**Technical University of Cluj-Napoca  
Faculty of Automation and Computer Science  
Computer Science Department**

***Drimbarean Maria Cristina***

***Group:30425***



**Table of contents**

1. **Assignment objectives**

1.1General view

1.2 Personal approach

**2. The analysis**

2.1. Otput compuations

2.2 Modeling

2.3. Scenarios

2.4 Use cases analysis

**3. Design**.

3.1. UML diagram

3.2. Data structures

3.3. Class projection

**4. Implementation and testing**

**5.Results**

**7. Conclusions**

**8. Bibliography**

**1.Assignment objective**

**1.1 General view**

*Task was described with the following statement:*

*Description:*

*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.*

*Input data: - Minimum and maximum interval of arriving time between clients; - Minimum and maximum service time; - Number of queues; - Simulation interval; - Other information you may consider necessary;*

*Minimal output: - Average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the simulation interval and for a specified interval; - Log of events and main system data; - Queue evolution; - Peak hour for the simulation interval;*

*1.2 Personal approach*

*As i needed to simulate the supermarketcheckouts i considered them as having a queue of clients waiting in line, and the simulation interval for each checkout as the sum of the service they have done. The waiting time for each client is the differance between exit time, and arrival time. Also the number of chekouts are fixed and they run separately. Considering the fact that it should represent a real model , each client that arrived should choose the queue that is the most empty. The clients come within a specified amount of time, and also can have a random service time generated between some values.*

***2.The analysis***

***2.1Output computations***

*As the checkouts run simultaneously, they have their service time, waiting time and empty time are generated in their class in comparison with the overall time. We should pay attention on synchronization as multithreading will be certaintly used. All operations involving global variables should be performed atomically.*

**2.2Modeling**

We will only need few classes that reprezent real objects. We will make differance between the oanes that ar supposed to perform something and the ones that are supposed to be operated on. The first model is for a Client, a separate class should represent it and will have some simple instance variables and a few getters and setters ( service time,id).

For the supermarket checkout a separate class will have it’s own queue of clients and should try to service them in line, this is a runnable class. A class for generateing clients will also be needed as it is very hard to input them from the keypad. So a new runnable class should be implemented to generate rondomly clients with a certain interval between them.

A scheduler class will take the responsibility to put the clients in the most empty queue given it’s number of waiting line. This is done separately from generating the clients.

A new class is given to run the logic of the application, this Simulation class should contain all necessary objects to perform the simulation.

Last but not least a new class should be given the duty to count the number of clients arriving at the supermarket in a certain amount of time, and therfore determining the peakhour. This is a new runnable class that should count each 10 seconds for exemple.

**2.3Scenarios**

**Title**: Simulate queue evolution

**Resume**: After inserting the input parameters in the user interface (number of queues, min. and max. serving time, min. and max. arrival interval, and the simulation interval), the user will see a short log in the lower part of the application window, while the complete log is available in the console.

**Actors**: User

**Scenarios**:

1. Preconditions: application is ready to use
2. Normal scenario:
3. User successfully inserts the input parameters;
4. User pushes the "Start!" button to start the simulation;
5. Application displays the short log;
6. The queue evolution is showed in console
7. Alternative scenario:
8. User types wrong data in the input fields;
9. User pushes the "Start!" button attempting to start the simulation;
10. An error message is displayed;
11. Another change of inserting/correcting the input data is provided.

**2.4 Use cases**

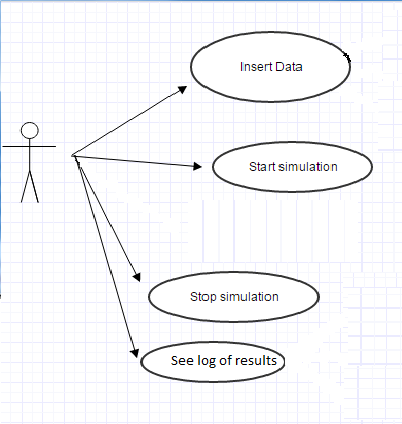
The user will see in console when a client arrived at a certain checkout, when a certain client has exited and the state of the queues. Also when the peak hour is ready it should be displayed in the console. Then the user cand pause and restart the application to see it’s partial results. And then press stop to see the final results in the text area.

**3.Projection**

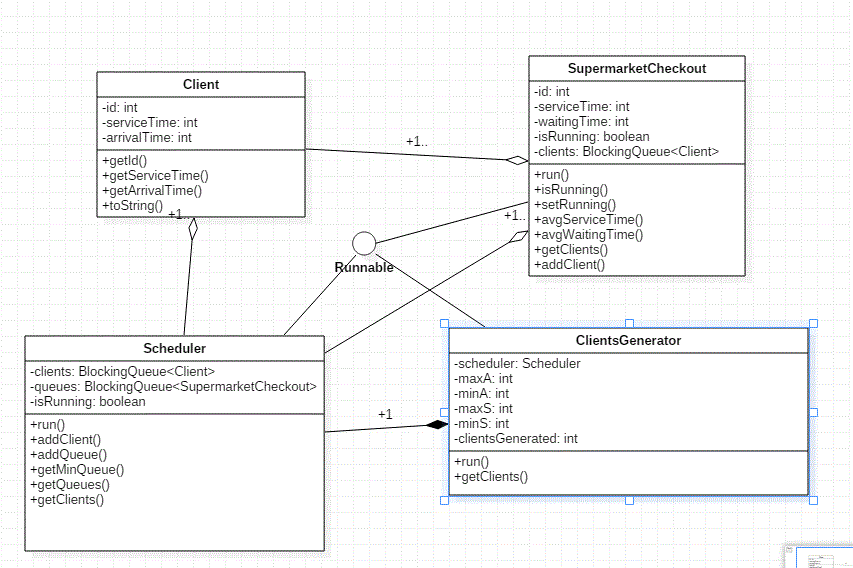
3.1 UML Diagrams

a)Use case

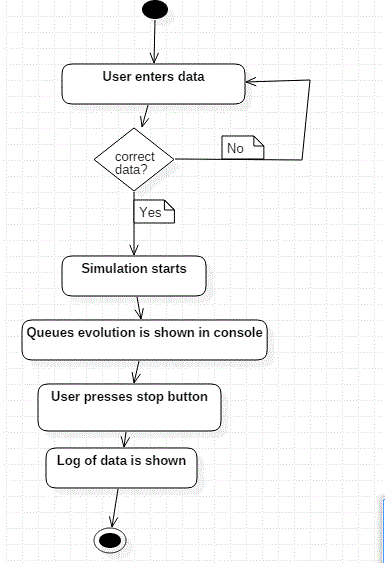
The user is selecting the behaviour of the app



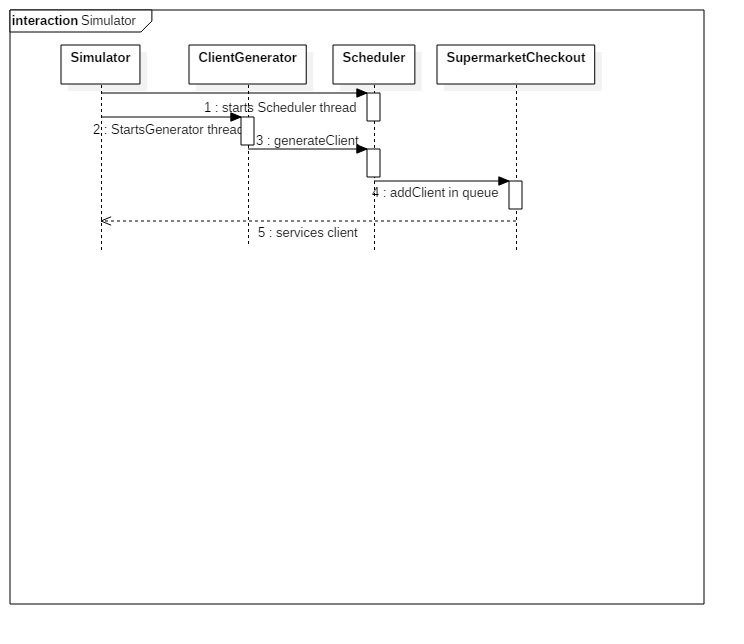
b)Class diagram



c)Activity Diagram



d)Sequence diagram



**3.2Data structures**

A [Queue](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

|  |  |
| --- | --- |
|  | *Blocks* |
| **Insert** | [put(e)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#put(E)) |
| **Remove** | [take()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#take()) |
| **Examine** | *not applicable* |

A BlockingQueue does not accept null elements. Implementations throw NullPointerException on attempts to add, put or offer a null. A null is used as a sentinel value to indicate failure of poll operations.

A BlockingQueue may be capacity bounded. At any given time it may have a remainingCapacity beyond which no additional elements can be put without blocking. A BlockingQueue without any intrinsic capacity constraints always reports a remaining capacity ofInteger.MAX\_VALUE.

BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) interface. So, for example, it is possible to remove an arbitrary element from a queue using remove(x). However, such operations are in general *not* performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled.

BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. However, the *bulk* Collection operations addAll, containsAll, retainAll and removeAll are *not* necessarily performed atomically unless specified otherwise in an implementation. So it is possible, for example, for addAll(c) to fail (throwing an exception) after adding only some of the elements in c.

A BlockingQueue does *not* intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added. The needs and usage of such features tend to be implementation-dependent. For example, a common tactic is for producers to insert special *end-of-stream* or *poison* objects, that are interpreted accordingly when taken by consumers.

**3.3 Class projection**

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the java.util.concurrent packages.

The application is devided in 2 packages: One for models **package** pt.processingQueues.simulation;, and one for simulation **package** pt.processingQueues.simulation;

There ar 5 classes: 

ClientGenerator, PeakHour , Client , Scheduler, SupermarketCheckout .

The Client class represents the model to pe worked on, has an id and a service time generated from the Client generator:

**public** **class** Client {

**private** **int** id;

**private** **int** serviceTime;

**public** Client(){}

**public** Client(**int** id,**int** serviceTime){

**this**.id=id;

**this**.serviceTime=serviceTime;

}}

ClientGenerator takes the inputs from keyboard and generates clients in a certain interval with certain service times:

@Override

**public** **void** run() {

***LOGGER***.info("Logger Name: "+***LOGGER***.getName());

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

**int** arrivalTime;

**int** processingTime;

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

**while** (isRunning) {

arrivalTime = r.nextInt(maxArrivalTime - minArrivalTime) + minArrivalTime;

**try** {

Thread.*sleep*(arrivalTime \* 1000);

processingTime = r.nextInt(maxServiceTime - minServiceTime) + minServiceTime;

clientsGenerated++;

clientsGeneratedSince++;

scheduler.addClient(**new** Client(clientsGenerated, processingTime));

} **catch** (InterruptedException ex) {

***LOGGER***.log(Level.***SEVERE***, "Cannot add generated client", ex);

}

}

}

}

The class implements the Runnable interface to assure that it will run separately by a thread. The run method has the logic of this class.

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

Scheduler class has a BlockingQueue of clients and finds out were the minimum queue is , then assigning the client were it should normally go in real life.

**public** **class** Scheduler **implements** Runnable{

**private** **static** **final** Logger ***LOGGER*** = Logger.*getLogger*(Scheduler.**class**.getName());

**private** ArrayList<SupermarketCheckout> queues=**new** ArrayList<SupermarketCheckout>();

**private** BlockingQueue<Client> clients=**new** ArrayBlockingQueue<Client>(100);

**public** **void** run() {

***LOGGER***.info("Logger Name: "+***LOGGER***.getName());

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

**while** (isRunning) {

**if** (queues.size() > 0 && clients.size() > 0) {

SupermarketCheckout queue = getMin();

Client client=**new** Client();

**try** {

client = clients.take();

} **catch** (InterruptedException e) {

***LOGGER***.log(Level.***SEVERE***, "Cannot take a client from queue", e);

}

queue.addClient(client);

}

}

}

}

Also it has methods for adding a checkout, because it also uses an arraylist of checkouts.

The SupermarketCheckout class has it’s own blocking queue of clients added by scheduler, and it service them:

@Override

**public** **void** run() {

***LOGGER***.info("Logger Name: "+***LOGGER***.getName()+id);

Client client=**new** Client();

**int** delay;

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

**while** (isRunning) {

**if** (clients.size() > 0) {

System.***out***.println(**this**.listToString());

**try** {

client = clients.take();

} **catch** (InterruptedException ex) {

System.***out***.println("EXCEPTION " + ex);

}

clientsServed++;

delay = client.getServiceTime();

**try** {

Thread.*sleep*(delay \* 1000);

waitingTime += serviceTime;

serviceTime+= delay;

System.***out***.println(**this**.toString()+" : "+client.toString()+" checked-out!");

} **catch** (InterruptedException ex) {

***LOGGER***.log(Level.***SEVERE***, "Cannot take a client from queue", ex);

}

}

**else** {

//System.out.println(this.toString()+" is empty!\n");

//this.setRunning(false);

}

The classes used for testing are in the simulation package: Simulation , TestQueue , TestScheduler , and also a graphical user interface for helping simulation presented in te implementation and testing chapter.

For the simulation class:

**public** **class** Simulation {

**private** **int** simulationTime=2000;

**private** **int** queuesNumber=3;

**private** **int** minS=1,maxS=3,minA=1,maxA=3;

**protected** ClientGenerator clientGenerator;

**protected** Scheduler scheduler;

**protected** SupermarketCheckout[] queues=**new** SupermarketCheckout[30];

**protected** Thread[] threads=**new** Thread[30];

**protected** Thread schedulerThread;

**protected** Thread generatorThread;

**protected** PeakHour peakHour;

**protected** Thread peakHourThread;

**public** **void** initSimulation(){

scheduler=**new** Scheduler();

schedulerThread=**new** Thread(scheduler);

**for** (**int** i=0;i<queuesNumber;i++)

{

queues[i]=**new** SupermarketCheckout(i+1);

threads[i]=**new** Thread(queues[i]);

threads[i].start();

scheduler.addSupermarketCheckout(queues[i]);

}

clientGenerator=**new** ClientGenerator(maxA,minA,maxS,minS,scheduler);

peakHour = **new** PeakHour(clientGenerator);

generatorThread = **new** Thread(clientGenerator);

peakHourThread = **new** Thread(peakHour);

schedulerThread.start();

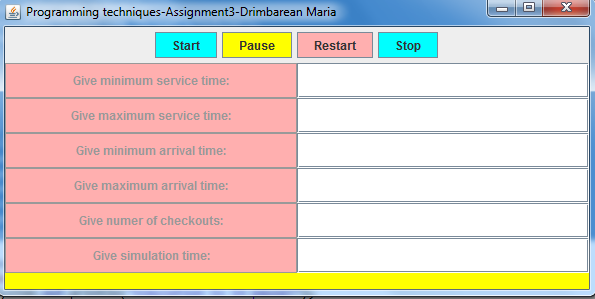
generatorThread.start();

peakHourThread.start();

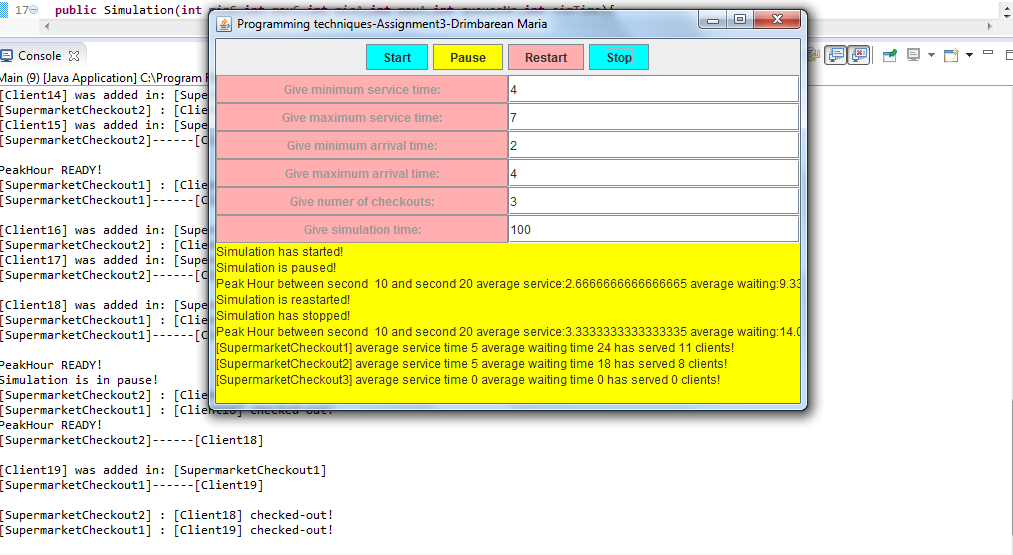
}

We have here the simulation logic for the program were each needed runnable class is assigned to a thread.

**4.Implementation an testing**



The input window lets you interact with the Simulation class, and given the corect inputs it starts the simulation.



**5.Results**

**6.Conclusions**

Without careful planning and algorithm design, even the simplest application can become complex. On the other hand, having developed the design stages in details and split the problem correctly, even the most difficult application can be less challenging than a familiar game.

At first sight, a queues simulatot does not sound like a very complex software. The mathematical knowledge is apparently easy, queues being easy objects to model and operate with. However, thorough the numerous development stages mentioned in the present documentation, it is clear that even such simple problems require careful solution-finding, such as the correct data-structure, or the correct mathematical approach of the algorithms.

**Further development**:

* display a real-time log of events;
* show an animation of customers waiting in queues, moving to other queues, and leaving (after being processed);
* develop an alogirthm the chooses when to open and close queues based on the input parameters, to allow a better distribution of the customers;
* implement functionality to pause and resume the simulation, before the simulation time reached the simulation end value.

7. *Bibliography*

1. <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.ht>
2. http://www.joda.org/joda-time/apidocs/index.html
3. http://www.youtube.com/watch?v=O\_Ojfq-OIpM
4. http://docs.oracle.com/javase/tutorial/essential/concurrency/
5. Effective Java 2nd Edition (Concurrency)