

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

Assignment 3 - Threads

-JAVA Project-

Coordinating: Claudia Pop

Student: Onaciu Andreea

Group 30424

Table of Contents

[1. Assignment Objectiv 3](#_Toc448225664)

[2. Problem Analysis 4](#_Toc448225665)

[2.1 Modeling 4](#_Toc448225666)

[2.2.1 Task 5](#_Toc448225667)

[2.2.2 Server 6](#_Toc448225668)

[2.2.3 Schedular 7](#_Toc448225669)

[2.2.4 Simulator 8](#_Toc448225670)

[2.2.5 SimulatorFrame 10](#_Toc448225671)

[2.2.6 Controller Part 11](#_Toc448225672)

[2.2 Scenarios 11](#_Toc448225673)

[2.3 Use Cases 11](#_Toc448225674)

[3.Design (UML Diagram) 12](#_Toc448225675)

[4.Implementation and testing 13](#_Toc448225676)

[5.Results 14](#_Toc448225677)

[6. Conclusions 14](#_Toc448225678)

[7.Future developments of the project 14](#_Toc448225679)

[8.Biography 14](#_Toc448225680)

## Assignment Objectiv

The task of the assignment was:

*“ TP Lab –Homework 3 Objective Design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time. 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;”*

In order to solve this task first I had to understand how threads work and how could I implement threads in Java. After having an understanding about these principles I had to find a logical and simple way to organize the project so as to perform the desired operations in a correct manier.

Threads:

A thread is an independent set of values for the processor registers (for a single core). Since this includes the Instruction Pointer (aka Program Counter), it controls what executes in what order. It also includes the Stack Pointer, which had better point to a unique area of memory for each thread or else they will interfere with each other.

Threads are the software unit affected by control flow (function call, loop, goto), because those instructions operate on the Instruction Pointer, and that belongs to a particular thread. Threads are often scheduled according to some prioritization scheme (although it's possible to design a system with one thread per processor core, in which case every thread is always running and no scheduling is needed).

In fact the value of the Instruction Pointer and the instruction stored at that location is sufficient to determine a new value for the Instruction Pointer. For most instructions, this simply advances the IP by the size of the instruction, but control flow instructions change the IP in other, predictable ways. The sequence of values the IP takes on forms a path of execution weaving through the program code, giving rise to the name "thread".

Threads in Java:

Java's creators have graciously designed two ways of creating threads: implementing an interface and extending a class. Extending a class is the way Java inherits methods and variables from a parent class. In this case, one can only extend or inherit from a single parent class. This limitation within Java can be overcome by implementing interfaces, which is the most common way to create threads. (Note that the act of inheriting merely allows the class to be run as a thread. It is up to the class to start() execution, etc.)

## Problem Analysis

## 2.1 Modeling

The process of modeling the problem begins with choosing a design pattern suitable for the requirements of the project. Given the fact that the assignments ask for a graphical user interface, one design pattern that matches these conditions is Model –View – Controller.

When considering the problem as a way to simulate a real behavior, the class Simulator is the first that comes in mind. The task ask for simulating a *q*ueuing based systems queuing based systems, so we have to have a class that holds the queue and a class that holds the items in the queue : Server and Task. The tasks are generated by the simulator and have to be dispatch to a server, choosing to which server to dispatch the task, it cannot be the role of the simulator, cause it has another meaning, so we need another class, which we will call :Scheldular. The scheldular contains all the servers and receives a task from the simulator and then decided to which server to give the task.

For the GUI part we need a frame, which we will call SimulatorFrame. This frame must have a method which comunicates with the simulator and updates the information after adding a new task. Because we want that the information needed to perform the simulation, like : number of queues, simulation interval and other things, to be change easily the frame will contain some text fields which will allow the user to set the simulation characteristics everytime she/he wants to start the simulation. For doing this we need a controller with action listener for the start button, this controller is named: SimulationController. Having all of these classes the problem now is divided in smaller tasks which correspond to each class.

Because the number of classes is smaller and the classes are strongly interconnected, I decided to place them in one single package the default one.

### 2.2.1 Task

The task can be seen as the model in this project, and represent the activity which has to be perfom by the server for an amount of time. The task its self doesn’t not have a behavior or some functionality, so it will have just some attributes, constructor, setters and getters. Also the task with its characteristics has to be display onto the frame so the class needs to have a toString method.

Attributes:

Each task is described by an arrival time, which will represent the time when the task was generated by the simulator, and also a processing time. The processing time represents the amount of time needed to complete the task. This will be set randomly by the simulator, but will have to be in the interval decided by the user via the SimulatorFrame. Because the attributes are related to time, their time will be DateTime and Period, respectively. This two classes are provided by the JodaTime package, and are very useful for simulating a real behavior.

Capabilities:

-besides the constructor, setters and getters, the Polynomial class must be able to provide a string representation (method toString), which will consist in displaying the arrival time and the processing time.

### 2.2.2 Server

The server is the class that holds the tasks. And because the tasks will disappear from the server when they are processed, and can enter any time in the server if there is enough space ( characteristic provided by the user at the interface ) the server must implement the Runnable interface. So this means it has to write the run() method.

Attributes:

The server contains a list of tasks, because of the dynamic behavior of this list, a BlockingQueue will represent a perfect choice for this class. A BlockingQueue is a synchronized queue, which will not block or return erros if there appear some synchronization problems. Also the Server class has the waitingTime characteristic which represents the amount of time which will pass till the queue will be empty. It is a way to represent how much time the last client/ task has to wait until its requirements will be processed. Because of this dynamic behavior the waiting time has to be an AtomicInteger variable, so as to be not incorrectly change by the threads accessing it.

Capabilities:

The class contains setters , getters and constructor.

The run method will simulate the way in which the tasks are processed. This means a task will be taken from the queue and the thread will be put to sleep for the amount of time which takes to process the task, which is a characteristic of the task.

After the task is processed the waiting time will be decrease with this processing time.

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

// **TODO** Auto-generated method stub

Task curT = **null**;

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

**try** {

curT = queue.take();

System.***out***.println("Start processing task : " +curT);

Thread.*sleep*(curT.getProcessTime());

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

waitingTime.addAndGet((-1)\*curT.processTime.getSeconds());

}

Another method is the addTask method which adds tasks to the queue increasing the waiting time with the randomly generated processing time of the task.

**void** addTask(Task t){

queue.add(t);

waitingTime.addAndGet(t.processTime.getSeconds());

}

The class also contains a method to get the tasks, this will be used in the display part.

A method which returns the size of the queue so as the scheduler to know if it can put another task on the server or not.

### 2.2.3 Schedular

The scheduler is the one which sets each task to a server. This is done according to the following algorithm : first it puts the task to the server which has the minimum number of tasks on it, if all the servers are full, a new server will be open, and the task will be place in that one. But if there cannot be open another server, due to the maximum number of servers set by the user in the frame, the task will not be processed.

Attributes:

* The scheduler contains a list of servers, a number of Queues, representing the maximum number of opened servers. Maximum number of tasks on a server, and an AtomicInteger variable s which will compute the sum of waiting time, for the statistics.

Capabilities:

The most important method of the scheduler class is dispatchThreadOnServer which performs exactly what it says.

**void** dispatchThreadOnServer(Task t) {

System.***out***.println("Dispatch task on server :" + t);

AtomicInteger index2 = **new** AtomicInteger(0);

index2=serverWithMinimumNumberOfClients();

**if** (index2.intValue()!=999){

computeSumOfWaitingTime(servers.get(index2.intValue()));

servers.get(index2.intValue()).addTask(t);

}

**else** **if** (servers.size()<numberOfQueues)

{

Server server = **new** Server();

server.addTask(t);

servers.add(server);

Thread th = **new** Thread(server);

th.start();

computeSumOfWaitingTime(server);

}

-method to compute the sum of waiting time;

-method to get the index of the server to the minimum number of clients;

-method to return it all the servers are empty so as to stop the simulation;

-method which returns the tasks, for displaying them on the frame;

-method which returns the number of tasks// for statistics;

### 2.2.4 Simulator

The simulator holds the main part of the projects. It simulates in real time the behavior. So it implements the runnable interface and contains the run() method. The simulator will have to take the information from the frame and according to this information to simulate *a place for a "client" to wait before receiving a "service".*

Attributes:

* Time- which represents the simulation time, how much the simulation takes
* minProcTime- minimum time to process a task;
* maxProcTime- maximum time to process a task;
* s- an instance of the scheduler;
* frame- and instance of the frame;
* averageWaitingTime- a variable to compute the average of waiting time for the statistics;
* processingTime- a variable to compute the sum of processing time for the statistics;
* maxNumberOfTasks- the maximum number of tasks in a queue;
* peakHour- the time in the simulation when the servers were the most full;l
* Capabilities:

-setters and getters;

-constructor;

- the run method : **public** **void** run() {

// **TODO** Auto-generated method stub

DateTime currentTime =**new** DateTime();// 0;// jodatIME;

**while** (currentTime.isBefore(time)) {

currentTime=**new** DateTime();

**int** processTime = (**int**) (Math.*random*() \* (maxProcTime - minProcTime) + minProcTime);

Period processTimeD=**new** Period(processTime\*1000);

Task t = **new** Task(currentTime, processTimeD);

processingTime.addAndGet(processTime);

**if** (s.getNumberOfTasks()>maxNumberOfTasks)

{

maxNumberOfTasks=s.getNumberOfTasks();

peakHour=currentTime;

}

System.***out***.println("Generate task at: " + t);

s.dispatchThreadOnServer(t);

frame.displayData(s.getTasks());

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**while** (!s.isDone()) {

frame.displayData(s.getTasks());

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

frame.displayData(s.getTasks());

averageWaitingTime=s.getSumOfWaitingTime()/time\_limit;

JOptionPane.*showMessageDialog*(**null**, "The avgWaitingTime="+averageWaitingTime+"\n"+

"The proccesingTime="+ getProcessTime()+"\n"

+"The peak hour was "+ peakHour.getHourOfDay()+":"+peakHour.getMinuteOfHour()+":"+peakHour.getSecondOfMinute());

}

-methods for computing the statistics;

### 2.2.5 SimulatorFrame

Another part of the project was building a friendly interface for the user to be able to set the some settings for the simulation. These are :

-maximum number of servers;

-minimum time of service;

-maximum time of service;

-simulation time;

-maximum number of tasks per server;

All these characteristics can be set in some text fields and after inserting all of them the user must press the start button, so as the simulation begins with the desired attributes set.

Besides the design part, the class has also a method to update the frame in each moment a new task is added, this method is called display data and looks like this :

**void** displayData(ArrayList<Task[]> tasks) {

contentPanel.removeAll();

contentPanel.revalidate();

**for** (Task[] task : tasks) {

**if** (task.length != 0) {

JList<Task> list = **new** JList<Task>(task);

JScrollPane sp = **new** JScrollPane(list);

contentPanel.add(sp);

}

}

contentPanel.repaint();

contentPanel.revalidate();

}

The frame contains setters and getters for all the text fields, and sets a listener for the start button

### 2.2.6 Controller Part

For making a link between the model and the view, the controller part is done via the SimulatorController. It has an instance of the SimulationFrame and an inner action listener class for the start button.

The inner class has the method action listener, which acts when the start button is press and creates an instance of the simulation, so as to start the simulation only when the button is pressed and all the necessary information from the text fields are filled by the user.

**public** **void** actionPerformed(ActionEvent arg0) {

// **TODO** Auto-generated method stub

**int** nrQueue = Integer.*parseInt*( frame.getMaxNrOfQueues() );

**int** minProcTime = Integer.*parseInt*( frame.getMinServiceTime());

**int** maxProcTime = Integer.*parseInt*( frame.getMaxServiceTime());

**int** simulationInterval = Integer.*parseInt*(frame.getSimulationInterval());

**int** maxNrOfTasks=Integer.*parseInt*( frame.getMaxNrOfTasks() );

//frame.setVisible(false);

Simulator sim = **new** Simulator(frame,nrQueue, minProcTime, maxProcTime, simulationInterval,maxNrOfTasks);

Thread th = **new** Thread(sim);

th.start();

}

## 2.2 Scenarios

In order to perform the simulation correctly, and the user to be able to see exactly how it works, the values enter must be in some desired intervals.

## 2.3 Use Cases

Name: “ simulate ”

Brief Description

The systems waits for the user to enter:

-maximum number of servers;

-minimum time of service;

-maximum time of service;

-simulation time;

-maximum number of tasks per server; The simulation will start according to the entered settings, and after the simulation interval passes, the queues will work till all of them are empty, only afterwards the system can be put again in function.

Actors

The users which wants to simulate a queue based system, it can be an online store manager, or a person which works with scheduling some programs or some tasks, which are randomly presented, or a person which wants to analyse these kind of problems.

Preconditions

At each new run of the application, the text fields will be ready to enter new polynomials by displaying a zero on each text field.

A frame containing labels, buttons and text fields, a controller are already created and ready to be filled up with new data, entered by the user.

The flow:

The systems waits for the user to enter: maximum number of servers, minimum time of service, maximum time of service, simulation time, maximum number of tasks per server;

The simulation will start according to the entered settings, and after the simulation interval passes, the queues will work till all of them are empty, only afterwards the system can be put again in function.

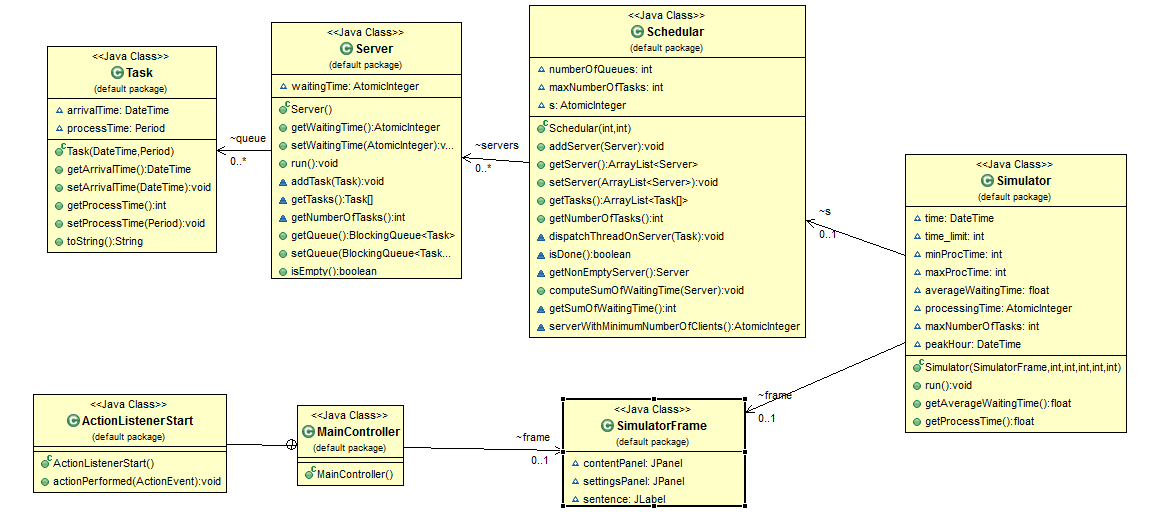
Post Conditions

After the use case is performed, the queues are empty, and the threads are stoped, a new simulation can be start.

## 3.Design (UML Diagram)

The UML Diagram was build using Eclipse IDE.

This is a more clear view of the diagram, without showing nesting and association multiplicity:



## 4.Implementation and testing

The project was implemented in Eclipse IDE, with the help of its features and some predefined classes and interfaces, the more important are : Collections,Array List, JodaTime package, Swing, Awt, ActionListener, BlockingQueue, threads system.

* The most important class was BlockingQueue, which came in handy when having to deal with a collection of tasks, with a variable and dynamic dimension. 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 (eithernull 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.

* The Swing and Awt classes provided the buttons, text field, layouts, labels to create a friendly user interface.
* The Joda Time package provided two important classes for this project, namely : Period and DateTime. The first one the one that helped with setting a period for the task to be performed. And the second one help with retrieving the real time and start the simulation from the current time to a given moment of time.
* The method actionPerfomed() form the interface ActionListener helped in making the bound between the view and the model.

## 5.Results

The results obtained can be viewed and tested in the user interface.

To verify if the application retrives well the statistics, one can perform the operations by hand on paper and check if they reseamble with the actual response of the application.

## 6. Conclusions

The assignment help me improve my knowledge regarding Object- Oriented- Programming. The most important part of the project was learning how to work with

threthrthreads I learned how to design from a project, how to transform some

threads, how to synchronized them and also how to perform a simulation. Another important thing that I learned are new structures like BlockingQueue, JList, Period, DateTime.

## 7.Future developments of the project

1. The project can be improved by adding more settings to the simulation.
2. Another will be to have a way to redistribute tasks when a new server is open.
3. To have a button to stop the simulation, or to pause it whenever you want.
4. To have predefined set of values for the simulation in case the user just wants to see the mechanism and not to set some predefined values.
5. To be able to close a server a given moment of time, and the clients of that server to be redistribute to the other active servers.
6. To improve the user interface, by making it more friendly, and more easy to use.

## 8.Biography

**Books**:

- Joshua Bloch, Effective Java (2nd Edition);

- [Kathy Sierra, Bert Bates](http://shop.oreilly.com/product/9780596009205.do#tab_04_2), Head First Java (2nd Edition), O'Reilly Media;

- Barry Burd, Java for Dummies (5th Edition), Wiley;

**Websites**:

- <http://stackoverflow.com/>

- <https://www.oracle.com/java/>

-http://www.oracle.com/technetwork/articles/javase/index-142890.html