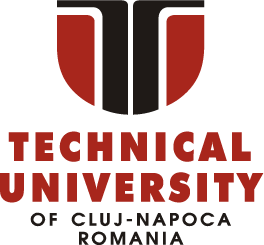
Technical University of Cluj-Napoca April, 2016

Programming Techniques

Laboratory - **HOMEWORK 3**

Queues management system

**

*Teacher*: Ioan SALOMIE *Student*: Iulia Maria ṢOMFELEAN

*Teacher Assistant*: Delia BALAJ Group 30425

1. 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;*

1. Dimensions of the problem

**Implementing queue processing across a pool of worker threads using basic Java thread synchronization and object monitor :**

This is the first in a series on Java concurrency patterns. Java has many built in classes and functionality to help write concurrent code, with varying ways to synchronize the activities of threads. For instance, there are monitors, locks, latches, atomic operations, delayed tasks, future tasks, barriers, the list goes on. In this installment, we'll implement queue processing across a pool of worker threads using basic Java Thread synchronization and object monitors.

**Multithreaded Queue Example**

Let's examine a common scenario in which tasks are executed by worker threads, and where the work is coordinated via a queue to ensure once-and-only-once execution of each task. In this case, we'll use the queue (itself a Java Object) as the monitor as well. Here's the overall design:

* Create a queue to act as a monitor and a way to transfer work
* Create a pool of worker threads (consumers) that each wait on the monitor (the queue)
* Create one or more producers that place items on the queue
* Notify the monitor when an item is placed on the queue, which in turn wakes up a worker to pull the next item off the queue

Let's start by looking at the worker's class, MyWorker, which itself extends Thread:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class MyWorker extends Thread {      private static int instance = 0;      private final Queue<String> queue;        public MyWorker(Queue queue) {          this.queue = queue;          setName("MyWorker:" + (instance++));      }        @Override      public void run() {          while ( true ) {              try {                  Runnable work = null;                    synchronized ( queue ) {                      while ( queue.isEmpty() )                          queue.wait();                        // Get the next work item off of the queue                      work = queue.remove();                  }                    // Process the work item                  work.run();              }              catch ( InterruptedException ie ) {                  break;  // Terminate              }          }      }        private void doWork(Runnable work) { ... }  } |

The code that creates a pool of MyWorker objects, which is essentially a Thread pool, looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for ( int i = 0; i < THREAD\_POOL\_SIZE; i++ ) {      // Create and start the worker thread, which will      // immediately wait on the monitor (the queue in      // this case) to be signaled to begin processing      MyWorker worker = new MyWorker(queue);      worker.start();  } |

The code to place work on the queue and signal a worker thread looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | synchronized ( queue ) {      // Add work to the queue      Runnable work = getMoreWork();      queue.add(work);        // Notify the monitor object that all the threads      // are waiting on. This will awaken just one to      // begin processing work from the queue      queue.notify();  } |

This code illustrates how straightforward basic Java thread synchronization is, although it is low-level. In Part 2 of this series, we'll look at the java.util.concurrent package and how it contains built-in classes to make synchronized queue and task processing even simpler.

Happy Coding!

* 1. Analyzing and modelling the problem

I decided that I will need a client generator to generate arriving clients and a client scheduler that spreads clients to the most cash register with the minimum waiting time.

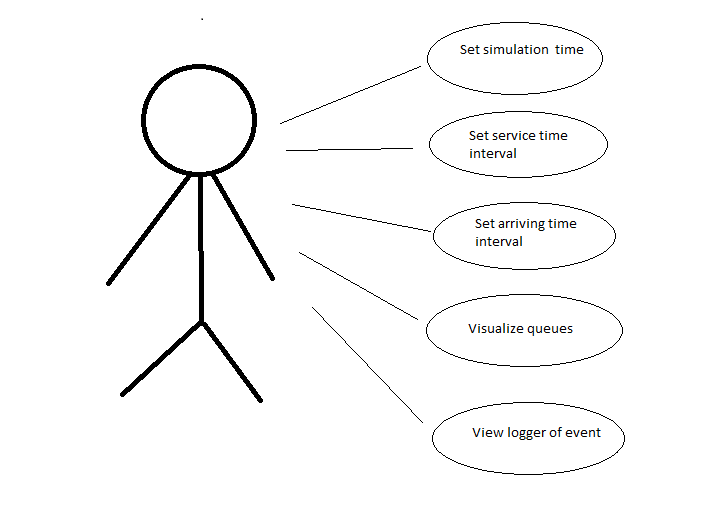
Each cash register is a thread, and every client has a service time and an arrival time.

* 1. Scenarios and use cases

A use case is a methodology used in system analysis to identify, clarify, and organize system requirements. The use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal.

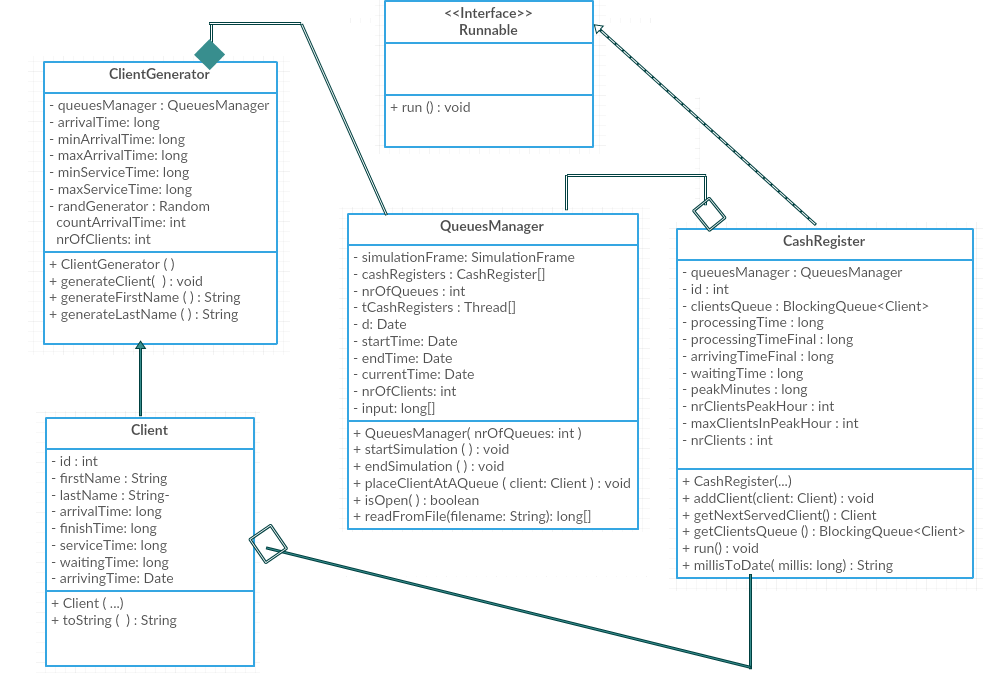
The use cases are strongly related to the user steps. I tried to design my interface in a user friendly mode, and that’s the result:

1. Implementation
   1. Diagrams
2. Use case diagrams



1. Class diagram

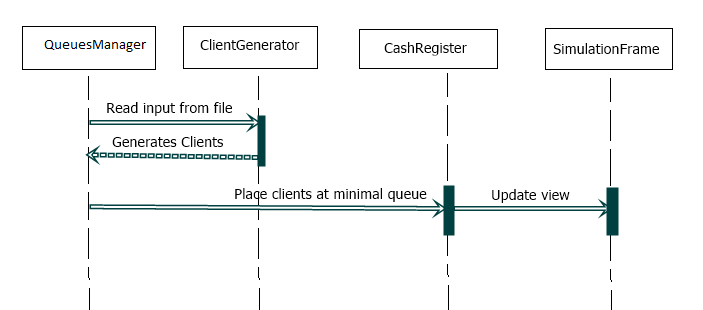
**Class diagram for the  *model* part**

****

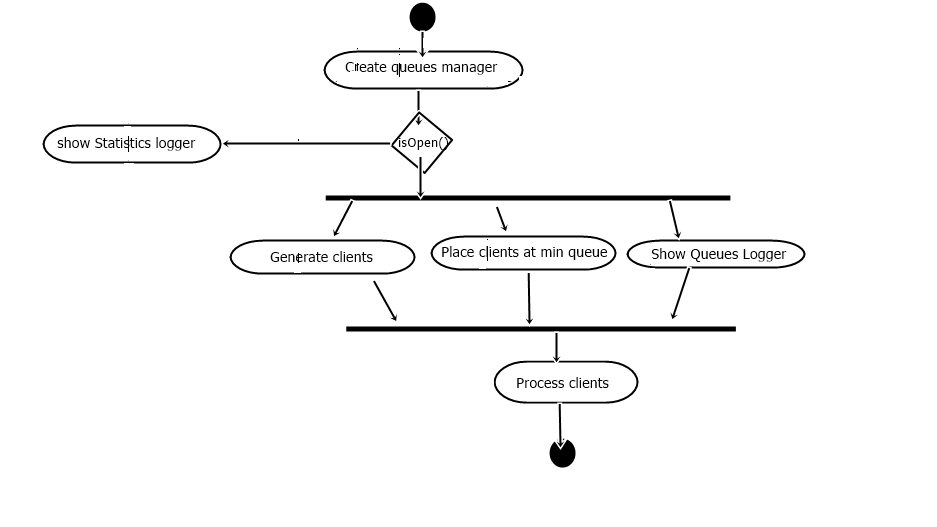
1. Sequence diagram

A **Sequence diagram** is an interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called **event diagrams** or **event scenarios**.

A sequence diagram shows, as parallel vertical lines (*lifelines*), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.



1. Activity diagram



**Activity diagrams** are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows). Activity diagrams show the overall flow of control.

Activity diagrams are constructed from a limited number of shapes, connected with arrows.The most important shape types:

* *rounded rectangles* represent *actions*;
* *diamonds* represent *decisions*;
* *bars* represent the start (*split*) or end (*join*) of concurrent activities;
* a *black circle* represents the start (*initial state*) of the workflow;
* an *encircled black circle* represents the end (*final state*).

Arrows run from the start towards the end and represent the order in which activities happen.

* 1. Data Structures
     1. ArrayList

It’s a resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null.

In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

* + 1. BlockingQueue

*public interface BlockingQueue<E>*

*extends Queue<E>*

A Queue 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 (either null 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. These methods are summarized in the following table:

Throws exception Special value Blocks Times out

Insert add(e) offer(e) put(e) offer(e, time, unit)

Remove remove() poll() take() poll(time, unit)

Examine element() peek() not applicable not applicable

A BlockingQueue does not accept null elements. Implementations throw NullPointerException on attempts to add, put or offer a null. A null is used as a sentinel value to indicate failure of poll operations.

A BlockingQueue may be capacity bounded. At any given time it may have a remainingCapacity beyond which no additional elements can be put without blocking. A BlockingQueue without any intrinsic capacity constraints always reports a remaining capacity of Integer.MAX\_VALUE.

BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the Collection interface. So, for example, it is possible to remove an arbitrary element from a queue using remove(x). However, such operations are in general not performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled.

BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. However, the bulk Collection operations addAll, containsAll, retainAll and removeAll are not necessarily performed atomically unless specified otherwise in an implementation. So it is possible, for example, for addAll(c) to fail (throwing an exception) after adding only some of the elements in c.

A BlockingQueue does not intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added. The needs and usage of such features tend to be implementation-dependent. For example, a common tactic is for producers to insert special end-of-stream or poison objects, that are interpreted accordingly when taken by consumers.

* 1. Packages

Java packages help in organizing multiple modules and group together related classes and interfaces. Packages avoid name conflicts.

In object-oriented programming development, model-view-controller (MVC) is the name of a methodology or design pattern for successfully and efficiently relating the user interface to underlying data models. The MVC pattern is widely used in program development with programming languages such as Java, Smalltalk, C, and C++.

The MVC pattern has been heralded by many developers as a useful pattern for the reuse of object code and a pattern that allows them to significantly reduce the time it takes to develop applications with user interfaces.

The model-view-controller pattern proposes three main components or objects to be used in software development:

* A *Model* , which represents the underlying, logical structure of data in a software application and the high-level class associated with it. This object model does not contain any information about the user interface.
* A *View* , which is a collection of classes representing the elements in the user interface (all of the things the user can see and respond to on the screen, such as buttons, display boxes, and so forth)
* A *Controller* , which represents the classes connecting the model and the view, and is used to communicate between classes in the model and view.

Given the fact that I have chosen a Model-View-Controller Pattern, I splitted my classes into the corresponding packages and, alongside them, some useful classes :

* **model**: contains the “brain” of the project, the classes that model the problem.
* **view**: represents the GUI
* **controller**: the controller part interconnects the model and the view
* **utilities:** contains useful classes, such as a class where I keep all my constants

. . . . . . . . … . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

* 1. Class Design

The whole idea of splitting your program into classes is based on a general rule named divide and conquer. This paradigm can be used almost everywhere: you divide a problem into smaller problems and then you solve these little, simple and well-known problems .  
Dividing your program into classes is one of the types of division which started to become common in last decade. In this programming paradigm we model our problem by some objects and try to solve the problem by sending messages between these objects.

I tried to design my project in the MVC architecture, that’s why I have 3 principal parts:

* + 1. The model – contains the logic of the application
* **Client**: contains data about the customer, such as name, email, username, password, and if it is an administrator or a regular user.
* **QueuesManager:** contains data about a product, such as title, author, etc.If the quantity of the product is smaller than MAX\_LIMITED\_QUANTITY, then the status of the product will be LIMITED. If the quantity it’s equal with zero, the product becomes UNAVAILABLE.
* **CashRegister:** it’s a class that defines an ordered product, the product itself along the ordered quantity.
* **ClientGenerator:** contains an id, an id of the customer that makes the order and a list of ordered products. In the *processOrder()* method, we check if we have enough products in stock
  + 1. The controller – contains the linking between the model and the view of the application.

Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

* **Controller**
  + 1. The view – View represents the visualization of the data that model contains.
* **SimulationFrame:** represents the login window that pops up at the starting of our application.
* **LoggerPanel:** represents the view for the case in which the logged in user is a regular user.
* **QueuePanel:** represents the view for the case in which the logged in user is a administrator.

I used JTable for various operations:

The JTable is used to display and edit regular two-dimensional tables of cells.

The JTable has many facilities that make it possible to customize its rendering and editing but provides defaults for these features so that simple tables can be set up easily.

JTables are typically placed inside of a JScrollPane. By default, a JTable will adjust its width such that a horizontal scrollbar is unnecessary. To allow for a horizontal scrollbar, invoke [setAutoResizeMode(int)](https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html" \l "setAutoResizeMode(int)) with AUTO\_RESIZE\_OFF. Note that if you wish to use a JTable in a standalone view (outside of a JScrollPane) and want the header displayed, you can get it using [getTableHeader()](https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html" \l "getTableHeader()) and display it separately.

* + 1. The comparators – are the classes in which I override the compare method, to define the order of the elements in the TreeSet.

I have 3 comparators, one for the orders TreeSet, one for the Warehouse TreeSet, and one for the Order TreeSet:

* **OrderComparator**
* **ProductComparator**
* **OrderedProductComparator**

All of them override the *compare()* method. They return 0 in case of equality, and 1 or -1, in the other cases.

* + 1. The utilities
* Main : which include a instantiation of the view and creates a controller.
* Constants: is a class which contains values often used inside, such as the number of queues, the limits of the different intervals used in my application.
  1. Algorithms
     1. Optimal queue

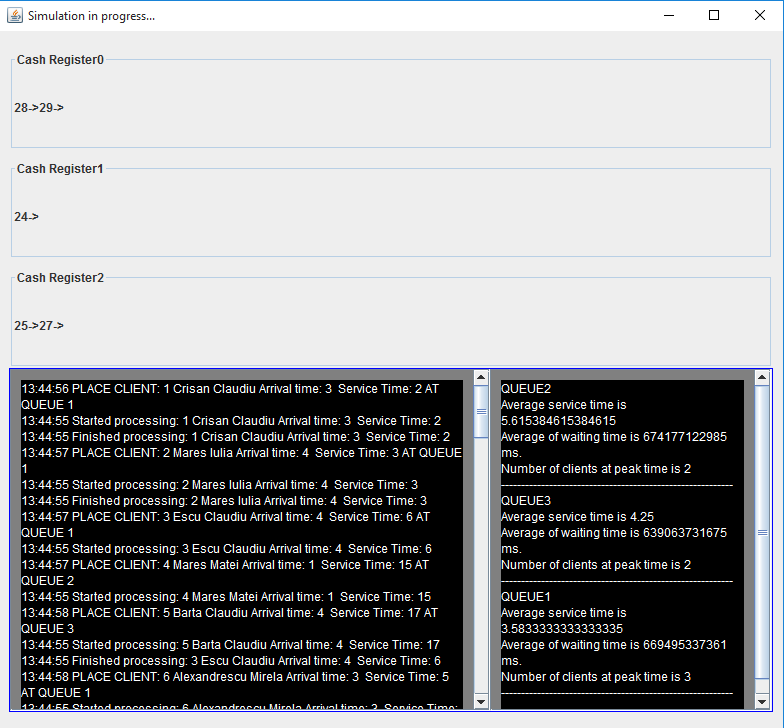
As the basic algorithm of my application, I can name the algorithm for selecting the queue with the minimum processing time. This helps to position the clients in an effective way to process their services faster.

After a certain interval of time passes, one queue is closed and by using the same algorithm of positioning a client at some queue, the clients from the closed queue are assigned to others.

* 1. User Interface

The interface represents a simulation of a queue processing system , it contains a logger of events where the user can observe the changes in the queues and a logger with statistics, that appear at the end of the simulation time. Each client is positioned at an optimal queue and, after it gets served, it disappears, being followed by other clients.

. . . . . . . . . . . . . . . . . . . . . .

 . . . . . . .

1. Implementation and testing

This application was developed and tested only in Eclipse, but this thing should not affect it’s portability.

Because time is involved, the results may differ with each run application.

1. Results and improvements

The application is a simulation of how clients are processed by the shop cash registers. As the application is developed on a Java platform, it is highly portable and allows it to run on several operating systems (as long as they have the Java SDK installed).

As improvements, I would enumerate:

* When it’s needed, to increase efficiency, open new cash registers
* Make the view more intuitive and creative

1. Conclusions

By the means of this project I managed to improve my knowledge about queues processing and how they are implemented on a system, I learned about Threads and about synchronization.

Also, I’ve noticed the importance of a logger and experienced using different libraries for date and time.

1. Bibliography
   1. Object-Oriented Programming - Lecture Slides of prof. Marius JOLDOS
   2. Programming Techniques – Lectures of prof. Ioan SALOMIE
   3. Head First Java 2nd Edition, Kathy SIERRA
   4. [www.stackoverflow.com](http://www.stackoverflow.com)
   5. <https://docs.oracle.com/javase/tutorial/essential/concurrency/>
   6. <http://whatis.techtarget.com/definition/model-view-controller-MVC>
   7. wikipedia.org
   8. <http://www.drdobbs.com/parallel/java-concurrency-queue-processing-part-1/232700457>