

**PROGRAMMING TECHNIQUES**

Homework 3

**Student:Nicolescu Alexandru**

**Group:** 30422

**Table of Contents**

**1. Objective 3**

**2. Problem analysis, modeling, scenarios, use cases 3**

**2.1. Problem Analysis 3**

**2.2. Modeling 4**

**2.3. Scenario and use cases 4**

**3. Design 5**

**3.1. UML Diagram 5**

**3.2. Data Structures 6**

**3.3. Class Projection 6**

**3.4. User Interface 8**

**4. Implementation and use of cases 9**

**5. Results 10**

**6. Conclusion and possible updates 10**

**7. References 10**

**Documentation Queuing Based Systems**

1. Objective

This homework’s aim is to use elements of object-oriented programming to implement a simulation application which analyzes queuing based systems for determining and minimizing clients’ waiting time.

1. Problem analysis, scenarios, use cases

Queues are commonly seen both in real world and in the models. 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 its "clients" are waiting in queues. 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 supplier. When a new server is added the waiting clients will be evenly distributed to all current available queues.

The application should simulate a series of clients arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the clients spend waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time depend on the individual clients – when they show up and how much service they need. The finish time depends on the number of queues, the number of other clients in the queue and their service needs.

2.1 Problem analysis

If we take a closer look to the problem of order management we shall find that it is a more complex problem that it may seem at first glance.

First of all, we need to find a way to make all the queues run in the same time, so that it will simulate a genuine real life situation.

Another problem concerning the queue system would be the way the user gives us the data and the way we tell him the result of the processing. After doing some research on the internet and looking at online applications that provide these services that cover the order processing problem I have reached the conclusion that the best way in terms of displaying data which comes from tables is using naturally a matrix of JButtons which is put in the Graphic User Interface.

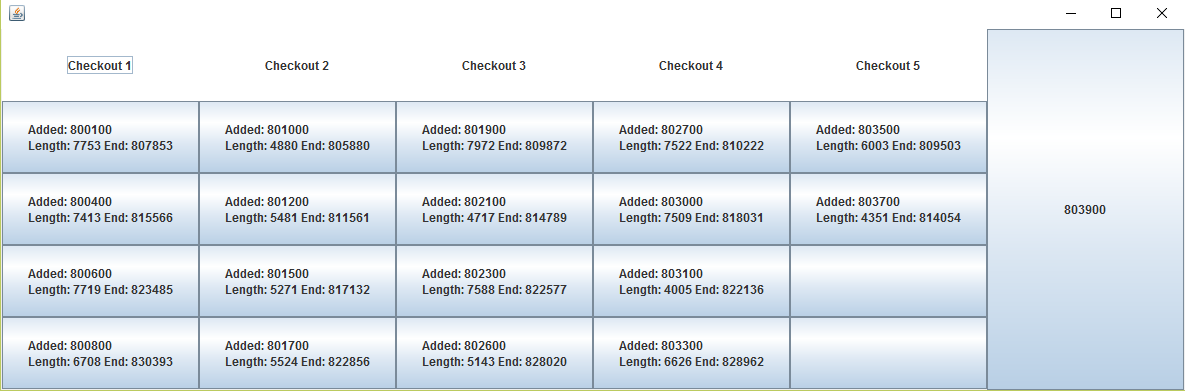
Further on we shall analyze all the aspects which needed to be managed in order for the right functioning of this project.

* 1. Modeling

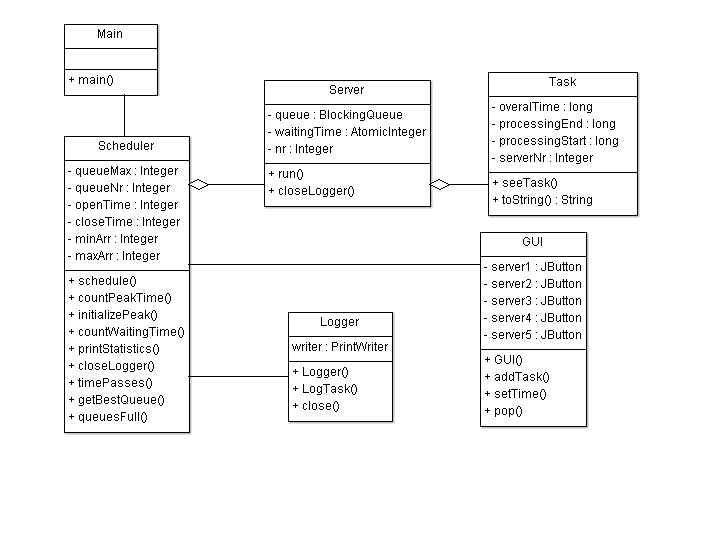
The implementation of this project is the same one as the one presented in the section 2.1. There is only one package which is used for storing all the classes. There are Task (which stores the data of every Task), Server, Scheduler, Generator, Main, Logger and GUI. We shall analyze the role of each of these classes further on.

2.3 Scenarios and use cases

The scenarios were already mentioned, but I will present the details here. Firstly I thought about how the tasks will be managed by the application and my idea was: Tasks stored in the Server, Servers stored in the Scheduler, a Generator in order to create Tasks, a Logger to store the execution timeline and a GUI to display the running of the application. I followed this plan and I did not have any unexpected surprise when it comes to the implementation and the structure of the program.

The use cases are strictly dependent on the user, and finally I order to make the application as user friendly as possible I decided to implement the following user interface:

From now on I will present the functionality of the application using the “lifetime” of a Task. A Task is created in the Scheduler class using the Generator class. The Scheduler is finding the best Server to add the Task, it opens servers if there is a need for a new one and the number of servers does not exceed the limit read from the input file. Further on, a Task is given its properties which are: Start time, End time, Processing Time. When the Task is in the Server BlockQueue it is already displayed on the user interface. It is popped from the Queue from the user interface when the Task ends (its processing time ends after he reaches the top of the Queue). This process is done between and interval of time in which the Queues are open. After the end of the OPEN status of the Queues the application creates the Logger file in which you can see all the Tasks that have been run and it displays additional information about how the processing process evolved.

1. Design

3.1 UML Diagram

The UML diagram is a class diagram in which we can find the relationship between classes and also the elements that the specified class contains.

One could observe that for joining the classes I used several types of relationships. Between the class GUI and the Scheduler class I used association relationship and between Scheduler and Servers I used aggregation. For Server and Tasks I also used aggregation, because it’s the same type of relationship.

3.2 Data structures

In this application I have used various data types including JButtons, JPanels, JFrame for the GUI package, Server, Task, for transferring data between classes and the classic int, String, Boolean for regular operations.

3.3 Class projection

Class projection refers mainly to how the model was thought, how the problem was divided in sub-problems, each sub-problem representing more or less the introduction of a new class. First I will start by mentioning exactly how my problem was divided into classes. I begin by creating the Main class which starts the execution of the application. Then, as you can see from the UML diagram, I called the Scheduler class which starts the actual execution by a repetitive loop while(true). Further on, everything is done mostly by Threads, being the most important feature of the application.

1. *assig3* package

This package contains all the classes which make this application run as it does. We shall start explaining the code from the GUI class.

**this**.setLayout(**new** GridLayout(1, 6));

**this**.setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);

JPanel panels[] = **new** JPanel[5];

**for** (**int** i = 0; i < 5; i++) {

server1[i] = **new** JButton();

server2[i] = **new** JButton();

server3[i] = **new** JButton();

server4[i] = **new** JButton();

server5[i] = **new** JButton();

panels[i] = **new** JPanel();

panels[i].setLayout(**new** GridLayout(5, 1));

add(panels[i]); **this**.setVisible(**true**);. . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . .. . . .. . . . . . . . . .

This is the main code of the GUI class which describes perfectly how everything is done. There are 5 arrays of JButtons used to create the main Frame, one for each Panel which is added to the window.

The Scheduler class is the most important of the application, as it contains most of the logic on which the program is based. We shall analyze some of it’s essential code.

**public** **void** schedule() {

timePasses();

initializePeak();

**while** (currentTime.get() < closeTime) {

**while** (!queuesFull()) {

**if** (needNewServer() && servers.size() < queueNr) {

servers.add(**new** Server(servers.size()));

Thread thr = **new** Thread(servers.get(servers.size() - 1));

thr.start();

}

Random rand = **new** Random();

**int** randomNum = rand.nextInt((maxArr - minArr) + 1) + minArr;

**try** {

Thread.*sleep*(randomNum);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**if** (!queuesFull()) {

Task t1 = generator.generate();

t1.setProcessingStart(currentTime.get());

**int** bestQueue = getBestQueue();

t1.setProcessingEnd(

currentTime.get() + t1.getOveralTime() + servers.get(bestQueue).getWaitingTime().get());

t1.setServerNr(bestQueue);

servers.get(bestQueue).getQueue().add(t1);

servers.get(bestQueue).getWaitingTime().addAndGet((**int**) t1.getOveralTime());

countWaitingTime(bestQueue, servers.get(bestQueue).getWaitingTime().get());

countPeakTime(bestQueue,servers.get(bestQueue).getWaitingTime().get());

graphic.addTask(t1);

}

}

}

}

This is the main Thread which keeps on running until the closing time is equal with the current time. I have used a special method timePasses() in order to simulate as good as possible the passing of the time.

The Task class only stored the data of each Task and therefore there is no important code in it only setters and getters.

The Generator class is used to generate random overalTimes for each new Task. We shall analyze its code:

**public** Task generate() {

Task t = **new** Task();

Random r = **new** Random();

**long** randomValue = ***MIN*** + (**long**) (r.nextDouble() \* (***MAX*** - ***MIN***));

t.setOveralTime(randomValue);

**return** t;

}

The Logger class is used to store in a file each Task that was processed in the execution time.

The Server class is very important as it is the core of the application. Here we have its code:

**while** (**true**) {

**try** {

Task t1 = queue.take();

**long** taskTime = t1.getOveralTime();

t1.setServerNr(nr);

logger.logTask(t1);

Thread.*sleep*(taskTime);

graphic.pop(nr);

waitingTime.set(waitingTime.get() - (**int**) taskTime);

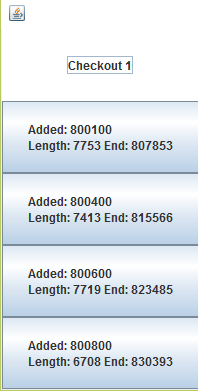
} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

3.4 The interface

 The interface is made out of 5 JPanels which are included in the JFrame in a GridLayout(5,1). We shall describe each JPanel in order to understand how the user interface is created.

This photo represents the first panel which is used in order for the user to see the Threads of Server 1. The user interface is made out of 5 these kind of panels and one JButton which displays the current Time in order for the user to see the perfect running of the application.

3.5 Implementation

In what the implementation is concerned this project was developed in Eclipse and it was only tested in this environment. However the program should maintain its portability. Concerning the code implementation I did not make use of laborious algorithms, but I have rather stayed faithful to the classical algorithms of computing polynomials learned in high school. However I have tried to implement my problem in a way that appears to me as being the most efficient one, this is why I have changed my model at first. Testing implies checking for any errors in the program or limitations of this program. Due to the fact that the program is rather simplistic, they are few errors that might generate this program to work wrong or to stop. These errors are mostly related to the interface or the database connection. I have assumed that the user reads the instructions from the interface and respects them, otherwise if he enters data with invalid format the program will probably generate some bugs and will stop. Hence this part with checking all the possible scenarios will be seen as future development.

1. Results

The application is an user friendly and useful application to analyze queuing based systems for determining and minimizing clients’ waiting time. As the application is developed on a Java platform, it is highly portable and allows it to run on several operating systems (as long as they have the Java SDK installed). The application is straightforward an easy to understand and to use by any user who respects the instructions given in the interface and who has some basic knowledge of database storing, of course. Even though being limited, this application can be considered as being a helpful tool that can be used when dealing with such data storing situations.

1. Conclusions

All in all, the application works perfectly on the required task and it is a user friendly interface which gives the users a simple comprehension of Tasks running and managing.

1. References
2. http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html