**Simulating and analyzing**

**the behavior of a queue**

Homework 2

Technical University of Cluj-Napoca

Student: Natalia Carina Sîrbu

Computer Science, 2nd year, group 30425

March 13, 2017

Table of Contents

[1. Introduction to queues 3](#_Toc479347369)

[2. Objectives 3](#_Toc479347370)

[2.1 Main objective 3](#_Toc479347371)

[2.2 Problem analysis and approach 3](#_Toc479347372)

[3. Scenarios 4](#_Toc479347373)

[4. Design 4](#_Toc479347374)

[4.1. Class Diagram 4](#_Toc479347375)

[4.2. Data structures 5](#_Toc479347376)

[4.3. Class details 6](#_Toc479347377)

[4.3.1. Task 6](#_Toc479347378)

[4.3.2. Server 6](#_Toc479347379)

[4.3.3. SimulationManager 7](#_Toc479347380)

[4.3.4. View 7](#_Toc479347381)

[4.3.5. Controller 8](#_Toc479347382)

[4.3.6. Animator 8](#_Toc479347383)

[4.3.7. Logger 8](#_Toc479347384)

[5. Lessons learned 9](#_Toc479347385)

[6. Bibliography 9](#_Toc479347386)

# **Introduction to queues**

Queues are commonly used to model real world domains. The principle of a queue is to provide a place for a customer/task/item to wait their turn until the service is delivered or they are to cared of. The main purpose of a queue based system is to provide as little waiting time as possible for its “clients”. The solution that is most wide used is to open a new server, i.e. to open new queues in the system. However, this could increase considerably the costs of the service suppliers.

For the simplicity of this, we should refer to the “client” of a queue as “task” and to the queue as a “server”.

A server works by the principle “first in, first out” (known in Computer Science by the abbreviation “FIFO”). This means that the first task to be entered in a server is also the first one that exits the server.

# **Objectives**

## Main objective

The purpose of this project is to design and implement a simulation application aiming to analyze queuing based systems for determining and minimizing clients’ waiting time.

The application should follow the pattern: there is a list of tasks that arrive for service. Each task will enter a server, will have a processing time (meaning the time needed by the server to process the task) and then will finally leave the queue.

The graphical user interface should display the real-time queue evolution and also the inputs needed for the application will be provided from the user: the simulation period of time in which the tasks are processed, the maximum time and the minimal time of processing for every task in particular, the number of servers that are available and the number of task the simulation starts with. It is known that not every user will be necessarily a programmer or a person with basic knowledge in programming. Therefore, the interface should be clear and user friendly, displaying different messages that can guide the user.

## Problem analysis and approach

The way the project was implemented using the bottom-up approach. This means that the first thing to be created is the base of the problem that is also presented using a great amount of details. By doing so, we will end up connecting these items together in order to obtain the whole project. The problem was split in two packages: one related to the graphical user interface – gui – and one concerning the queues – model -.

One good starting point is to pick different nouns that are related to the subject and they will be the name of the classes. Moreover, in each class, the methods will bear the name of different verbs that are also related to the subject.

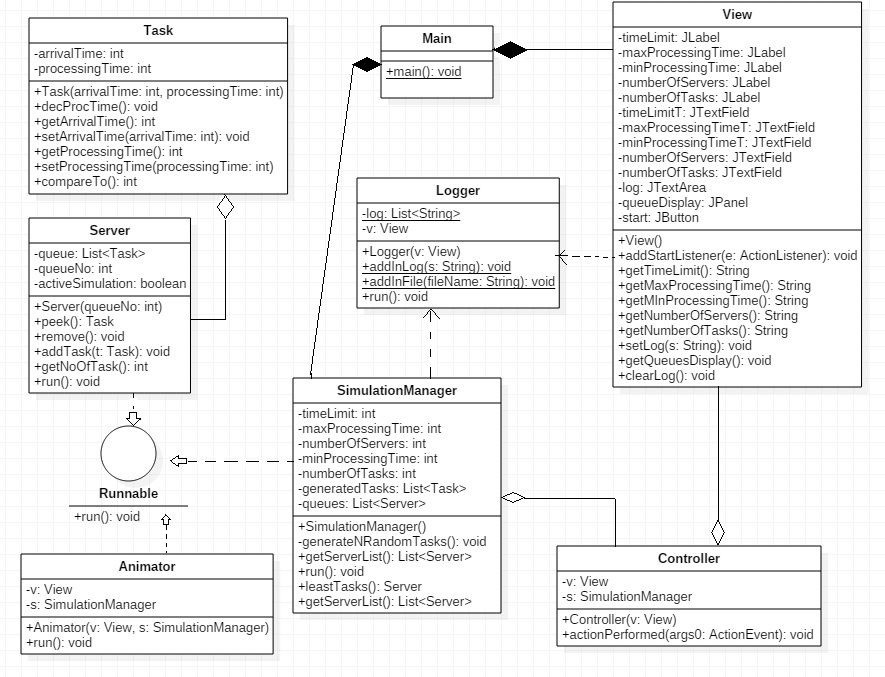
The first thing to do is to generate the task to be processed by the queues. They will be generated randomly using a method especially created for that particular purpose. Each queue/server will work on different thread meaning the project will be a multithreading one. The real-time processes will be displayed in a log window where all the new entries and removals are displayed.

# **Scenarios**

The scenario for this project is plain simple: the user is requested to insert the simulation time, the maximum time and the minimum time of the range in which the task will have their processing time, the number of servers that will deal with the tasks and the number of tasks. The “START” button is then pressed and the user will see the evolution of the task processing.

Possible faults of the algorithm will be that the user can introduce a simulation time that is too short and a great number of tasks to be processed and therefore the tasks. This could be an improvement for the future.

# **Design**



## 4.1. Class Diagram

This class diagram was designed using UML, i.e. Unified Modeling Language.

All the classes from this project are displayed in UML diagram above and a short description of their purpose will be presented in following moment:

1. The **Task** class creates the basic of this project and that is “the task”. Each task you be given an arrival time (arrivalTime) and a processing time (processingTime).
2. The **Server** class will refer to the queue itself and it will create a list of tasks to be processed. The methods in this class will act upon the relationship between each queue and its tasks. This class uses a thread. Therefore, it implements the interface **Runnable.**
3. The **SimulationManager** class will set the time limit, minimum time for the processing, maximum time for the processing, number of servers and number of tasks. Here will also be the random generator for the task that will make sure that the task will have a random time arrival and processing time. This class also uses a thread and it implements the interface **Runnable**.
4. The **Main** class, also known as the MVC – model-view-controller -, has only one purpose: to connect the model, the view and the controller.
5. The **View** class is a part of the graphical user interface where all the visual elements are developed. The buttons, text fields ad labels are declared here. There are some methods that require the input from the user and that is why here are methods that add action listeners to the required areas.
6. The **Controller** class will control and customize the action performed by each button, text field, text area or panel from the user interface.
7. The **Animator** class will deal with visually displaying the tasks in the JPane.
8. The **Logger** class deals with displaying the task added to each queue and the tasks that were removed in the JTextArea. The displaying will be performed from a file in which was previously written.

## 4.2. Data structures

Two types of data structures were used in this project and are presented below:

* *List<>* data structure was very useful in classes as **Server** (where a list of tasks was declared to store all the tasks from one queue), **SimulationManager** (where a list of tasks was generated and also a list of queues representing all the servers from the application) and in the **Logger** class (where it was needed in order to display the real-time evolution).

## 4.3. Class details

### 4.3.1. Task

This class has two private attributes that will be used to construct the object of this type. One attribute is the *arrivalTime* of the task which represents the time when the task arrived at the queue. The other one is the *processingTime* of the task which represents the time it takes to the server to finish processing the task. For each attribute, this class also contains its getter and setter.

* *public Task (int arrivalTime, int processingTime)* – this is the constructor of the class that has two parameters: one parameter is the *arrivalTime* of the task which represents the time when the task arrived at the queue and the other one is the *processingTime* of the task which represents the time it takes to the server to finish processing the task;
* *public void decProcTime ()* – this is a void method that will only decrement the processing time of the task that uses the method;
* *public String toString ()* – this method will return the string associated with the value of the number object.

### 4.3.2. Server

This class implements one interface: *Runnable*. This interface is used because a thread is used in this class and its behavior is presented in the method *run ()* which is the interface’s method that is going to be overridden. Two private attributes are declared at the beginning of the class: *queue* which is a list of tasks and *queueNo* which represents the number of a certain queue/server. The attribute queue is also declared as being **volatile** because it will be used by more threads and this indicates that its value will be modified by more threads. For each attribute, this class also contains its getter and setter.

* *public Server (int queueNo)* – is the class constructor that has only one parameter and it shows the number of the queue;
* *public Task peek () ­*– this method has only one instruction and that is to return the head of the list. This is done using the methods available by the “list” data structure. The return will be of type *Task*;
* *public void remove ()* – this method has a suggestive name and it will always remove from the *queue* the first element when called;
* *public void addTask (Task t)* – the beauty of object oriented programming is the we can use nouns and verbs to name our methods. That’s why this method is very intuitive. It will add in list of tasks *queue* the new task given as parameter.
* *public int getNumberOfTasks ()* – the purpose of this one line of code from the method is to return the number of tasks from the specific queue with the number *queueNo.*
* *public void run () ­*– inside this method, the behavior of the thread will be described. As long as the simulation is still running or the queue is not empty, the code will execute. The processing time of the first task from the queue will be decremented and when it will be equal to 0, the task will be removed using the methods from the **Task** class;
* *public List<Task> getQueue ()* – this will return a list of tasks.

### 4.3.3. SimulationManager

This class also implements the *Runnable* interface. This interface is used because a thread is used in this class and its behavior is presented in the method *run ()* which is the interface’s method that is going to be overridden. There are six private attributes declared in this class: the *timeLimit* of the simulation which represents the time the simulation will run, the maximum processing time of a task (*maxProcessingTime),* the minimum processing time listed for a task to be taken care of (minProcessingTime), the number of servers that will deal with the tasks (*numberOfTasks*), the number of tasks that will be shared to the queues (*numberOfTasks*), the list of tasks that will be randomly generated will be stored in a list of task called: *generatedTasks,* and another volatile variable *queues* which is a list of servers and it will be used by more threads and volatile indicates that its value will possibly be modified by more threads. In this class were implemented only the necessary getters and setters for the private attributes.

* *public SimulationManager (int timeLimit, int minProcessingTime, int maxProcessingTime, int numberOfServers, int numberOfTasks)* – this is the class constructor that has five parameters. The parameters were described earlier;
* *public void generateNRandomTasks () ­*­– this method will generate the tasks for the servers using the Random constructor. Each task should have a random time of arrival and a random processing time. After the generation, the tasks from the *queues* will be sorted in an increased order by their time of arrival;
* *public List<Server> getServerList ()* – this method only returns the list of servers denoted by *queues*;
* *public Server leastTasks ()* – this is a “find a minimum” method that will return the servers with the least tasks. This method is needed in order to place the new tasks in the server with smaller number of tasks;
* *public run ()* – this the run method from the Runnable interface. The code begins by starting every thread, i.e. each queue/server has a thread. The idea is that as long as the simulation time is greater than zero, the while loop will execute, but it will execute if the queue is not empty and if the simulation time is the same with the arrival time of the task. A message will be displayed in the logger saying that a new task was added. It will be added to the queue with the least tasks and then removed from the list of generated tasks. The thread will sleep for one second and then do this all again.

### 4.3.4. View

The **View** class includes the declarations of the attributes needed to build the graphical user interface. This graphical user interface requests:

* 5 labels that display a text for the user to be guided in the application: “Simulation time”, “Minimum processing time”, “Maximum processing time”, “Number of servers” and “Number of tasks”;
* 5 text fields: one for each label, in which the user may enter the values needed for the simulation;
* 1 button that will start the simulation after the data the introduced;
* 1 panel in which the animations for the queue will be displayed;
* 1 text area in which the user is not allowed to write and where the real-time events of the simulation will be displayed. This attribute was declared volatile because it will be used by more threads and this indicates that its value will be modified by more threads.

In the constructor of the class, the bounds for each attribute is set and then added to the frame that will be displayed on the screen.

### 4.3.5. Controller

The **Controller** class will take care of the mouse events. This will occur when the user opens the application and after he or she inputs the simulation time, the maximum processing time, the minimum processing time, the number of servers and the number of tasks. Once the “START” button is pressed, the class will take care of the followings.

An instance of the class **View** is created: *v*. Moreover, the class constructor has a parameter *view* of type *View*.

There is another class inside this class that is declared as private and that will take care of the events after pressing the button. An instance of the **SimulationManager** class will receive as parameters the methods created in the **View** class that will get the inputs from the user and used them. Three threads will be instanced here that will refer to the **Animator** class and **Logger** class.

### 4.3.6. Animator

The **Animator** class will also implement the *Runnable* interface. This interface is used because a thread is used in this class and its behavior is presented in the method *run ()* which is the interface’s method that is going to be overridden. There are two private instances that are declared in the beginning: an instance of the **View** class: *v* and an instance of the **SimulationManager** class: *s.*

* *public Animator (View v, SimulationManager s)* – this is the constructor of the class that has two parameters;
* *public void run ()* – there is an infinite loop at the beginning and it starts by removing everything that was written there before. It works like a repaint and it repaints it also updates the panel. The JPanel will display a black square for every task that will be added in the queue. After that, the thread will go to sleep for 999 milliseconds because in case one task is removed and another one is added at the same simulation time, there will not be a displaying error.

### 4.3.7. Logger

The **Logger** class will use the JTextArea which is an area for text that displays the real-time simulation messages. This class has two private parameters: *log* is declared as a list of strings and will be printed in the text area for showing the real-time evolution of the application; *v* is an instance of the **View** class.

* *public Logger (View v)* – this is the constructor of the class that has only one parameter which is *v*;
* *public static void addInFile (String s)* – this is just a method that adds a new string in the list of strings declared as private attribute at the beginning of the class.
* *public void run ()* - – there is an infinite loop at the beginning and it starts by removing everything that was written there before. It works like a repaint and it repaints it also updates the text area. After that, the thread will go to sleep for 999 milliseconds because in case one task is removed and another one is added at the same simulation time, there will not be a displaying error;
* *public List<Server> getServerList ()* – this method only returns the list of servers denoted by *queues*;
* *public Server leastTasks ()* – this is a “find a minimum” method that will return the servers with the least tasks. This method is needed in order to place the new tasks in the server with smaller number of tasks;
* *public run ()* – this the run method from the Runnable interface. The code begins by starting every thread, i.e. each queue/server has a thread. The idea is that as long as the simulation time is greater than zero, the while loop will execute, but it will execute if the queue is not empty and if the simulation time is the same with the arrival time of the task. A message will be displayed in the logger saying that a new task was added. It will be added to the queue with the least tasks and then removed from the list of generated tasks. The thread will sleep for one second and then do this all again.

# Lessons learned

During the development of this project, I came to a better understanding of the object-oriented programming principles. The threads handling was quite challenging, but it was a great feeling after I started to understand how they work. Also, displaying the content of a file in a text area (JTextArea) was something that I learned during this application.

All these are useful things that will help me in the future for the incoming projects.

# **Bibliography**

<http://www.javamex.com/tutorials/synchronization_volatile.shtml>

<https://www.tutorialspoint.com/uml/uml_standard_diagrams.htm>

<https://www.tutorialspoint.com/uml/uml_standard_diagrams.htm>

## 