting time, being the optimal strategy. The *SelectionPolicy* is an enumeration used to represent those cases and switch between them.

*Scheduler* creates the threads (servers, queues), switches the *SelectionPolicy* and assigns the *Strategy*.

SimulationManager is the class that manages the entire simulation: it checks the user input, it generates random clients, it decides when the clients should be assigned to the queues (which queue depends on the selection policy), it updates the simulation GUI and the log of events.

This can be more clearly seen in the UML package diagram below.

Graphical user interface

Description automatically generated with medium confidence

The UML class diagram can also be seen below, but first, a few mentions.

* A + sign marks the fact that the method / variable is public.
* A – sign marks the fact that the method / variable is private.
* A method which is underlined is a static method.
* This notation, method(a : b) : c, means that the method takes a parameter named a, of type b, and returns something of type c.

I will leave out the instance variables and the methods of the *StartMenu* and *Simulation* classes for now.

Timeline

Description automatically generated

Now, for the trickiest part from the project implementation: thread synchronization. We have two classes which implement the Runnable interface, *Server* and *SimulationManager*.

The first and simplest synchronization technique we can apply is using the *synchronized* keyword for the critical region of the threads, those being the *run*() methods. This will provide atomicity through the *run()* method, which will help for the second step, using *AtomicInteger*.

Text

Description automatically generated

We declare the *waitingPeriod* for the Server as an *AtomicInteger*. With this, we make sure that only one operation at a time can be performed on the *waitingPeriod*. Since both the *Server* and *SimulationManager* read and/or write the *waitingPeriod* instance variable, making it atomic is key for synchronizing the threads and making sure the tasks get added to the right server. To keep the operation atomic, we only used *getAndAdd()*, *addAndGet()*, *getAndIncrement()*.



Finally, we use a *BlockingQueue* to represent the tasks from the servers. This ensure thread safety, being designed for concurrent access from multiple threads (which, again, is needed, since the server removes tasks from it and the manager decides when to add new tasks). The *BlockingQueue* provides blocking put and take methods, meaning it supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element. For this project’s purposes, we assume the maximum capacity in the queue is the total number of clients. So, when we only have one queue, all the clients can be placed to that queue as soon as they arrive.



# Implementation

# *Task* record

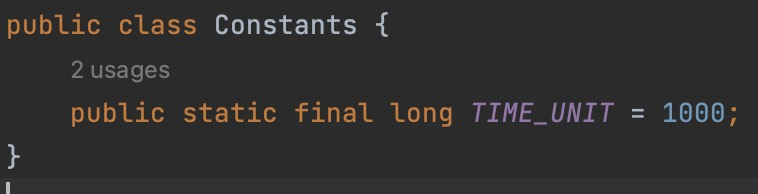
* A record containing an id, and arrival time and a service time.
* An additional toString method to print the task as a string.

Chart, text

Description automatically generated

# *Constants* class

* By modifying the TIME\_UNIT, the application will run faster or slower, because all the time units in the code are computed based on this constant.



# 

# *Server* class

* A *Server* models a queue, so it contains a list of clients (Tasks).
* Implements *Runnable*.
* It can report its size, and it has a maximum capacity, which means it also must determine whether it is full or not.

Graphical user interface, text, application

Description automatically generated

* Tasks can be added to the queue, and the queue can close itself.

Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generated

*run()* method:

* @Override from Runnable.
* The server only runs while it is open.
* It continuously takes the first client in the queue, waits for a period equal to its service time and removes it from the queue.

# *StartMenu class*

* The user enters the number of clients and queues, the simulation time, the minimum and maximum arrival time, and the minimum and maximum service time.
* After pressing the “Simulate” button, a new GUI will appear where the user can see the real-time evolution of the simulation.

Graphical user interface

Description automatically generated

# *Simulation* class

Graphical user interface, text, application, email

Description automatically generated

# *Strategy* interface

* Contains only one method which is implemented by each individual stragety.

Text

Description automatically generated with medium confidence

* The ShortestQueueStrategy class will add the task to the shortest queue, i.e., the one with the smallest size.
* The ShortestTimeStrategy class will add the task to the queue with the smallest waiting period, which is the fastest queue.

# *SelectionPolicy* enum

* Used to select which strategy the simulation should use. It is essential for implementing the strategy design pattern.

Text

Description automatically generated with medium confidence

# *Scheduler* class

* Creates, initializes and starts all the queues (has a list of servers).
* Contains a strategy that is changed based on a selection policy.

Text

Description automatically generated

* Calls the add task method on the strategy object.

Graphical user interface, text

Description automatically generated with medium confidence

# *SimulationManager* class

* On initialization, it waits for the user input, after which it starts the scheduler (here the queue threads are created

Text

Description automatically generated

* It generates a list of random tasks, with a sequential id (from 1 to the number of clients), a random arrival time within an interval given by the user, and a random service time, again within a user given interval.
* It generates the execution log, by creating a file in the format log-noClients-noQueues-simulationTime-minArrivalTime-maxArrivalTime-minServiceTime-maxServiceTime.txt. This way, every time the user runs the application, it will have a new log of events. In case the user runs the same simulation as for a previous case, the existing log will be overwritten.
* It updates the execution log from the simulation (by appending text to the text area from the GUI) and from the .txt file (still by appending, but with more information and details).
* It checks the user input to make sure only integers were inserted.

Text

Description automatically generated

*run()* method:

* @Override from *Runnable*.
* Iterates through the randomly generated tasks (a sorted list).
* Picks the first task from the list, adds it to a queue if the arrival time is greater or equal to the current time, removes it from the list of generated tasks.
* The current time is incremented by one at every step. The manager stops the simulation when the current time reaches the time limit (simulation time).
* At the end of the simulation, it closes all the queues.

# Results

Using the application itself, multiple tests were performed. The result of each test is stored in a separate log of events (.txt file) with a unique name. Every time a new simulation is started, a new log of events will be created, or, if the simulation was already performed before (that is, the user inputs are the same), the log of events for that scenario will be overwritten.

# Conclusions

This project is a very good and simple introduction to Programming Techniques with Threads, having the added benefit of designing a GUI. It allowed me to put into practice some design patterns and methods for thread synchronization, while learning some more advanced data structures, such as the BlockingQueue and the AtomicInteger.

On top of that, the program simulates a real-life scenario, so it can be utilized in a lot of scenarios.

# Bibliography

1. Thread Synchronization - <https://www.javatpoint.com/synchronization-in-java>

2. Volatile Variables - <https://www.javatpoint.com/volatile-keyword-in-java>

3. BlockingQueue - <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html>

4. AtomicInteger - <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html>

5. Java Records - <https://www.baeldung.com/java-record-keyword>

6. Java Swing GUI Design - <https://docs.oracle.com/javase/tutorial/uiswing/index.html>