*Technical University of Cluj-Napoca*

*Faculty of Automation and Computers*

*Department of Computers*

*2nd Year, 2nd Semester 2015-2016*

Programming Techniques

Homework 1

Student: Robert Văcăreanu

Group: 30424

CONTENTS

1. Project specification
2. Problem analysis, model, testing and usage
3. Design
4. Implementation and testing
5. Results
6. Conclusions
7. Bibliography
8. Project Specification

**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;

The problem specification gives us hints about the feel of the application, but it also gives us flexibility in terms of: data structures used, design patterns, structure, functionality. The approach we’ve took in designing the application is described in the following paragraph.

We followed the requirements and divide the whole problem into 3 modules ( packages ):  
**M**odel – **V**iew – **C**ontroller.



It is clearly that every action is done through the controller. This is an architectural design pattern, a triangular one, to be more concise. It should split the code into three re-usable modules.

1. Problem Analysis, Model, Testing and Usage
2. Problem Analysis

In order to fulfill the requirements, we had to analyze the problem. After splitting the whole application into three modules, we have to further design the classes in each of them. Since this problem is more or less a produces – consumer type of problem , we had to make the synchronization. This could be done either by manually ensure it using wait, yield e. t. c. or by using what JDK provides to us: blocking queues. More specifically, in this application was used an linked blocking queue. A **blocking queue** is a **queue** that blocks when you try to de queue from it and the **queue** is empty, or if you try to en queue items to it and the **queue** is already full. A thread trying to de queue from an empty **queue** is **blocked** until some other thread inserts an item into the **queue**. This solved, partially our synchronization problems.

1. Model

The package model of this application consists of following classes:

1. ParameterObserver
2. Server
3. Task

**Class ParameterObserver**

This class extends the class Observable provided in JDK. This is used to notify the controller if the user changed the parameters of the application: minimum serving time, maximum serving time, maximum number of queues and time limit. Also, it makes the methods: setChanged and clearChanged from protected ( in the super class ) into public.

**Class Server**

This class is the actual queue. It contains a blockingqueue (as specified above) of tasks ( using generics ) and an atomic integer that represents the waiting time. The atomic integer was used to ensure the operations to be atomic, hence to not worry about synchronization safety. Also, this class implements Runnable, hence it is used when creating the queues threads. Among its methods, there are:

* **addTask:** adds a task into the queue
* **getBq:** return an array of tasks
* **getTotalTime:** returns the total waiting time
* **getLastTask:** it is used when adding a new queue to redistribute some of customers. Move the last from the queue into the new queue
* **getId:** returns the id of this queue

**Class Task**

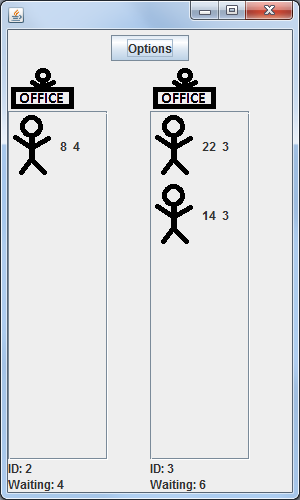
This class is the representation of a customer. It have two fields: arrivalTime and processTime. It has only one constructor that takes as arguments two integers. Therefore, at creation time you must provide those parameters. It contains getters and setters for those two fields. The method toString is overrode in order to provide meaningful details on the screen at the queue

1. Testing

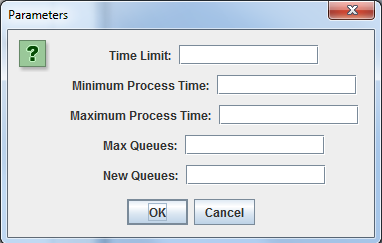
The correct functionality of the application was proven by multiple tests that covers most of the situation a consumer will encounter in mostly any situations.

1. Usage

The application comes with a fairly simple to use GUI. There are no overwhelming buttons, settings or other things like that. Everything is kept as simple and minimalistic as possible while providing in handy places all the functionality needed.



At start the users have two options: either change the running parameters or let them default. The default parameters were chosen so that it models the real life situations in the best possible way. However, if the end user feels like modifying those parameters for testing purposes they are allowed to do so.



1. Design

As mentioned above, we will split the implementation into four packages, following a **M**odel – **V**iew – **C**ontroller type of architecture. This favors the re-usability of the modules and keeps everything well organized

The whole application is split into three big packages, along with the default one that is used only to keep the main method in it. However, this can be easily changed since that package doesn’t provide any functionality at all.

The **package model**:

Contains the following classes: ParameterObsever, Server, Task.

**Class ParameterObserver**

This class extends the class Observable provided in JDK. This is used to notify the controller if the user changed the parameters of the application: minimum serving time, maximum serving time, maximum number of queues and time limit. Also, it makes the methods: setChanged and clearChanged from protected ( in the super class ) into public.

**Class Server**

This class is the actual queue. It contains a blockingqueue (as specified above) of tasks ( using generics ) and an atomic integer that represents the waiting time. The atomic integer was used to ensure the operations to be atomic, hence to not worry about synchronization safety. Also, this class implements Runnable, hence it is used when creating the queues threads. Among its methods, there are:

* **addTask:** adds a task into the queue
* **getBq:** return an array of tasks
* **getTotalTime:** returns the total waiting time
* **getLastTask:** it is used when adding a new queue to redistribute some of customers. Move the last from the queue into the new queue
* **getId:** returns the id of this queue

**Class Task**

This class is the representation of a customer. It have two fields: arrivalTime and processTime. It has only one constructor that takes as arguments two integers. Therefore, at creation time you must provide those parameters. It contains getters and setters for those two fields. The method toString is overrode in order to provide meaningful details on the screen at the queue

The **package controller**:

Contains the following classes: Scheduler, Simulator

**Class Simulator**

This class is responsible with the scheduler and the simulator frame classes. It implements the observer interface provided in JDK since java 1.0 because this class needs to be notified if the end user changes the parameters to further notify the scheduler class in order to update all the classes involved. It also connects the view to the model through the scheduler class. Since we are following the model view controller architectural design pattern, the view is not allowed to directly access the model. Hence this is done in the controller class by using the scheduler class. This class also implements the runnable interface, hence the code inside is running on a new thread. This is done to further improve the performance of the application and to provide the end user with a very smooth application. It contains two additional methods, besides the one that are implied to be there by the use of interfaces. Those two methods are: reset and createTask.

The reset method is used to set those fields used for final feedback, after the specified time have passed.

**Class Scheduler**

This class is responsible with the queues/servers. Therefore it contains a list, linked list to be precise, of servers. It also contains a comparator which is an anonymous inner class used for sorting the list when adding a new task. This is done because we want to add a new task to the server with least waiting time, hence be as effective as possible. It contains a method called totalTasks that returns an integer that represents the total number of tasks that are at that particular moment in the servers. Needed when we decide if we create another server or not. This is because we have an upper limit of servers, hence we cannot create them every time is wanted or needed. The second method is getMaxTime that is used to get the maximum waiting time among all the available servers. This method is needed in the end to give the user the final feedback for its simulation. The third method is startNewServer that is used when we decide to create a new server. It also contains calls to addition methods that are private, hence are only utility methods. The fourth method, redistribution, is needed when creating a new server to redistribute evenly the tasks among servers. This makes the creation of a new server valuable in the sense that after creation, it is populated first with tasks from other servers and then with new tasks. Hence, a new server tries to even out all the servers. The fifth method is newServer where all the logic of a new server creation is put together. The sixth and last method is reset method used to re-simulate everything after the end user changed the parameters. This class is aggregated by the Simulator class. No other class have direct access to it. They are forced to go through simulator class.

To sum up, this class is responsible with putting the task to the appropriate server as well as creating new server based on some random values and the limit set by the user or by the default parameter set at run time.

The **package view**:

This package contains the following classes: Image Cell Renderer, Queue Representation, Simulator Frame.

**Class ImageCellRenderer**

This class extends the class DefaultListCellRenderer in order to provide custom row in j list. Oracle documentation recommandations:

public class **DefaultListCellRenderer**

extends [JLabel](https://docs.oracle.com/javase/7/docs/api/javax/swing/JLabel.html)

implements [ListCellRenderer](https://docs.oracle.com/javase/7/docs/api/javax/swing/ListCellRenderer.html)<[Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)>, [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

Renders an item in a list.

**Implementation Note:** This class overrides invalidate, validate, revalidate, repaint, isOpaque, and firePropertyChange solely to improve performance. If not overridden, these frequently called methods would execute code paths that are unnecessary for the default list cell renderer. If you write your own renderer, take care to weigh the benefits and drawbacks of overriding these methods.

**Warning:** Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications running the same version of Swing. As of 1.4, support for long term storage of all JavaBeansTM has been added to the java.beans package.

It only overrides two methods, namely get list cell renderer component and get icon/ The first method is overridden because we need to attach to the label inside the list a representative image, more precise, we put the customer image.

This class is used inside the queue representation class to represent a customer

**Class QueueRepresentation**

This class is used to represent a queue graphically. It takes advantage of the class image cell renderer in order to provide the end user with a graphical representation of the tasks as the customers. It also adds a label that represents the office in order to make everything as realistic as possible.

Its only responsibility is to display the look and the feel of a real queue, without adding any functionality to it

**Class SimulatorFrame**

This class aggregate the clas queue representation in order to display graphically a queue. This class extends JFrame since it will be the main window of the application. It contains 6 fields, namely:

CUSTOMER: a public static final String with the path of an image in order to display the customer image inside the queue. It is a constant

OFFICE: a public static final String with the path of an image in order to display the office image inside a queue. It is a constant

Options: a private j panel that holds only a button

Panel: a private j panel that holds all the components beside the option panel.

ParameterObserver: used in order to notify the observers if the user changed the data

Int[] : used to get the data from the user and to further send it to thecontroller

As a layout manager, we used:

For JFrame:

Border Layout : A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center. Each region may contain no more than one component, and is identified by a corresponding constant: NORTH, SOUTH, EAST, WEST, and CENTER. When adding a component to a container with a border layout, use one of these five constants.

For panel:

Grid Layout with only one row: The GridLayout class is a layout manager that lays out a container's components in a rectangular grid. The container is divided into equal-sized rectangles, and one component is placed in each rectangle. It is important to note that, upon creation, when you specify the rows and columns to the grid layout, it will not really respect them. This means that it will check if the specified number of rows is greater than 0. If it is, this means that the columns will be variable in order to be able to add all the desired components. If the specified number of rows is 0, this means that the number of rows will be variable while respecting the desired number of rows. In simpler terms, this means that you cannot force the grid layout to a specified number of rows and columns but to only one.

Among the methods that it contains, the most notable ones are:

Display data that takes as arguments: an array of tasks, an integer that represents the waiting time and an integer that represents the id of the queue. In this method we create a new queue representation with the given arguments and add it to the panel. Then we revalidate and repaint it in order to make the changes visible.

This method is overloaded such that it can take a list of server as arguments. This is because, when we call it we will actually use this method and do the needed logic to call the above method inside it.

Display error: this method takes a string as argument. This string contains the data that will be displayed on a j option pane when an error will occur.

Get results : this method is called when the user press the button options. This button has an action listener implemented as an anonymous inner class that calls this method. This method contains all the logic wrapped inside a panel in order to display it to the end user inside an option pane. Then, the user enters the data into text fields and press a button, either OK or CANCEL. If the user pressed the ok button, we further parse the data and notify the observers. We do this in order to update the controllers with the new data inserted by the end user. However, if the user press the cancel button, the whole application will stop since it is considered that the user no longer wants to simulate queues.

Add observer: this method is used in order to add a new observer inside the aggregated class: parameter observer. This method is called by any object that want to be notified when the user enters new parameters.

1. Conclusions

While the difficulty depends on the model and the approach, it also depends on how many features the developer wants to add into its application. However, I evaluate the difficulty of this project somewhere between easy and medium. This is because almost every synchronization problem that a developer shall encounter while working on this small project is solved in jdk. This is because it provides:

Executor: An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads.

Blocking Queues: 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.

Atomic primitives: In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

Hence, lots of possible problems encounter by a developer on this task are partially or totally solved by Java.

By implementing this project, I got a brief look into the world of synchronizations and threads.

[http://grepcode.com/static/app/images/1x1.gif](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/awt/GridLayout.java)