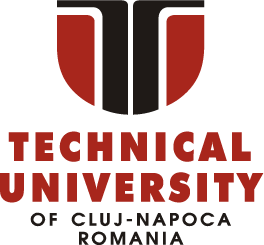
Technical University of Cluj-Napoca April, 2016

Programming Techniques

Laboratory - **HOMEWORK 2**

Order management for processing customer orders

**

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

*Teacher Assistant*: Delia BALAJ Group 30425

1. Objective

*Consider an application* ***OrderManagement*** *for processing customer orders.*

*The application uses (minimally) the following classes:* ***Order, OPDept (Order Processing Department), Customer, Product,*** *and* ***Warehouse****.*

*The classes OPDept and Warehouse use a* ***BinarySearchTree***  *for storing orders.*

*a. Analyze the application domain, determine the structure and behavior of its classes, identify use cases.*

*b. Generate use case diagrams, an extended UML class diagram, two sequence diagrams and an activity diagram.*

*c. Implement and test the application classes. Use javadoc for documenting the classes.*

*d. Design, write and test a Java program for order management using the classes designed at question c). The program should include a set of utility operations such as under-stock, over-stock, totals, filters, etc.*

1. Dimensions of the problem

**An order management system, or OMS, is a computer software system used in a number of industries for order entry and processing.**

Orders can be received from businesses, consumers, or a mix of both, depending on the products. Offers and pricing may be done via catalogs, websites, or broadcast network advertisements.

An integrated order management system may encompass these modules:

* Product information (descriptions, attributes, locations, quantities)
* Inventory available to promise (ATP) and sourcing
* Vendors, purchasing, and receiving
* Marketing (catalogs, promotions, pricing)Customers and prospects
* Order entry and customer service (including returns and refunds)
* Financial processing (credit cards, billing, payment on account)
* Order processing (selection, printing, picking, packing, shipping)
  1. Analyzing and modelling the problem

We have to implement a management order system, of course, not so high-level as in the definition mentioned above.

Our management system will be accesed by users, which are divided in 2 categories:

1. *The administrator*, which has as responsabilities the maintenance of the system.

He/she can perform the following operations:

* View stock and warehouse items
* View orders
* Add items
* Remove items
* Modify stock

1. *The customer*, which is a normal user, can do the following:

* View items
* Search for certain items, corresponding to different fields
* Buy items
* View order

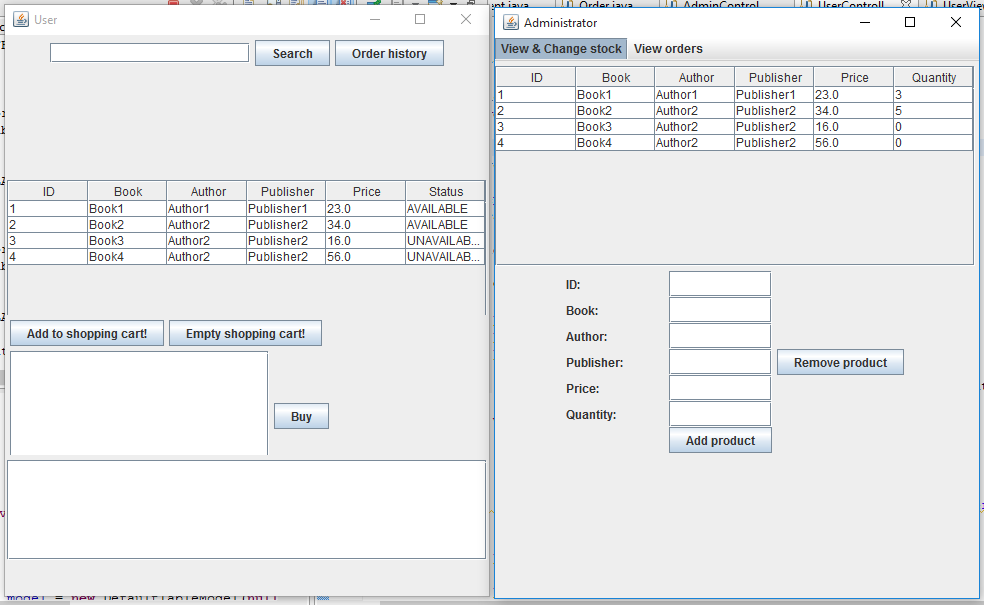
The *warehouse* contains products from the domain of *Books.*

Each *product*  posseses a series of characteristics:

* ID
* Title
* Author
* Publisher
* Price
* Quantity
  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.

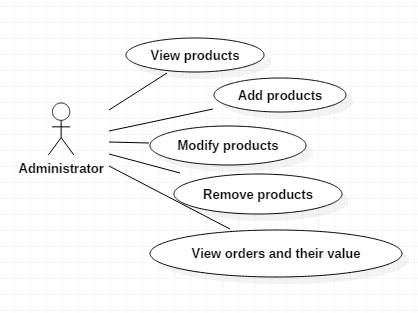
The use cases are strongly related to the user steps. I tried to design my interface in a user friendly mode, and that’s the result:



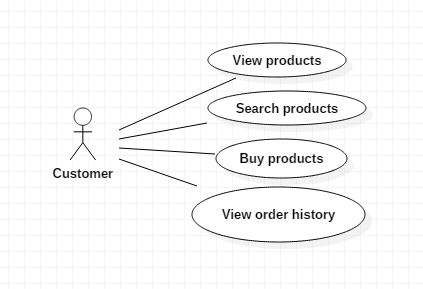
1. Implementation
   1. Diagrams
2. Use case diagrams

The use case diagram presents the actors, which in our case can play the role of *administrator* or *regular user*(=*customer).*

The administrator has the rights to manage the system, he can perform CRUD type operations: create, read, update, delete.

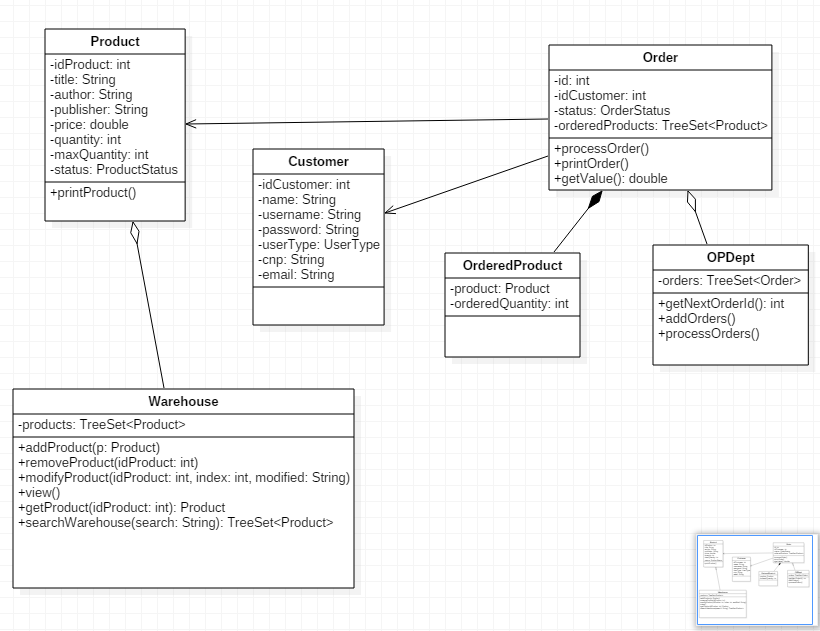


The regular user is a customer that wants to buy products, he can search them using different keywords, and buy the desired quantity.



1. Class diagram

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

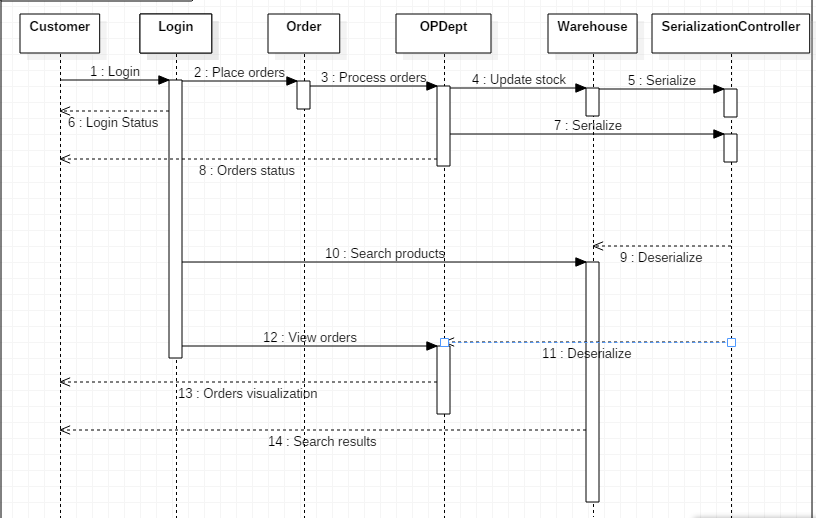


1. Sequence diagram

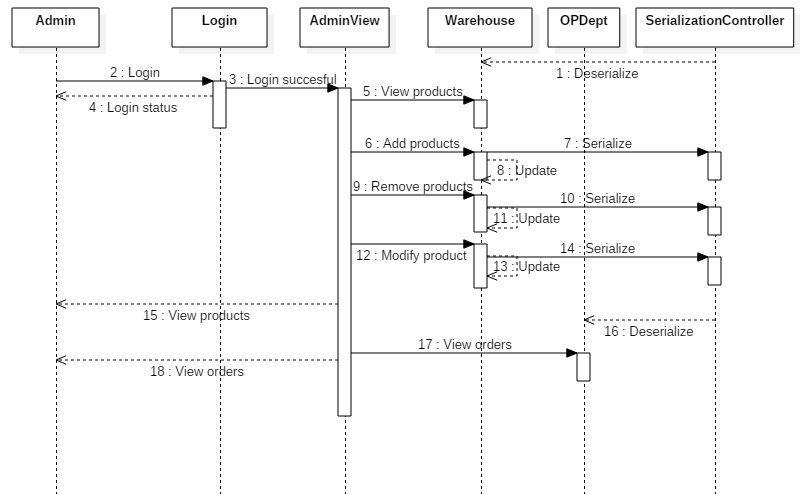
A **Sequence diagram** is an interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and 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.

*Sequence diagram for regular user:*

**

Sequence diagram for administrator



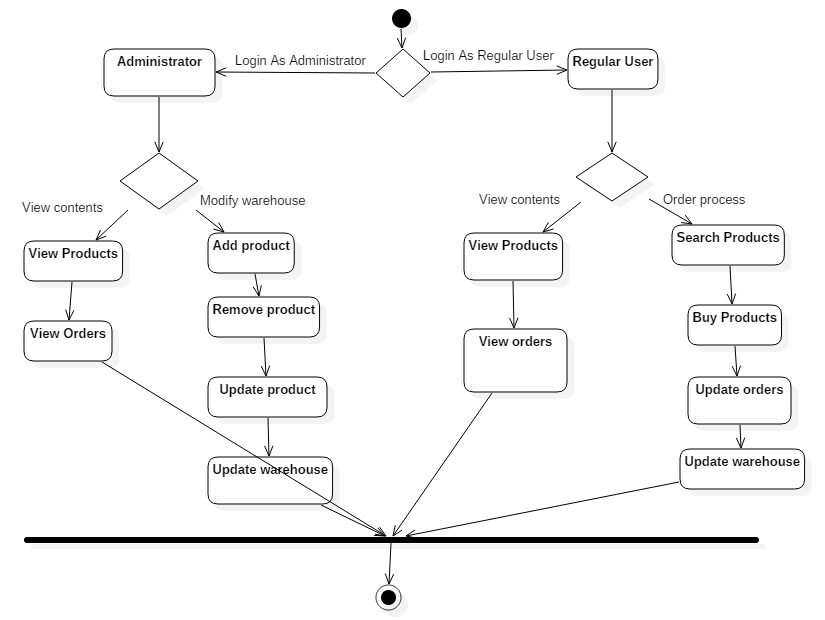
1. Activity diagram

**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. TreeSet

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

* guarantees log(n) time cost for the basic operations (add, remove and contains)
* guarantees that elements of set will be sorted (ascending, natural, or the one specified by you via its constructor) (implements [SortedSet](http://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html))
* doesn't offer any tuning parameters for iteration performance
* offers a few handy methods to deal with the ordered set like [first()](http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html#first--), last(), [headSet()](http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html" \l "headSet-E-), and [tailSet()](http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html" \l "tailSet-E-) etc
* [**TreeSet**](https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html#TreeSet(java.util.Comparator))([**Comparator**](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<? super [**E**](https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html)> comparator)
* Constructs a new, empty tree set, sorted according to the specified comparator.

According to these type of constructor, I build my own comparators which give me the kind of order I desire for my TreeSets.

* 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
* **comparators** : contains the comparators used for the TreeSet objects
* **utilities:** contains useful classes, such as a class where I keep all my constants, enumeration classes and the main one.

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

* 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
* **Customer**: contains data about the customer, such as name, email, username, password, and if it is an administrator or a regular user.
* **Product:** 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.
* **OrderedProduct:** it’s a class that defines an ordered product, the product itself along the ordered quantity.
* **Order:** 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
* **OPDept:** is the class that contains all the orders, and processes them, setting their status and computing their value.
* **Warehouse:** contains a TreeSet of products. We can add, remove or modify an item from the warehouse.
  + 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.

* **SerializationController:** it’s the class that deals with the serialization and deserialization of the Warehouse and OPDept.
* **LogInController:** checks if the introduced username and password are correct.
* **AdminController:** interconnects the admin view and model, has listeners that implement some logic.
* **UserController:** interconnects the user view and model, has listeners that implement some logic.
  + 1. The view – View represents the visualization of the data that model contains.
* **LogInView:** represents the login window that pops up at the starting of our application.
* **UserView:** represents the view for the case in which the logged in user is a regular user.
* **AdminView:** 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 used inside the code, such as usernames and password used for login and the maximum quantity that can exist in stock for products: *MAX\_QUANTITY* or the quantity below the product is in limited stock: *MAX\_LIMITED\_QUANTITY*.
* OrderStatus: is an enumeration that defines the status of an order: ***PENDING***, ***PROCESSING***, ***DELIVERED***, ***FAILED.***
* ProductStatus: is an enumeration that defines the status of a product: ***AVAILABLE***, ***LIMITED***, ***UNAVAILABLE.***
* UserType : an enumeration that defines the users that can access our system: ***ADMINISTRATOR***, ***REGULAR\_USER.***
  1. Algorithms
     1. Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

I used serialization to store my warehouse and orders in external files, in order to maintain changes even if the application is closed.

* + 1. Stock and processing orders

The products are stored in the TreeSet from the warehouse, and every change we make to the products, they are updated.

1. A customer can search for products and buy a list of them.
2. If the quantity in the stock is not exceeding the ordered quantity, then the order is finished with success, following to be delivered.
3. In the other case, the order receives as status : failed, and there is signaled the UNDERSTOCK ( = supply with less stock than required or necessary) .
   * 1. Search products

The user can search after a different product, writing it into the search field.

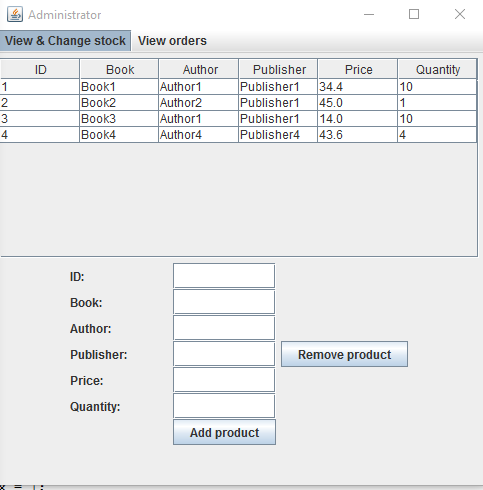
As a result, a TreeSet with the product fulfilling the search keyword is returned.

* 1. User Interface

The user interface has the role of connecting the user with our application. He can perform some operations to manage orders and stock.

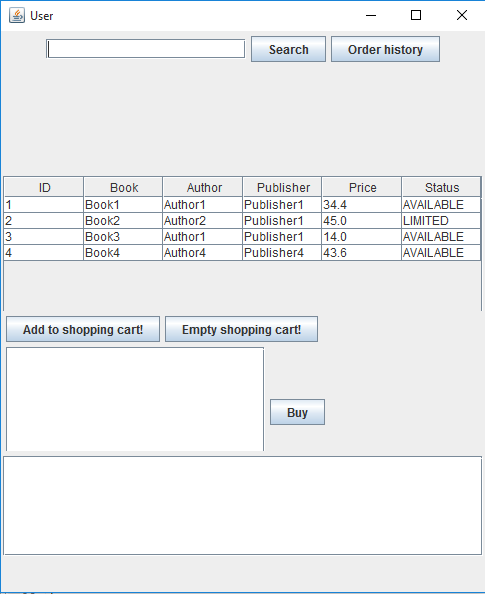
The user interface splits in two parts : the administrator view and the regular user view.

* + 1. The administrator view



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

* + 1. The regular user view

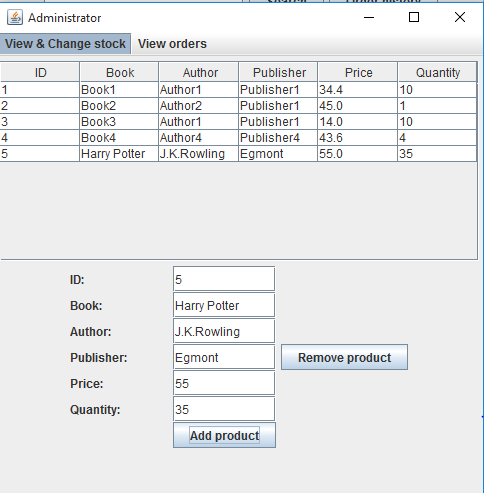
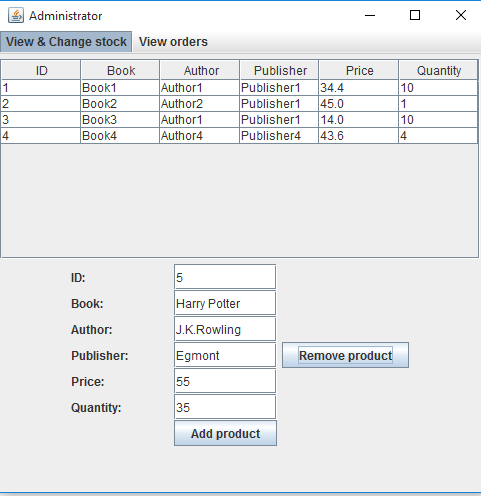


1. Implementation and testing

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

* 1. Administrator mode:

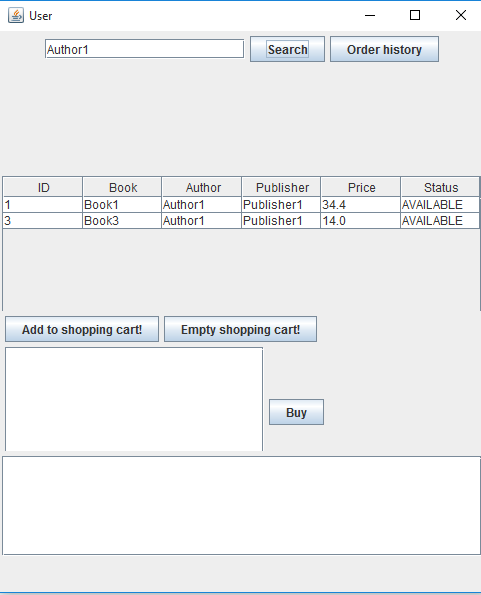
1. View product : click on “*View & Change Stock”*
2. Add a product: introduce the desired product in the corresponding text fields and click on “*Add product”*



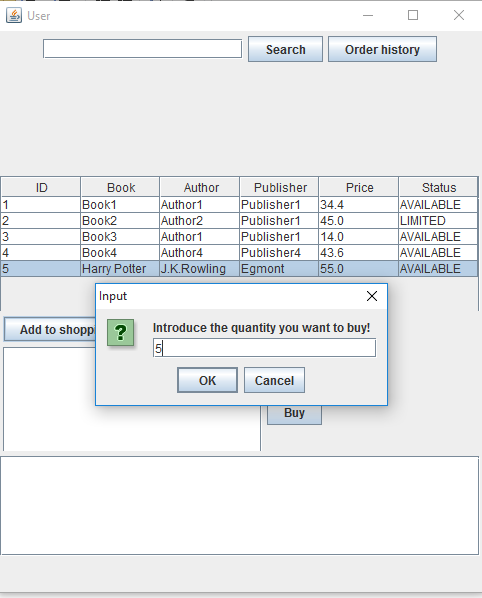
1. Remove a product: select the product you want to remove from the table, and then click “*Remove product”*
2. Modify product: just introduce your changes in the table, and then press “Enter”.

The changes will be updated in the warehouse.

1. View orders: select the menu “*View orders”* to visualize the details about the existing orders
   1. The regular user mode
   2. Searching a product: introduce the keyword you want to search in the text field and click on “*Search” button*. If the keyword generates some results, they are shown in the table. If not, an *“Item not found”* message will pop up.



* 1. Refreshing the table: empty the search text field and click on “*Search”* button.
  2. View order history: just click on *”Order History”* button and the orders will appear in the box from the bottom of the window.
  3. Add items to shopping cart : select a product, and introduce the wanted quantity in the message box that appears. The product will appear in the box which represents the shopping cart.
  4. Buy items: when you finally added all the products along the desired quantities you want to order, click on “*Buy”.* The shopping cart will empty itself.



* 1. Empty the shopping cart: if, by mistake, you introduced an unwanted product, you can delete it by clicking “*Empty shopping cart!”.*

1. Results and improvements

The application is an user friendly and helpful application to perform 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:

* Adding more clients
* Making some changes to the interface to improve aspect
* Save invoices externally
* Using a database to store and retrieve the information

1. Conclusions

By the means of this project I managed to improve my knowledge about CRUD operations and how they are implemented on a system, I learned about TreeSet Collection and about serialization. Also, I experienced working on two levels, admin and customer, and how to synchronize in real time the changes made in the system . I found out how useful is to know how to use properly elements of Swing, such as JTable, JList, etc.

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/7/docs/api/java/util/TreeSet.html>
   6. <http://whatis.techtarget.com/definition/model-view-controller-MVC>
   7. wikipedia.org
   8. https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html