**Assignment 2 PT**

**-Andriesa Maria, group 30423-**

**-Queue Simulator-**

1. **Objectives**

Designing and implementation of a queue simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time.

Queues are commonly used to model real world domains. 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 their "clients" are waiting

in queues before they are served. 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 service supplier. When a new server

is added the waiting customers will be evenly distributed to all current available queues.

**2. Analysis, Modelling, Use Cases, Scenarios**

2.1. Analysis

The application provides a real-time evolving simulation of a super market with 3 cashiers and 3 waiting lines. For the purpose of simulating this and in order to be able to see how asynchronous modification of a program can be used when wanting to work in parallel with different execution streams which use the same resources I used multithreading and concurrency structures provided by Java libraries . For each queue I assigned a different thread that would wait for a client to arrive and then processes it by simulating the service provided by a cashier in a certain amount of processing time. By doing this, I ensured the behavior of the real life situation in a market where cashiers serve in parallel multiple clients and where clients can choose which queue to enter to get their products processed as fast as possible.

2.2. Modelling

Internally, the modelling of the simulator is comprised of three main classes of threas , one which generates clients and puts them in the BlockingQueue , one which is the SimulationManager that delegates the clients among queues and the third one is the ClientConsumer class , a thread which waits for clients to be put in the queue and then takes them from the queue and processes them in FIFO order. Also, each customer is characterized by its own id, arrival time and processing time and is all put in a single Customer class.

2.3. Use cases

As per the use case definition, the control lies in the hands of the user once the application starts. The application has been created with the user’s experience in mind and provides a little guide for using every ability of the application correctly. The next diagram shows every function which the user is provided with:

Scenario 1:

a) Identification summary:

Title: The user correctly introduces the input data:

Summary: In this scenario the simulation takes place without errors.

b) Flow of events:

1) The user starts the application

2) The user introduces the chosen data

3) The user presses start

4) The simulation displays real time data

5) The log is generated in the console

Scenario 2:

a) Identification summary:

Title: the user introduces invalid data.

Summary: In this scenario the user introduces data which is not supported

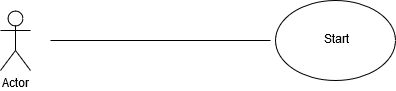
b) Flow of events:

1) The user starts the application

2) The user enters invalid data

3)The user presses the start button

4)An error message is displayed, announcing the user that an invalid data has been introduced



1. **Design:**

The design of the project is based on the model view controller (MVC) design pattern, adapted on the given task. Each class plays a significant role in the design of the project. The SimulationManager, as the name suggest, is the heart of the program.

Also, as it was fitted, a producer-consumer pattern was used. Blocking queues provide blocking put and take methods as well as the timed equivalents offer and poll. If the queue is full, put blocks until space becomes available; if the queue is empty, take blocks until an element is available. Queues can be bounded or unbounded; unbounded queues are never full, so a put on an

unbounded queue never blocks.

Blocking queues support the producer-consumer design pattern. A producer-consumer

design separates the identification of work to be done from the execution of that work by placing work items on a “to do” list for later processing, rather than processing them immediately as they are identified .Here we only initialize the threads, using the ClientProducer thread to fill up the Linked BlockingQueue with a client at a time use the RandGenerator to create the new customer and let the decision of choosing the queue to be put in at the latitude of the ScheduledExecutorService which will manage the consumer threas to take the clients out of the BlockingQueue in a FIFO manner.Each thread in the pool is activated at a fixed interval by the ExecutorService. The logic behind how the customers arrive and are processed is done in the threads. The Client Producer thread takes the randomly generated arrival time from the newly generated customer and decrements it to simulate the time remains until the next customer should come in the queue. Once put in the queue , the consumer threads are free to “consume” the customers by taking them out of the queue and processing them by decrementing the process time which was generated randomly when the client was created. The display on the GUI was a major difficulty due to the synchronization problems that occur when working with concurrency. All the messages about the customer’s state reperesent the data which represents the log messages . A customers’ state informs the consumer thread if the customer has been put into the queue and can be taken out to be processed.

**private** ScheduledThreadPoolExecutor executor = **new** ScheduledThreadPoolExecutor(4);

**private** ScheduledExecutorService ses = Executors.*newScheduledThreadPool*(4);

**private** BlockingQueue<Customer> custQ = **new** LinkedBlockingQueue<Customer>();

ClientProducer producer = **new** ClientProducer(custQ, view);

ScheduledFuture<?> pf = ses.scheduleAtFixedRate(producer, 0, 1, TimeUnit.***SECONDS***);

ClientConsumer consumer1 = **new** ClientConsumer(1, custQ.take(), view);

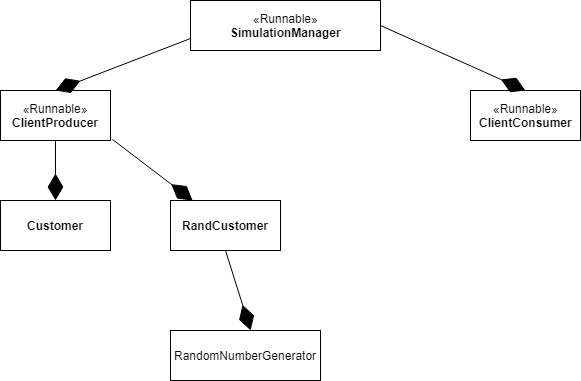
ScheduledFuture<?> cf1 = ses.scheduleAtFixedRate(consumer1, 0, 1, TimeUnit.***SECONDS***);

ClientConsumer consumer2 = **new** ClientConsumer(2, custQ.take(), view);

ScheduledFuture<?> cf2 = ses.scheduleAtFixedRate(consumer2, 1, 1, TimeUnit.***SECONDS***);

ClientConsumer consumer3 = **new** ClientConsumer(3, custQ.take(), view);

ScheduledFuture<?> cf3 = ses.scheduleAtFixedRate(consumer3, 2, 1, TimeUnit.***SECONDS***);



**4.Implementation:**

SimulationManager:

Is a class that implements Runnable interface and is the one that manages the produced and processed clients. It makes use of the ScheduledExecutorService which is part of the library java.util.concurrent.ScheduledExecutorService . It is an ExecutorService which can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution. Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to the ScheduledExecutorService. In order to use it I had to use its implementation in a thread pool, a ScheduledThreadPool with a starting pool of 4 threads. Because of the multithreading ,each queue needs a separate thread and I have three queues ,therefore need three threads . Also I need another thread for the production of clients, the ExecutorService will have to schedule four threads that are in the pool.

The simulation is running for a given time, input by the user in the user interface text field Running time .

ClienProducer:-implements Runnable and manages the producing of customers

Customer: contains all the information about a customer and this is reperesented by the fields id, arrivalTime, processingTime and the state( arrived or not) of the customer

ClientConsumer: implements Runnable , is a thread that simulates the serving of the customers by the cashiers

RandCustomer – has a static function which returns a Customer object with randomly generated arrival and processing times . This is its signature

**public** **static** Customer generateCustomer(Integer minAt, Integer maxAt ,Integer minPt, Integer maxPt) {

RandomNumberGenerator: is a class which has methods that return a number within a certain range ,in this case minimum arrival time and maximum arrival time and minimum and maximum processing time. Or generates an upper bounded number for the id of the Customer.

Graphical interface: The graphical interface has been created using Swing and the WindowBuilder plugin for displaying the built graphical components. The interface has been created with the users’ experience in mind and consists of just the input text fields for minimum arrival time, maximum arrival time, minimum processing time, maximum processing time, number of queues( which later I will change to be of variable length) and , also , the running time which is used simulate for a fixed period how the flow of incoming and served clients should behave. Also , each customer has a customer format : ID-AT-PT which is id of customer ,arrival time and processing time .These are displayed in each of the queues as the customers enter and wait to be processed.

**private** JTextField minArrivTime = **new** JTextField();

**private** JTextField maxArrivTime = **new** JTextField();

**private** JTextField minPtTime = **new** JTextField();

**private** JTextField maxPtTime = **new** JTextField();

**private** JTextField nbOfQs = **new** JTextField();

**private** JTextField simTimeTf = **new** JTextField();

**private** JButton startBtn = **new** JButton("START");

**private** DefaultListModel<String> modelQ1;

**private** DefaultListModel<String> modelQ2;

**private** DefaultListModel<String> modelQ3;

**private** JList<String> list1;

**private** JList<String> list2;

**private** JList<String> list3;

**Used Structures**

While the producer-consumer pattern enables producer and consumer code to

be decoupled from each other, their behavior is still coupled indirectly through the

shared work queue.

Threads may block, or pause, for several reasons: waiting for I/O completion,

waiting to acquire a lock, waiting to wake up from Thread.sleep, or waiting for

the result of a computation in another thread. When a thread blocks, it is usually

suspended and placed in one of the blocked thread states (BLOCKED, WAITING, or

Blocking and interruptible methods TIMED\_WAITING). The distinction between a blocking operation and an ordinary operation that merely takes a long time to finish is that a blocked thread must wait for an event that is beyond its control before it can proceed—the I/O completes, the lock becomes available, or the external computation finishes. When that external event occurs, the thread is placed back in the RUNNABLE state and becomes eligible again for scheduling.

The put and take methods of BlockingQueue throw the checked InterruptedException,

as do a number of other library methods such as Thread.sleep.

When a method can throw InterruptedException, it is telling you that it is a

blocking method, and further that if it is interrupted, it will make an effort to stop

blocking early.

**5.Results**

The application works without bugs on every test it has been given. For exemple, for the input minimum arrival interval = 1 , maximum arrival interval = 4, minimum service time = 3 , maximum service time = 14 , queues = 3 , running time = 20 , the output log is

86 arrives in 2

86 arrives in 1

86 arrives in 0

71 arrives in 0

26 arrives in 1

26 arrives in 0

Q:2 finished client : 71

38 arrives in 1

48 arrives in 1

38 arrives in 0

48 arrives in 0

65 arrives in 0

82 arrives in 1

Q:1 finished client : 86

82 arrives in 0

98 arrives in 2

98 arrives in 1

89 arrives in 1

98 arrives in 0

89 arrives in 0

30 arrives in 0

43 arrives in 0

13 arrives in 1

Q:1 finished client : 38

Which is the expected output, like in real life, because some clients do not get to be served due to closing of the queues at the market.

**6.Improvements**

I believe that using the ScheduledExecutorService is a major improvement in the logic of the application at it handles the submission of tasks to the threads ,handles the resource distribution of the threads and the activation times and idle times of the threads. I successfully integrated this service in my application.

**7.Conclusions:**

This assignment was a real challenge for myself having never worked with thread usage before. Thus, I began a process of documenting on this subject , going as far as reading the book recommended for this complex subject, Java Concurrency in Practice . Leafing through this book I could better decide on which concurrent structures to use for this assignment. Though it has taken me longer to understand how to use for example the ExecutorService it has proven to be quite useful and I am sure I will use it in my future work with threads.

**8.Bibliography**

**1.** Java Concurrency In Practice, Brian Göetz, Tim Peierls ,Joshua Bloch, Joseph Bowbeer, David Holmes ,Doug Lea