DOCUMENTATION

ASSIGNMENT 2 – QUEUES MANAGEMENT APPLICATION USING THREADS AND SYNCHRONIZATION MECHANISMS

NUME STUDENT: Bumbuc Ioana

GRUPA: 30421

# CUPRINS

[1. Assignment’s objective 3](#_Toc95297885)

[*2.* Problem analysis, modelling, scenarios, use-case*s 3*](#_Toc95297886)

[3. Design 3](#_Toc95297887)

[4. Implementation 3](#_Toc95297888)

[5. Results 3](#_Toc95297889)

[6. Conclusions 3](#_Toc95297890)

[7. Bibliography 3](#_Toc95297891)

# Assignment’s Objective

The main objective of this assignment is to design and implement a queue management system that provides an efficient queue allocation mechanism in the form of a user application. It must simulate several clients arriving for service, waiting at queues, performing their task and finally leaving the queue. The application should compute the average waiting time.

The sub-objectives of this assignment are the following:

* Analyzing the problem and identifying its requirements (detailed in chapter 2)
* Designing the simulation application (chapter 3)
* Implementing the simulation application (chapter 4)
* Testing the simulation application (chapter 5)

# Problem analysis, modelling, scenarios, use-cases

The problem that this application wishes to solve is that of poor queue management. This issue leads to high waiting times for the clients and inefficient usage of resources.

* 1. **Functional Requirements**

The application should be able to read and process input provided by the user, through a graphical user interface. The input given by the user is the number of clients, number of queues, the maximum simulation time period, and the time intervals for arrival and service times and they are used to setup the simulation.

Furthermore, the simulation app should allow users to start the simulation after inserting the data, by pressing a button. The results of the simulation will be written in a text file, representing the log of events describing the evolution of the queues after each unit of time.

* 1. **Non-Functional Requirements**

Non-functional requirements refer to how the application behaves and what limits there are on its functionality. Thus, the application should be intuitive, easy to use by the user, while the output of the simulation must be concise and relevant.

* 1. **Theoretical Considerations – Java Concurrency**
     1. **Programs. Processes. Multitasking.**

A program can be defined as a series of instructions that can be put into a computer in order to make it perform an operation. In other words, a program is an algorithm written in a programming language. A process, however, is an instance of a program running in a computer, close in meaning to a task, having all system resources allocated by the operating system. A process executes a series of instructions sequentially, but what if we want to execute more processes at the same time?

This is where multitasking becomes useful. In computer science, multitasking refers to the concurrent executions of more processes in an interleaved manner, while they share common resources such as CPUs and memory. Multitasking can be cooperative or preemptive.

Cooperative multitasking is a style in which the operating system is never the one initiating the switch from a running process to another process. Instead, processes cooperate, sharing resources and yielding control periodically when they’re in blocked state. On the other hand, preemptive multitasking consists of allowing to each process a “slice” of operating time by the operating system. Thus, at any moment in time a process can be either I/O bound (waiting for input or output) or utilizing the CPU (CPU bound).

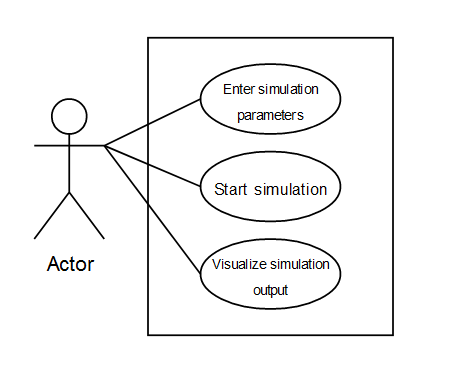
* + 1. **Threads**

A thread of execution is defined as the smallest sequence of instructions that can be run independently by the operating system, playing the role of a scheduler. Threads allows a program to run efficiently by doing multiple tasks in parallel. One of the advantages of using threads is that extensive tasks can be performed in the “background”, without disrupting the main thread.

A process can define more threads inside it, that share the same memory space. In this way, code that runs concurrently can exchange data more easily without overhead. However, threads that share data can become prone to race conditions creating bugs that are difficult to isolate and solve.

In Java, threads can be implemented in two ways: by extending the Thread class and overriding it’s “run” method, or by implementing the Runnable interface. In this assignment I chose to implement the Runnable interface. There exists several differences between these two approaches, such as:

* A class that extends “Thread” cannot extend another class, while one implementing “Runnable” can extend or implement anything
* Extending “Thread” means extending thread and job specific behavior code, while “Runnable” provides only the functionality needed for the “run” method.
* “Thread” defines the identity of the class extending it, while “Runnable” describes the peripheral abilities of the class implementing it.
  1. **Use Cases**



* + 1. **Enter simulation parameters**

Primary actor: the user

Main success scenario: User inserts data into the respective fields of the graphical user interface – number of clients, number of queues, simulation time, minimum and maximum times for arrival and for service.

Alternative sequence: The data inserted by the user is invalid and an exception will appear when starting the simulation.

* + 1. **Start simulation**

Primary actor: the user

Main success scenario: User clicks on “Start simulation” button after inserting the data in the text fields.

Alternative sequence: The simulation fails due to invalid input

* + 1. **Visualize simulation output**

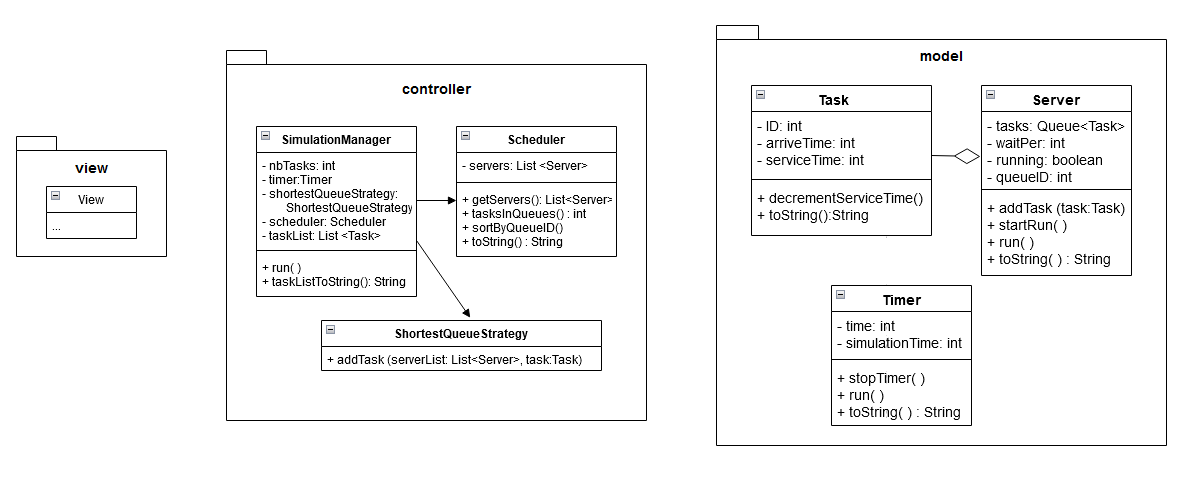
Primary actor: the user

Main success scenario: After the simulation is complete, the output will be written into a text file given by the programmer. The user can visualize the output in the form of an event log.

Alternative sequence: The text file doesn’t exist, in which case an exception appears.

1. **Problem design**

The design of this problem follows the MVC architectural pattern. There are three packages: the Model, which handles the data in the form of tasks and servers, the View, which provides the user interface for the user to interact with the program, and the Controller, which is responsible for the simulation as well as the interchange of data between view and model.



* 1. **Model Package**

The model package is the one responsible of data management. It is the package that contains the classes which actually perform the computations needed for the application. The classes contained in the model are Task, Server and Timer.

The Task class corresponds to a client that requests a service. A task is defined by its ID and two parameters that represent moments of time, that is the time of arrival and the time of service.

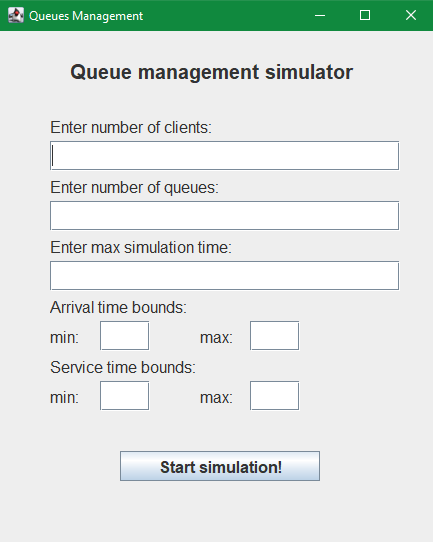
The Server class represents a queue of tasks that need to be performed. An object of Server class has an ID, a waiting period, a list of tasks and a variable that tells us whether the queue is open or closed.

The Timer class is used when simulating for incrementing periods of time until reaching the total time of simulation.

* 1. **View Package**

The view contains one class that is the graphical user interface. It allows the user to enter the parameters of the simulation, namely the number of clients and queues, the total simulation time and the time bounds of arrival of the clients and service.

The user interface is composed of labels, text field for the data to be inserted and a button which commands the start of the simulation.



* 1. **Controller Package**

The controller is the one responsible for setting up and starting the simulation. It is composed of 4 classes: Scheduler, ShortestQueueStrategy, SimulationListener, SimulationManager.

The Scheduler is the class that manages the queues. It has a list of servers, and it is the one which creates the queues.

The ShortestQueueStrategy class implements the idea of choosing the queue where to place the next task that arrives at a given moment. It is the one responsible of taking this decision, according to the shortest queue available.

SimulationManager class is the one that sets up the simulation. It takes data from the View class and writes the log of events into a text file. It creates a list of randomly generated tasks and periodically sends them to a queue by calling the strategy.

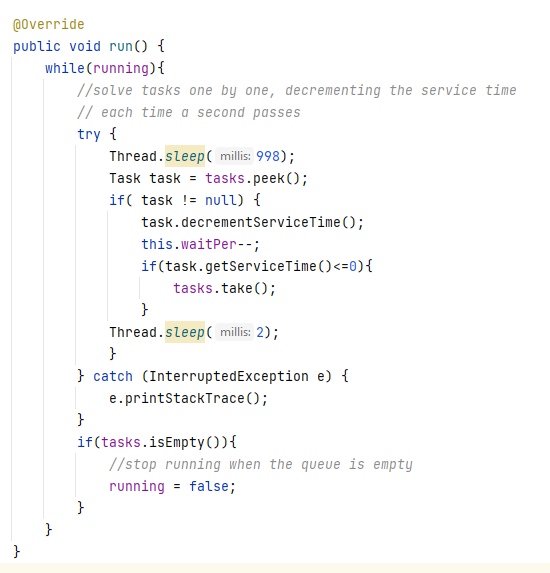
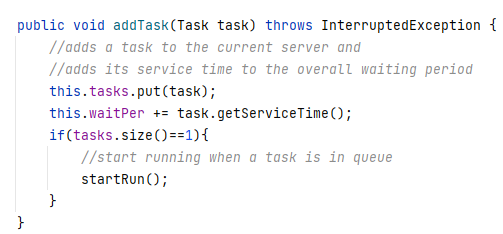
SimulationListener is a class that implements the ActionListener interface. Its role is to start the simulation by creating a new instance of the SimulationManager class when the button on the user interface is pressed.

1. **Implementation**
   1. **Task**



The important methods for this class are:

* decrementServiceTime ( ) – which is called after a time unit passes, to decrement the time left to do that task
* toString ( ) – this method is overridden by almost every class. It is called when writing the output into the text file for a cleaner code.
  1. **Server**

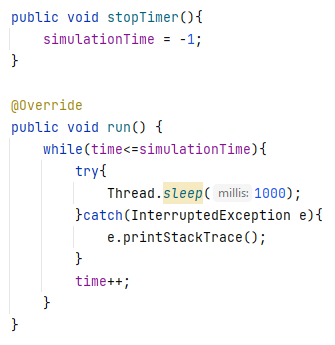
****

The method addTask adds a new task to the current queue and adds its service time to the overall waiting period. If the size of the queue is 1 it means that the thread should start running.

The method run ( ) implements the functionality of one thread, representing a queue. The method solves each task in the order that they appear, by decrementing the service time. If the queue is empty, the thread is stopped. “Run” would be called again by addTask if another task is added in the queue afterwards.

* 1. **Timer**

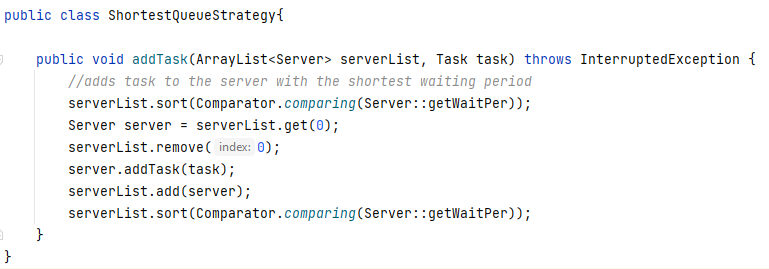
The timer is used to keep track of time passing in the simulation. When it is run, the variable time is incremented after each second. The method stopTimer( ) makes simulation time equal to -1. That way, the while loop will stop and thus the timer stops.



* 1. **Scheduler**

The role of the scheduler class is to create as many queues as the user introduces. It also provides a method returning the total number of tasks, as well as a method that sorts the queues by ID for better printing.

* 1. **ShortestQueueStrategy**



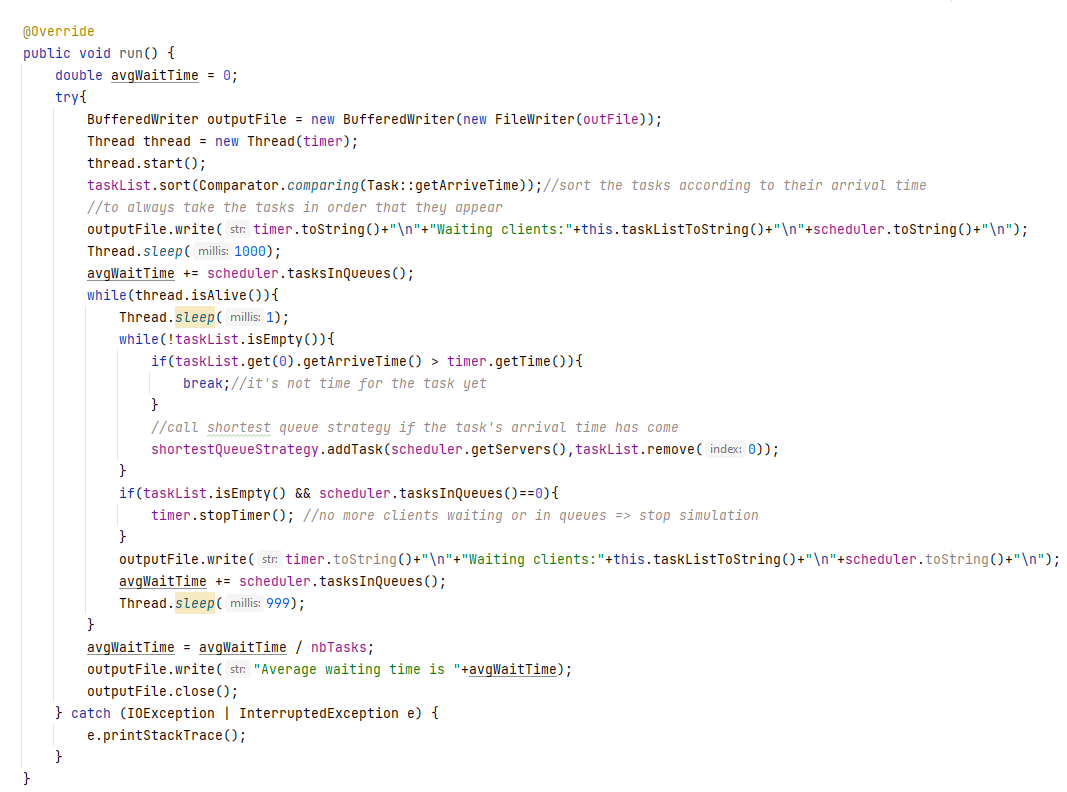
The method addTask chooses the most appropriate queue to add the task to.

* 1. **SimulationManager**

This is the class that implements the functionality of the queue simulator. It takes input from the user interface and generates a number of random clients with parameters in the bounds given.

It creates a timer thread that would count the seconds that pass in the simulation. At each step, the method checks if arrival time of the next client is reached. If it is, then it calls the strategy to add that client to the appropriate queue. The timer is stopped when there are no more clients in queues or tasks in the task list. Otherwise, it stops after the whole simulation time is over.

After each second, the state of the simulation is printed by calling the overridden method toString() on every object involved (timer, task list, scheduler);



* 1. **View**

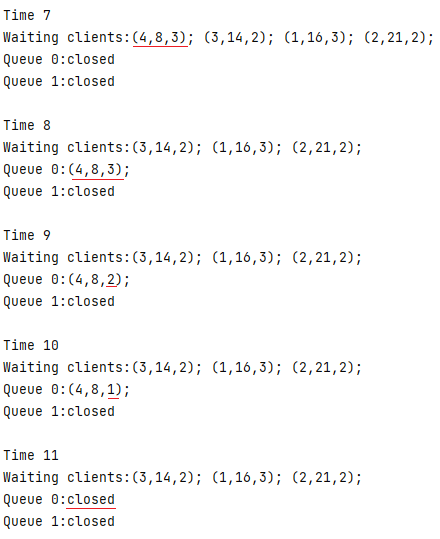
The View class extends JFrame and consists of a method which adds every component on the screen. There are several methods which return the data inserted by the user in the text fields.

1. **Results**

The simulation is tested on the 3 sets of input given by the assignment. The results of these tests are printed in three different files.

* 1. **Test 1**
* N = 4
* Q = 2
* T max simulation = 60
* T min arrival = 2
* T max arrival = 30
* T min service = 2
* T max service = 4

The simulation ends in 23 seconds, without using the entire time given for the simulation. The average waiting time computed is 2.5 seconds. The following example shows how client with ID equal to 4 enters the simulation, its service time is decremented until finally it leaves.



* 1. **Test 2**
* N = 50
* Q = 5
* T max simulation = 60
* T min arrival = 2
* T max arrival = 40
* T min service = 1
* T max service = 7

For a set of randomly generated clients, the simulation stops after 45 seconds and the average waiting time computed is 4.68.

* 1. **Test 3**
* N = 1000
* Q = 20
* T max simulation = 200
* T min arrival = 10
* T max arrival = 100
* T min service = 3
* T max service = 9

For these input data, the simulation finishes before it can solve all the tasks. The number of clients is very large compared to the number of queues. The average waiting time computed is 82.435.

1. **Conclusions**

The purpose of this application is to create a queue management system that can efficiently take care of clients arriving at queues. The aim is to solve the problem created by improper management, causing big waiting times for clients. The results of the simulation can create an image of what needs to be improved.

The results of the first two tests prove that for the given input parameters, the queues advance in proper time. On the other hand, the conclusion drawn from the third test is that for a number of clients that big, a bigger number of queues is needed to minimize waiting times.

1. **Bibliography**

* <https://en.wikipedia.org/wiki/Computer_multitasking>
* <https://en.wikipedia.org/wiki/Cooperative_multitasking>
* <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
* <https://www.w3schools.com/java/java_threads.asp>