Programming Techniques

Assignment 3

Student : Rujita Alexandra

Group : 30422

Contents

[1. Problem specification. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3](#_Toc448226874)

[2. Problem analysis , modeling, scenarios , use cases. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .3](#_Toc448226875)

[2.1. Problem analysis 3](#_Toc448226876)

[2.2. Modeling 4](#_Toc448226877)

[2.3. Use cases 4](#_Toc448226878)

[3. Design. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5](#_Toc448226879)

[3.1. UML Class Diagram 5](#_Toc448226880)

[3.1.1. CashRegister Class 6](#_Toc448226881)

[3.1.2. FileOp Class 6](#_Toc448226882)

[3.1.3. Generator Class 6](#_Toc448226883)

[3.1.4. GUI Class 6](#_Toc448226884)

[3.1.5. Server Class 6](#_Toc448226885)

[3.1.6. Task Class 7](#_Toc448226886)

[3.1.7. mainC Class 7](#_Toc448226887)

[3.2. Data structures 7](#_Toc448226888)

[3.3. Arhitecture 7](#_Toc448226889)

[3.4. Relationships 7](#_Toc448226890)

[3.5. Algorithms 7](#_Toc448226891)

[3.6. User interface 8](#_Toc448226892)

[4. Implementation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8](#_Toc448226893)

[5. Further developments and importance. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11](#_Toc448226894)

[6. References. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11](#_Toc448226895)

1. Problem specification

Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time.

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;

On the user interface you will be able to see:

* Average of waiting time, service time
* Log of events and main system data – in a text file
* Queue evolution;
* Peak hour for the simulation interval;

1. Problem analysis , modeling , scenarios , use cases

## Problem analysis

The problem can be modeled in several ways, it can take the information it needs from a text file, from keyboard or be hardcoded. In my impelementation I chose the first way, because the data can be easily modified without having to change any code, and also because it is more flexible. The result of the operations can be printed in the console, on an user interface or in a text file. Also the summary of the execution of the program is written in a text file.

To simulate the real life store we use the java threads to represent the queues. To slow down the process of a thread in order to resemble a cashier we use the sleep() method.

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The sleep method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements, as with the SimpleThreads example in a later section.

Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts, as we'll see in a later section. In any case, you cannot assume that invoking sleep will suspend the thread for precisely the time period specified.

## Modeling

In this implementation I chose to read all the input data from a text file, every element is stored on a different line so the reading is done easier, and I chose to show the execution of the program on an user interface and at the end of the „program” the summary is written in a text file, for each queue ( nr of clients and average time of waiting ).

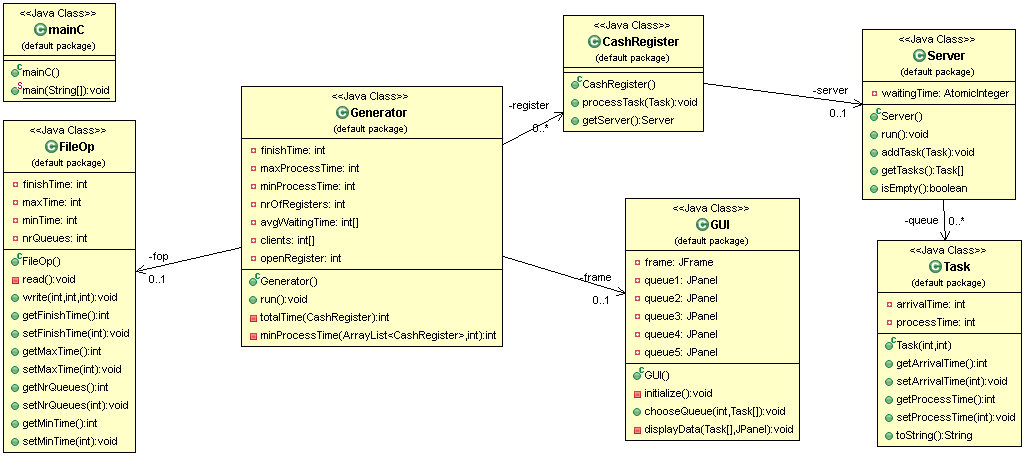
For the representation of a client I made a new class ( task ) that will hold all the necessary information.

## Use cases

Clients ( tasks ) come at a moment of time, because we can aproximate their processesing time we distribute the clients at the queue with the smallest waiting time. Also when a certain time limit is reached a new queue ( thread ) is opened and the last client who arrived is given to that new opened cash register. Every time a new customer arrives he will chose the cash register with the minimal waiting time.

1. Design

## UML Class Diagram

The UML diagram is a class diagram that shows a set of classes, interfaces and collaborations and their relationships. Class diagrams may also contain packages or sub-systems, both of which are used to group elements of the model.

### CashRegister Class

This class is the abstract model of a cash register , it creates a new thread which represents the cash register , after creating it new tasks can be added to it for precessing. The processing represents an amount of time ( in seconds ) in which the task has to be executed which is equivalent with a sleep() for the thread ( 1000 \* processesing time ).

### FileOp Class

This class purpose is to work with text files. It does the reading from file ( line by line ) and converting the strings to integer numbers and storing them in an array, after all the lines are read it sets the values for the finishing time ( until what hour the store is open ) , the max time ( the maximum limit a customer can take to process ), the min time ( how little a customer can stay at the cash register ) and the nr of cash registers ( nr of threads ).

It also writes in the file, when all the customers have been processed and the threads are empty , the summary of the work that has been done in that day ( nr of clients per queue and the average waiting time ).

### Generator Class

This class is the abstract model of a store. It can open new registers ( create new CashRegister objects ) , welcome new customers ( create new Task objects , with random values for process time ) and distribute them to the most appropriate cash register. It also can calculate how much time it takes a certain queue to be done with its current customers and which is the queue with the minimal waiting time. A new cash register is opened ( a thread is created ) only when the time of the cash register with the minimal waiting time is greater than the threshold for opening a new cash register.

Another purpose of this class is to comunicate with the user interface and „tell” it to remodel what the user sees. It does this for every queue and every new client or everytime a client is proccessed, also it shows what all the queues are doing in parrallell and not one at a time.

### GUI Class

As the name sugests this class single purpose is to ilustrate swing class components. It has 5 labels and 5 panels. The panels represent the queues that are made visible only when we open that certain cash register, and is repainted after the execution of each task. It is a minimal representation of a real life store.

### Server Class

This class is the handling class. It adds tasks to the queue ( BlockingQueue ) , it verifies if the queue is empty or not ( if it had been opened ) , it returns the array of information about the tasks and it puts the thread to sleep for the specified amount of time ( 1000 \* processTime of that particular task ). This task is an intermidiar between the store ( generator object ) and the thread ( cashRegister object ) .

### Task Class

This class is the abstract model of a client. It has as attributes the arrival time and the processing time, normally the last one would be unknown but for this project we „know” it , but to maintain a real life aproach we set to it a random calculated value.

### mainC Class

It is the class that starts the application. It creates a new generator object and a main thread that holds the generator object.

## Data structures

There are 3 main data structures in this project. The base one is the task ( which is the customer ) , the next one is the CashRegister that processes the tasks and the Server that comunicates between the customers and cashRegister. The data structure that runs the hole store is the Generator that evenly distributes all the new clients to its cash registers.

The tasks are hold in a blockingQueue in the cashRegister. All the cash registers are stored in an array list.

## Arhitecture

The arhitecture of the project is a basic one, following all the object oriented programming rules ( **Encapsulation, Abstraction and Polymorphism ).**

## Relationships

There is a strong relationship between the Generator class and the CashRegister Class, but also between it and the GUI class and FileOp class ( used to communicate with the user ).

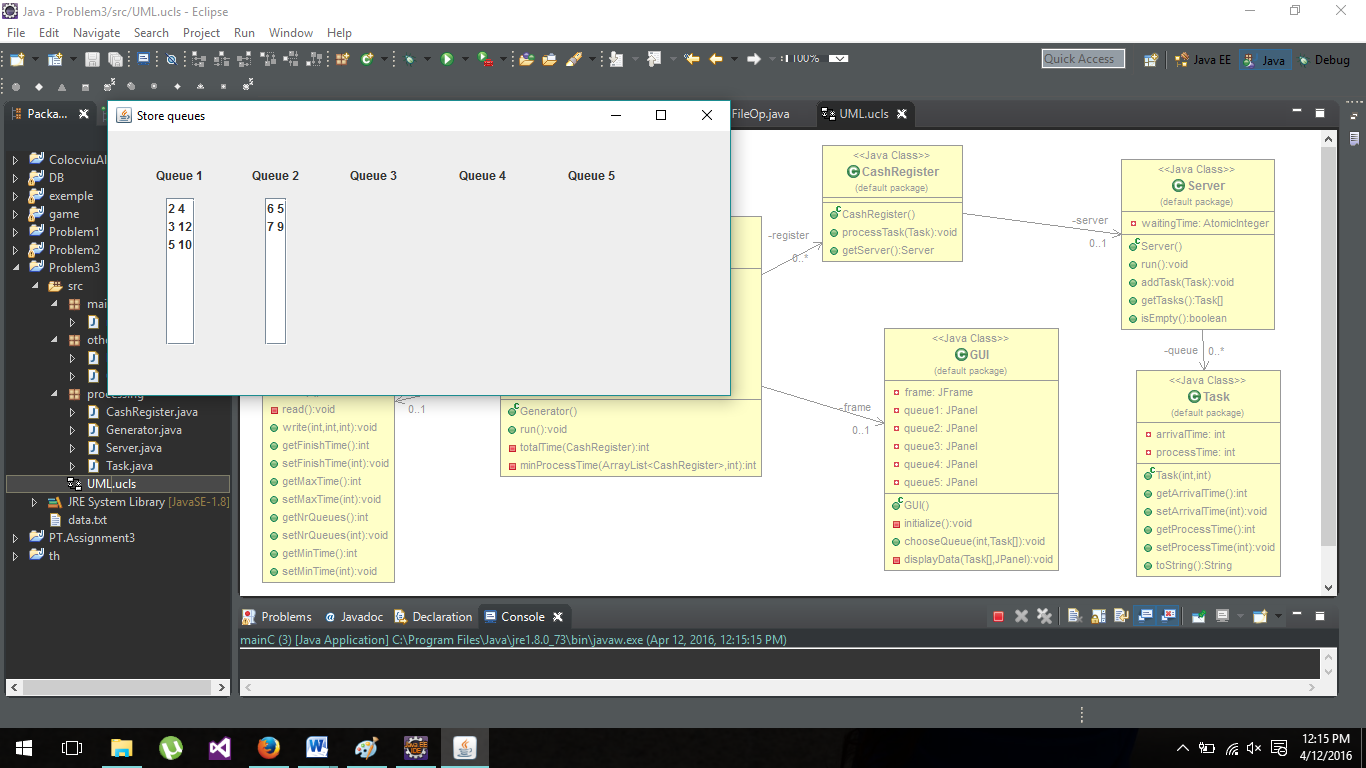
A strong relationship is also between the CashRegister class and the Server class, and further between the Server class and the Task class.

Following the object oriented programming rules there is a weak coupling and strong cohesion between classes.

## Algorithms

The algoritms that stand at the base of this project are pretty basic. We create new threads only when we really need one, based on a find the minimum waiting time algortihm. Otherwise we process the clients we already have in the queue.

## User interface



We can see on the user interface the queues that are open ( 1 and 2 ) and how many queues the store has available ( even if for the moment are closed ).

The first number in the column is the arrival time and the second number is the processing time. We can see that the second queue was opened only because the threshold for maximum waiting time was reached ( that is 15 \* 5 minutes ) and the rest of the queues where not opened yet because we still have a queue that has a waiting time smaller than the threshold.

1. Implementation

For this project we used the concept of threads. A thread, in the context of Java, is the path followed when executing a program. All Java programs have at least one thread, known as the main thread, which is created by the JVM at the program’s start, when the main () method is invoked with the main thread. In Java, creating a thread is accomplished by implementing an interface and extending a class. Every Java thread is created and controlled by the java . lang . Thread class.  
 When a thread is created, it is assigned a priority. The thread with higher priority is executed first, followed by lower-priority threads. The JVM stops executing threads under either of the following conditions: If the exit method has been invoked and authorized by the security manager ; All the daemon threads of the program have died.

In the Task class we have the following methods :

public int getArrivalTime () {

return arrivalTime;

}

public void setArrivalTime( int arrivalTime ) {

this.arrivalTime = arrivalTime;

}

public int getProcessTime() {

return processTime;

}

public void setProcessTime( int processTime ) {

this.processTime = processTime;

}

These four methods set and get the values for the attributes that define a task.

public String toString() {

return String.*valueOf* (arrivalTime ) + " " + String.*valueOf*( processTime ) ;

And the last one represents the method used for displaying the information in the user interface.

In the Server class we have :

**public** **Server** () {

waitingTime = **new** AtomicInteger(0);

queue = **new** LinkedBlockingQueue < Task >();

}

In the constructor we initialize the queue that will hold all the tasks ( clients ). And the waiting time that is an atomic integer. The AtomicInteger provides atomic ( thread safe ) operations on an integer value.

***@Override***

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

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

**Task** **t**;

**try** {

t = queue.take();

**Thread**.*sleep*(t.getProcessTime()\*1000);

waitingTime.addAndGet((-1) \* t.getProcessTime());

} **catch** (**InterruptedException** **e**) {

e.printStackTrace();

}

}

}

This method adds the tasks to the queue and sets the thread’s sleep period.

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

In the Generator Class we have:

**private** **int** **totalTime**( **CashRegister** cr ) {

**int** **time** = 0;

**for** (**Task** **t** : cr.getServer().getTasks())

time + = t.getProcessTime();

**return** time ;

}

The method above calculates how much time a cash register requires to finish all the clients, by passing through all the clients and retrieving the processing time, that it has at the current moment and return the calculated value.

**private** **int** **minProcessTime** ( **ArrayList**<CashRegister> cr, **int** pos ) {

**int** **min** = **Integer**.***MAX\_VALUE***, **minCrt** = 0, **posMin** = 0, **i** = 0;

**for** (**CashRegister** **c** : cr) {

**if** ( pos < nrOfRegisters ) {

**if** ( !c.getServer().isEmpty() ) {

minCrt = totalTime( c );

**if** ( minCrt < min ) {

min = minCrt;

posMin = i;

}

}

} **else** {

minCrt = totalTime( c );

**if** ( minCrt < min ) {

min = minCrt;

posMin = i;

}

}

i++;

}

**return** posMin ;

}

This method returns the position ( or the number ) of the cash register with the minimal waiting time, it does this only if the cash register was opened today , if not, it does not take it into consideration and goes further in the list.

In the FileOp Class we have :

**public** **void** **write**(**int** cashReg, **int** avgTime, **int** clients ) **throws** **IOException** {

**File** **fout** = **new** File("Summary.txt");

**if**(!fout.exists() )

fout.createNewFile();

**PrintWriter** **out** = **new** PrintWriter(**new** FileWriter(fout, **true**));

out.append("Cash register no " + cashReg + " had "+clients+" customers, with average waiting time: "+avgTime+"\n");

out.close();

}

This method appends to the already created file, all the information about how the day evolved.

1. Further developments and importance

The project can be improved by adding an interval that will resemble the peak hour of a real life store; by not knowing how much time it takes an customer to be processed and base our distribution on how many customers are at a certain queue; when a new queue is opened all the current clients ( except the first one; the one that is currently processed ) be evenly distributed to the open queues ( that are now more ).

The importance of this project is to learn about the threads in java, the concurrency and how it works, how to synchronize the threads and send information between them.

Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system   
  
In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For example, when you use a word processor you perform a many different tasks such as printing, spell checking and so on. Multithreaded software treats each process as a separate program.  
  
In Java, the Java Virtual Machine (JVM) allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time

1. References

* [www.cs.unicam.it/culmone/?download=java\_concurrency\_in\_practice.pdf](http://www.cs.unicam.it/culmone/?download=java_concurrency_in_practice.pdf)
* <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
* <http://www.tutorialspoint.com/java/util/timer_schedule_period.htm>
* http://www.javacodegeeks.com/2013/01/java-thread-pool-example-using-executors-and-threadpoolexecutor.html