*Technical University of Cluj-Napoca  
Faculty of Automation and Computers  
Department of Computer  
2nd Semester 2016*

*Programming Techniques*

*Homework 3*

*Student: Ghiurutan Alexandru   
 Group: 30425*

***Contents:***

1. *Problem Specification ................................................................ 3*
2. *Example of working .............................................................. 4*
3. *Design*

*3.1 Relational Diagram …............................................... 5*

*3.2 UML Diagrams …………………………………… 6*

*3.3 Sequence Diagram………………………………….. 7*

*3.4 Activity Diagram…………………………………... 8*

*3.5 Classes Design ............................................................ 9*

*3.5.1 Gui Class*

*3.5.2 Shop Class*

*3.5.3 Queue Class*

*3.5.4 Client Class*

*3.6 Packages and Interfaces............................................. 15*

*3.7**Runnable Interface …............................................... 16*

*3.8 User Interface........................................................... 17*

1. *Using and testing the application …………………………….. 19*
2. *Conclusions …………………..……………………………..... 19*

***1. Problem Specification***

TP – Homework 2

Objective:

Design and implement a simulation application aiming to analyse queuing based systems for determining and minimizing customers 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 "customer" to wait before receiving a "service". The management of queue based systems is interested in minimizing the time amount its "customers" 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 customers will be evenly distributed to all current available queues. The system should simulate a series of customers arriving for service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the customers 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 customers - when they show up and how much service they need.

The finish time depends on the number of queues, the number of other customers in the queue and the service needs of those other customers.

Input data:

Minimum and maximum interval of arriving time between customers;

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;

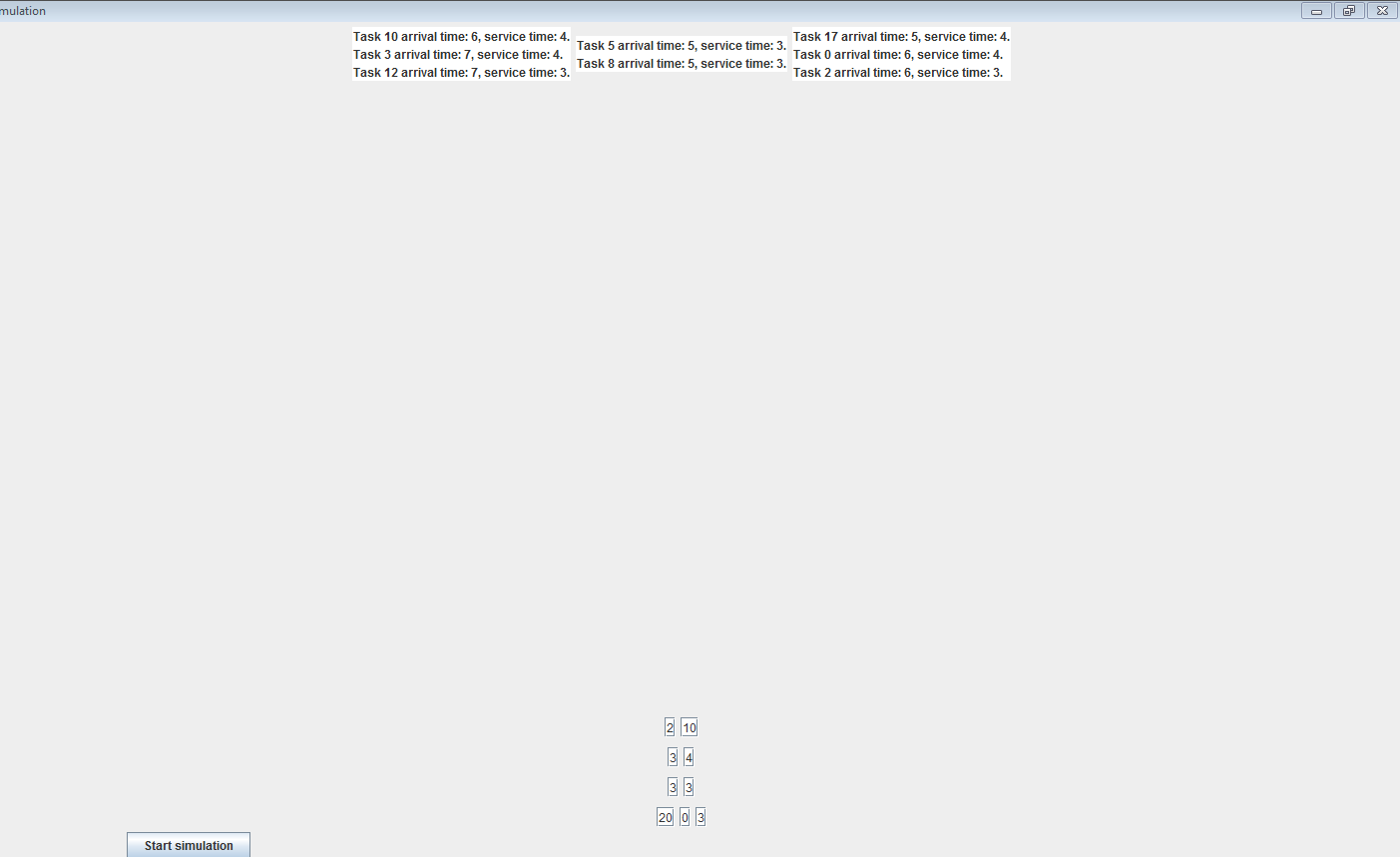
Log of events and main system data

Queue evolution

Peak hour for the simulation interval

***2. Example of working***

Pressing the Start button the application will display the queues. With the passage of time it will show different clients arriving and then waiting at one of the queues (the new client chooses a queue depending on how many clients are already waiting in that queue).



What is more, in the Console text area, there will be displayed all the actions that happened in that simulation time and at the end of the simulation time there will be printed the requested outputs like average waiting time.

***3. Design***

***3.1 Relational Diagram***

For resolving the problem specifications, I’ve chosen to use four different classes so that the design of the application would be as good as possible. The names of the classes and relations between them are represented in the figure below.

About their use and the reasons why I have chosen to organize the objects as it is written are explained in section 3.2: “Classes Design”

TaskGenerator

1

1

Gui

\*

Task

1

1

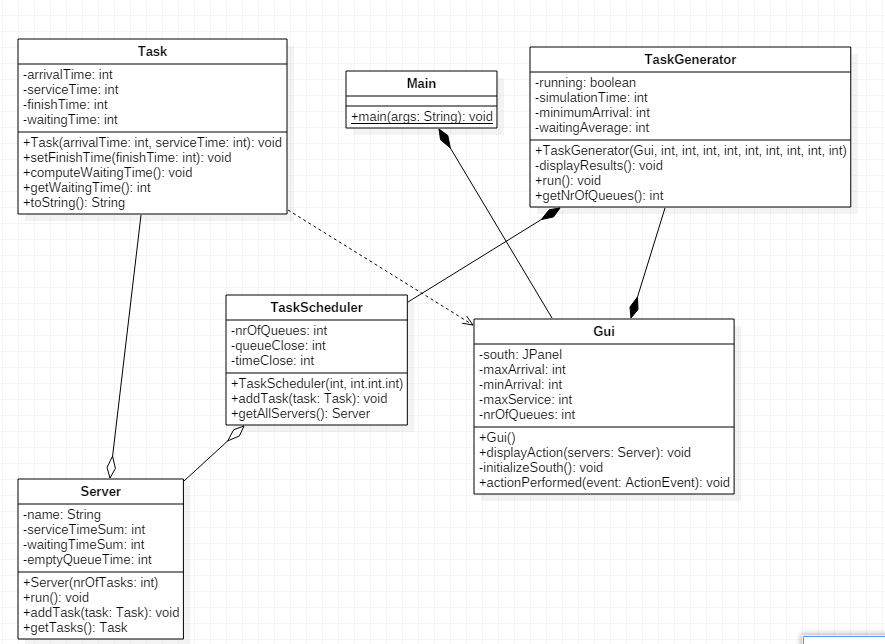
\*

TaskScheduler

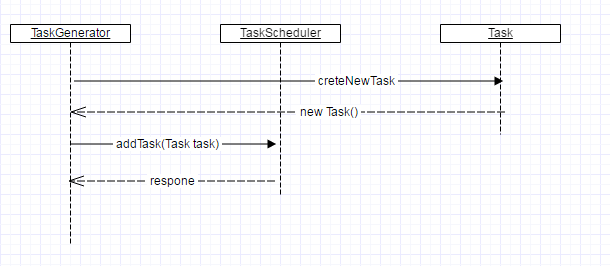
Relationship Diagram

To get a better view related to the attributes of each class, there are below the UML diagrams for each class. Thus, we can see every class with objects and their methods.

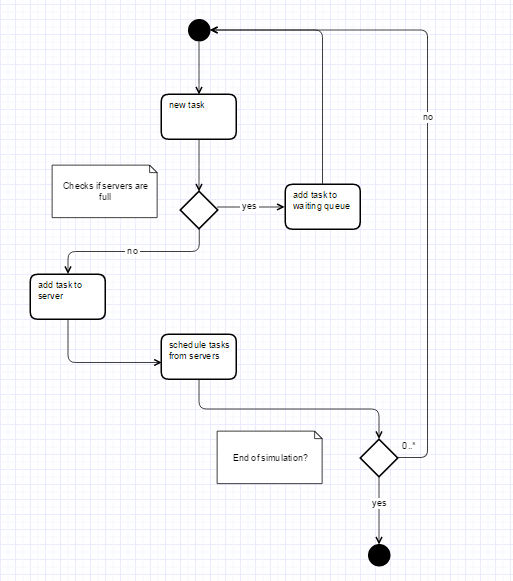
***3.2 UML Diagram***



***3.3 Sequence diagram***

******

***3.4 Activity diagram***

******

***3.5 Classes Design***

1. Gui Class : public class Gui extends JFrame

This class is designed to create a graphical user interface so the application would be easier to use. We consider the Gui class as a subclass of the predefined class "JFrame" so we can use objects of type "button", "frame" or "panel".

In this class there is declared the function public static void main (String [] args) which makes the whole application running. As a result of this run, on the screen it will open a window through which the user can enter data and perform operations that are possible with this application.

Observe the six TextFields for reading the necessary information at the top of the window. The user can insert any integer, each of them representing:

Number of existing queues in the shop

Service time interval: for each client, the program randomly generates a value in the specific range (contains the minimum and the maximum); this has the significance of time which takes for the customer to be served

Range of arrival: like the “service interval” this is randomly generated for each customer when he/she arrives at a queue.

Simulation time

There are two buttons: one to start the program after input data were introduced (Start) and the other for closing the application (Quit).

Below them there is a space for placing the graphical representation of the queues and the clients for this shop simulation.

The Attributes of the Gui Class:

To achieve the desired GUI we need several types of attributes:

The Constructor of the Gui Class: public Gui ()

It initializes all the attributes declared above; the window with the following elements will be constructed: frame, buttons, text fields. Here there are the frame settings, such as the size, the visibility, title and some predefined operation such as:

public Gui() {

this.setTitle("Simulation");

this.setLayout(new BorderLayout());

initializeSouth();

simulation = new JPanel();

simulation.setLayout(new FlowLayout());

this.add(simulation, BorderLayout.CENTER);

this.setExtendedState(JFrame.MAXIMIZED\_BOTH);

this.setLocationRelativeTo(null);

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

this.setVisible(true);

}

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

What is more, for every button (operation) there will be added an “ActionListener” which contains the instructions that are needed to be executed in the moment a certain operation button is clicked. For each button there is a different class declared inside the Gui class which implements the “ActionListener” interface.

More information about the “ActionListener” implementation are in the chapter about the user interface where there are presented all the swing components used in the Gui constructor.

The Methods of the Gui Class:

Gui contains methods for achieving operations when you press one of the two existing buttons in the interface. Moreover, it contains the main function which is run every time the application is opened.

main method:

This method aims to achieve the interface, this means that within this function we declare a new object's constructor called Gui to achieve early window:

Methods for the two buttons:

* + The Start button:

It declares a class which implements the predefined class called ActionListener. When the Start button is pressed by the user, it generates the simulation of the store by drawing a graphical representation for queues and clients. This is done by constructing a new object of type Shop. Parameters like number of queues or time intervals from the frame are transmitted to the Shop class constructor.

If the user wants to run the application for other inputs, he needs to enter them in the text fields and press again the Start button. To get the information from the text field we use a parsing function like this:

Integer.parseInt (numar.getText ());

1. TaskGenerator Class: public class TaskGenerator implements ActionListener

This class is the graphical representation of the "store" to consider. This means that under the information panel it will draw the specified number of queues, but also the clients waiting in line.

The Attributes of the TaskGenerator Class:

The information entered by the user in the text fields:

private static int currentTime = 0;

private int minimumArrival, maximumArrival, minimumServiceTime, maximumServiceTime;

private int simulationTime;

private int numberOfQueues, nrOfTasksPerQueue;

private double waitingAverage, serviceAverage, emptyQueueTimeAverage;

private TaskScheduler taskScheduler;

private Gui gui;

private Server[] servers;

private BlockingQueue<Task> tasks;

private DecimalFormat decimalFormat;

private volatile boolean running;

The Constructor of the TaskGenerator Class:

The constructor has the role to initialize the drawing area by declaring the images representing clients and queues and setting the background of this area.

this.gui = gui;

this.minimumArrival = minimumArrival;

this.maximumArrival = maximumArrival;

this.minimumServiceTime = minimumServiceTime;

this.maximumServiceTime = maximumServiceTime;

this.simulationTime = simulationTime;

this.setNumberOfQueues(numberOfQueues);

this.setNrOfTasksPerQueue(nrOfTasksPerQueue);

decimalFormat = new DecimalFormat();

taskScheduler = new TaskScheduler(numberOfQueues, nrOfTasksPerQueue,queueClose,time);

The methods of the TaskGenerator Class:

private int getRandomWithinRange(int min, int max)

private void displayResults()

private void setAllTasks()

private void scheduleTasksPerTime()

public void run()

public int getNrOfTasksPerQueue()

public static int getCurrentTime()

public void setNrOfTasksPerQueue(int nrOfTasksPerQueue)

public int getNumberOfQueues()

public void setNumberOfQueues(int numberOfQueues)

1. TaskScheduler Class: public class TaskScheduler implements Runnable

It represents a queue as a list of several client objects. Moreover, we have variables for each queue, like the time to empty the queue, so we can be able to determine the average drain of a queue.

The Attributes of the TaskScheduler Class:

Typical attributes for a queue: list of customers waiting in the queue ( ArrayBlockingQueue < Task > waitingQueue ) , the number of the tasks who are already waiting in that queue .

private int nrOfQueues ;

private int nrOfTasksPerQueue , queueClose , timeClose , nrOfTasks , peakHour ;

private List < Server > servers ;

private ArrayBlockingQueue < Task > waitingQueue ;

private Iterator < Server > iterator ;

The Constructor of the TaskScheduler Class:

Initializes client list and queue number (the class constructor is called Board)

The methods of the TaskScheduler Class:

The methods : private void distributeServers(Task task) {

verifyTimeToClose();

boolean fullServers = areServersFull();

if (servers.isEmpty() || ((servers.size() < nrOfQueues) && (fullServers))) {

Server server = new Server(nrOfTasksPerQueue);

servers.add(server);

server.addTask(task);

new Thread(server).start();

} else if (fullServers) {

waitingQueue.add(task);

} else {

Collections.sort(servers, new Comparator<Server>() {

@Override

public int compare(Server o1, Server o2) {

if (o1.getNrOfTasks() < o2.getNrOfTasks()) {

return -1;

} else if (o1.getNrOfTasks() > o2.getNrOfTasks()) {

return 1;

}

return 0;

}

});

if (waitingQueue.isEmpty()) {

servers.get(0).addTask(task);

} else {

servers.get(0).addTask(waitingQueue.remove(0));

}

}

displayServers(servers);

}

public void addTask(Task task) {

distributeServers(task);

}

It creates a new task calling the constructor of the Task class and then the task is added to the tasks list with the instruction:

waitingQueue.add ( task ) ;

* + It one changes the variables peekHour, timeClose , queueClose and nrOfTasks so they can be used in the final information display, after simulating queues

The method:

* + Comes from the implementation of the Runnable interface and is supposed to execute some instructions for a period of time specified by Threads.
  + In other words, this function is executed after a certain time period: a client does not move from the queue only after his waiting time has passed.

1. Task Class: public class Task

The Attributes of the Task Class:

Each customer has a name and a time that is forced to wait in line.

This time is calculated when a customer enter the queue and waiting time depends on the other clients queue

private String name;

private static int number = 0;

private int arrivalTime;

private int serviceTime;

private int finishTime;

private int waitingTime;

private HelperTask helper;

The Constructor of the Task Class:

Initialize a customer's attributes (is called from the Queue class when adding a new client to the list)

public Task(int arrivalTime, int serviceTime) {

this.arrivalTime = arrivalTime;

this.serviceTime = serviceTime;

this.name = "Task " + number++;

helper = new HelperTask();

}

The methods of the Task Class:

public int getArrivalTime() {

return arrivalTime;

}

public void setFinishTime(int finishTime) {

this.finishTime = finishTime;

}

public void computeWaitingTime() {

this.waitingTime = helper.getWaitingTime(arrivalTime, finishTime, serviceTime);

}

public int getWaitingTime() {

return waitingTime;

}

public int getServiceTime() {

return serviceTime;

}

@Override

public String toString() {

return String.format("%s arrival time: %d, service time: %d.\n", name, arrivalTime, serviceTime);

}

}

**3.6 *Packages and Interfaces***

A Java package is a mechanism for organizing Java [classes](http://en.wikipedia.org/wiki/Class_%28computer_science%29) into [namespaces](http://en.wikipedia.org/wiki/Namespace_%28computer_science%29). Java packages can be stored in compressed files called [JAR files](http://en.wikipedia.org/wiki/JAR_file), allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality. A package provides a unique namespace for the types it contains. Classes in the same package can access each other's package-access members.

A package allows a developer to group classes (and interfaces) together. These classes will all be related in some way – they might all have to do with a specific application or perform a specific set of tasks.

For this application the following packages are imported, each of them having a certain role for the proper working of the application. We import them in the Gui Class (most of them relate to the user interface properties):

* import java.awt: Contains all of the classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components.
  + java.awt.BorderLayout: A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center.
  + java.awt.Color: The Color class is used encapsulate colors in the default RGB color space or colors in arbitrary color spaces identified by a [ColorSpace](http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/color/ColorSpace.html).
  + java.awt.Dimension: This encapsulates the width and height of a component (in integer precision) in a single object.
  + java.awt.GridLayout: The GridLayout class is a layout manager that lays out a container's components in a rectangular grid for a better view of all the buttons and textfields which are added to the main panel.
* import java.awt.event
  + java.awt.event.ActionEvent;
  + java.awt.event.ActionListener;
* import javax.swing: Typical Swing applications do processing in response to an event generated from a user gesture. For example, clicking on a JButton notifies all ActionListeners added to the JButton. That’s why we use this package for creating the user interface Gui.
  + javax.swing.JButton;
  + javax.swing.JFrame; javax.swing.JLabel; javax.swing.JPanel;

***3.7 Runnable Interface***

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 behaviour of the class.

In this program for each queues from the array of queues declared in the Shop class, we declare a thread. This means that we will have a run method in the class Queue which will determin whether or not to execute an action depending on the time. In this case the queue will remain unchanged while a client is being served and only after this time has passed the client is removed from the queue. This is done with the following instructions:

try {

Thread.sleep(time);

}

catch (InterruptedException e) {}

clienti.remove();

***3.8 User Interface***

When running the application, the window will open and it will provide to the user the possibility of giving inputs and choosing what operation he likes to be executed. This window is constructed in the Gui class using some predefined classes and instructions.

The user interface is based on the properties of the above mentioned packages. All the objects we need are declared as attributes of the Gui class and they are initialized in the constructor of this class. For executing the operation commanded by the user we use the predefined functions from the ”ActionListener” interface.

The ActionListener functions

The listener interface is for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked. In this case the only events that occur are when the user clicks on one of the operation buttons from the graphical interface.

We take as an example the instructions that need to be executed when clicking on the “P(val)” button, which leads to the determination and displaying the value of the polynomial written in ”First Polynomial” field in point val. For this we need another class which implements the ActionListener:

private class StartListener implements ActionListener

This class will contain the method which executes all the instruction needed in order to fulfill the selected operation.

public void actionPerformed (ActionEvent e)

***4. Using and testing the application***

In order to use the application open Homework1.JAR. This will open a window which generates the Gui class. Thus the user can enter the desired values and selecting the operations by pressing one of the 9 buttons.

***5.Conclusions***

Achieving such a program may be hard both in terms of algorithms, graphical structure.

For a better performance there should be implemented all cases where exceptions can occur and the application stops working due to an error made ​​by the user. Also, the division method should be done so it can calculate the reminder as well. Another thing that could be improved is the display so that it would be more elegant.

***6.References***

[*http://users.utcluj.ro/~jim/OOPE/*](http://users.utcluj.ro/~jim/OOPE/)

[*http://docs.oracle.com/javase/7/docs/api/overview-summary.html*](http://docs.oracle.com/javase/7/docs/api/overview-summary.html)

[*http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html*](http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html)

[*http://stackoverflow.com/*](http://stackoverflow.com/)