*Technical University of Cluj-Napoca  
Faculty of Automation and Computers  
Department of Computer  
2nd Semester 2016*

*Programming Techniques*

*Homework 2*

*Student: Ghiurutan Alexandru  
Group: 30425*

***Contents:***

1. *Problem Specification ................................................................ 3*
2. *Example of working .............................................................. 4*
3. *Design*

*3.1 Class Diagram …............................................... 6*

*3.2 The sequence diagram …………………………… 8*

*3.3 The Activity diagram……………………………. 9*

*3.4 Classes Design........................................................ 10*

*3.4.1 Gui Class*

*3.4.2 Client and Product Class*

*3.4.3 Warehouse Class*

*3.4.4 Order Class*

*3.4.5 OpDept Class*

*3.5 Packages and Interfaces............................................. 17*

*3.6 Serializable Interface …………………………...….. 18*

*3.7**Comparable Interface …........................................... 19*

*3.8 User Interface........................................................... 20*

1. *Using and testing the application …………………………….. 22*
2. *Conclusions …………………..……………………………..... 22*

***1. Problem Specification***

TP Lab – Homework 3

Consider an application OrderManagement for processing customer orders. The application uses (minimally) the following classes:

Order

OPDept (Order Processing Department)

Customer

Product

Warehouse

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

Analyse the application domain, determine the structure and behaviour of its classes, identify use cases and generate use case diagrams, an extended UML class diagram and two sequence diagrams.

Implement the application classes.

Use javadoc for documenting classes.

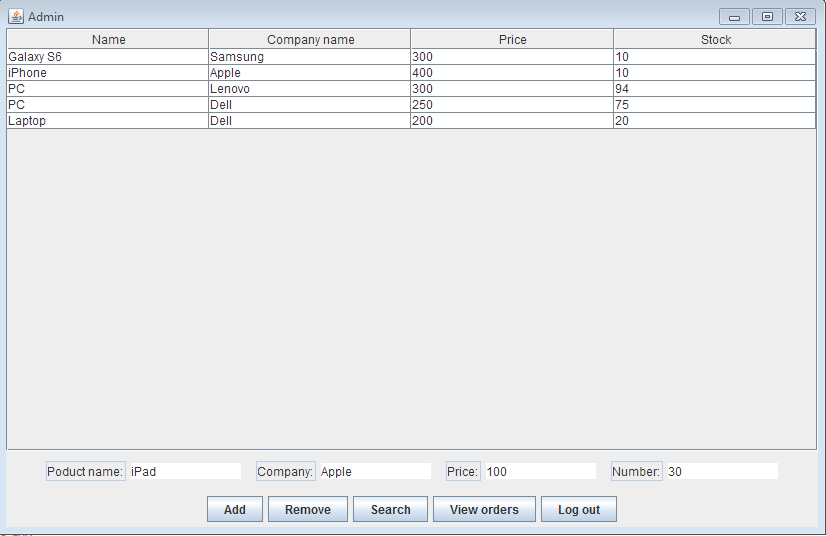
Implement a system of utility programs for reporting such as: under stock, overstock, totals, filters, etc.

Write the appropriate test drivers

***2. Example of working***

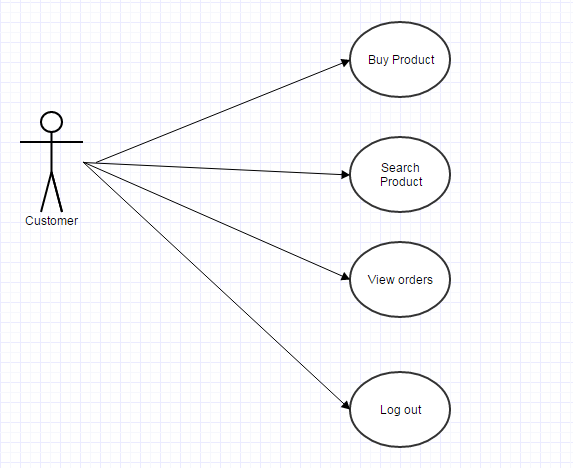
The user can select one of the 5 options displayed on the bottom side of the application window. Afterwards, depending on what the user selected he can add and remove products or view orders or even make an order.

Example of inserting the information for a Product:

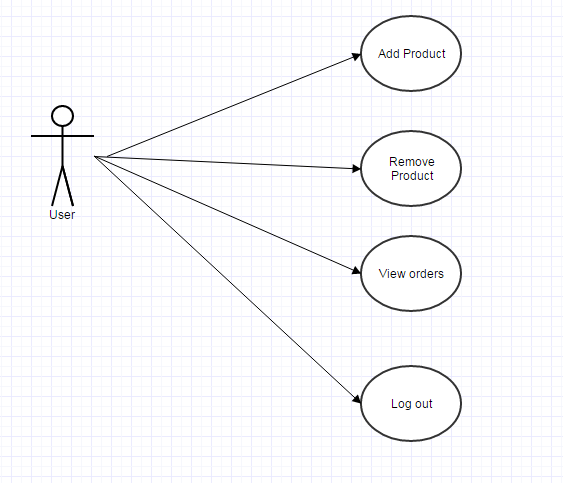
****

***3. Design***

***3.1 Use case diagram***



Such a diagram shows the customer’s options provided by the application. Consider the case of a client. He can order a product , view its previous orders or search for a product.

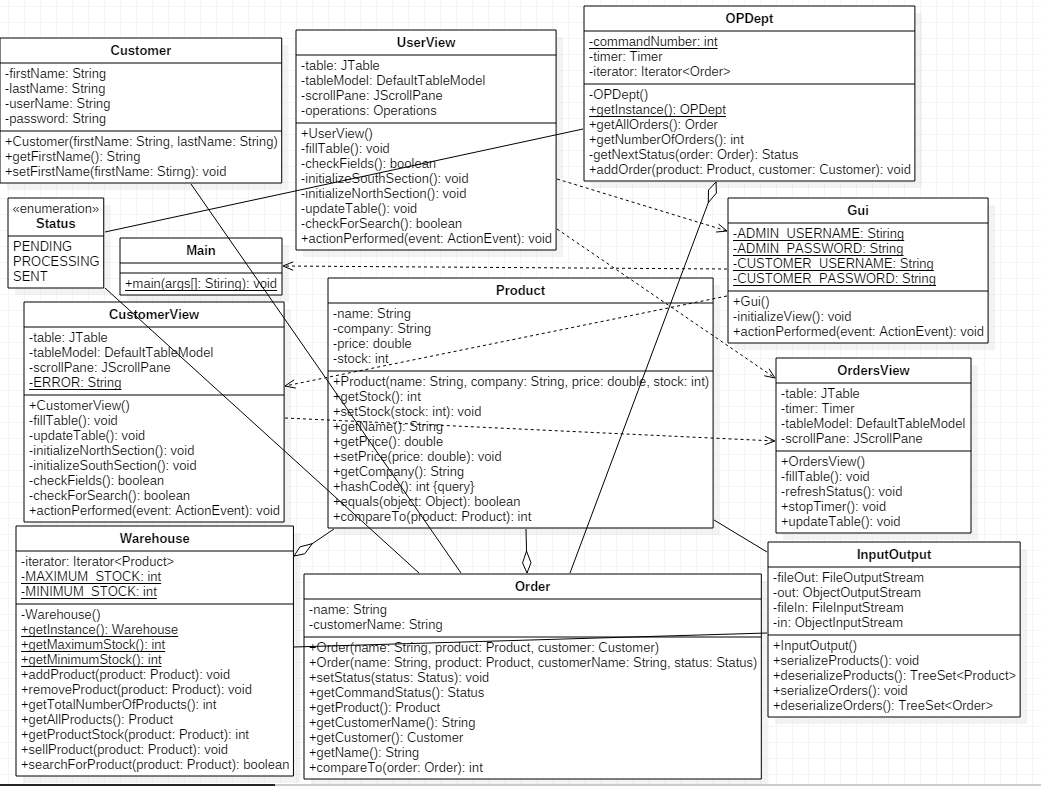


In this application we differentiate between users and admins, so as an admin we have the posibility of looking at the database and edit it as needed (you can delete products).

***3.2 Class Diagram***

For resolving the problem specifications, I’ve chosen to use eleven different classes and an enumeration ,so that the design of the application would be as good as possible. The names of the classes and relations between them are represented in the figure below.

About their use and the reasons why I have chosen to organize the objects as it is written are explained in section 3.2: “Classes Design”.



Class Diagram

To get a better view related to the attributes of each class, there are below the UML diagrams for each class. Thus, we can see every class with objects and their methods.

***3.3 The sequence diagram***

For adding a product:

UserView

Warehouse

Product

Add Product

Create Product

Retrieves Product

Returns set of products

Prints table

For removing a product:

CustomerView

Product

Warehouse

Create Product

Search Product

Retrieves Product

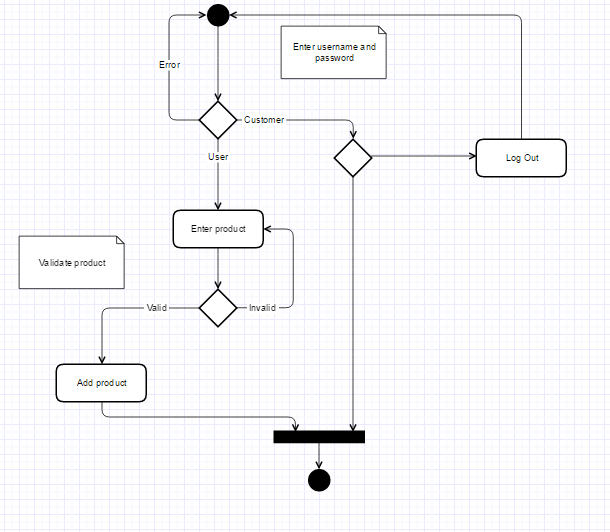
Product found

Remove Product

Prints table

***3.4Activity diagram***

***Shows the process of adding a new product by the admin.***

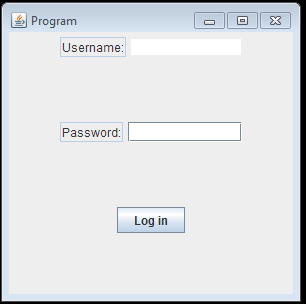
******

***3.5 Classes Design***

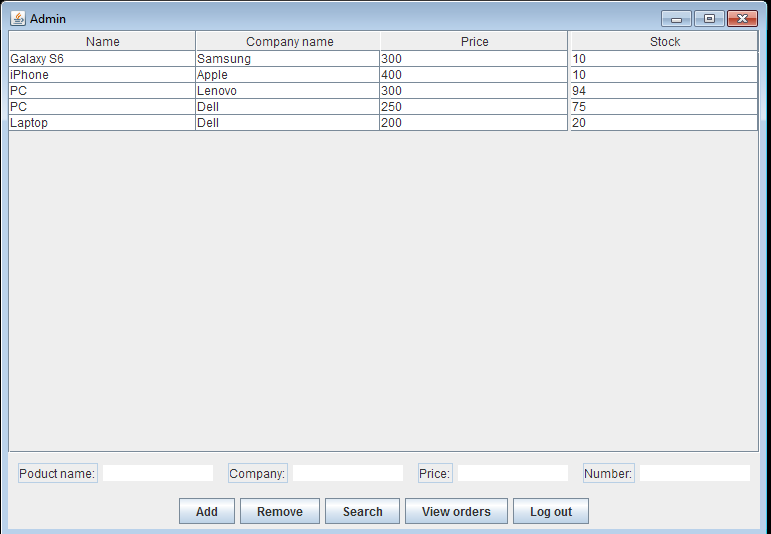
1. Gui Class : public class Gui extends JFrame

This class is designed to create a graphical user interface so the application would be easier to use. We consider the Gui class as a subclass of the predefined class "JFrame" so we can use objects of type "button", "frame" or "panel".

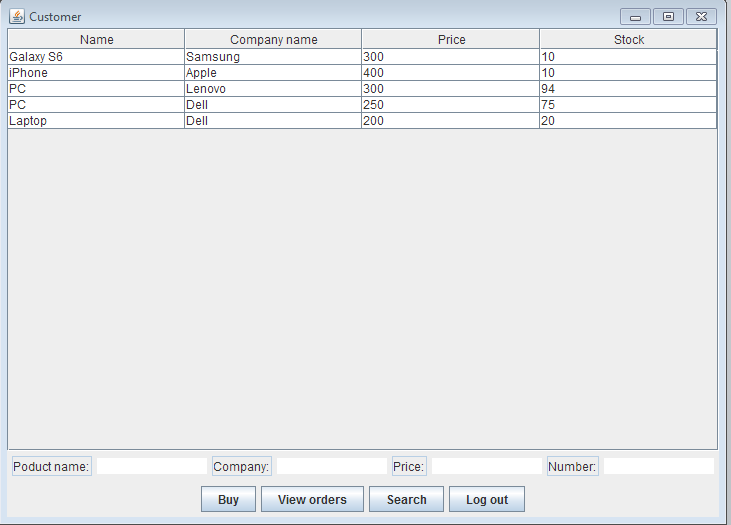
As a result of this run, on the screen it will open a window through which the user can enter its username and password , so that our application would know what interface to display. This window looks like this:



For the admin access we will have the below interface:



And for the customer will have the below interface:



Observe the five buttons on the bottom side of the window for admin and the four buttons on the bottom for the customer. The user can chose one of those operations depending on what he wants to do. The user can:

Add a new product to the “database” of products

Remove a new product to the Warehouse (a list of products)

View all the orders from the database

Search for a specific product

Log out

The Attributes of the Gui Class:

To achieve the desired GUI we need several types of attributes:

private static final long serialVersionUID = 211431534849779406L;

-for log in:

private static String ADMIN\_USERNAME="Admin";

private static String ADMIN\_PASSWORD="Admin";

private static String CUSTOMER\_USERNAME="Customer1";

private static String CUSTOMER\_PASSWORD="Customer1";

-for interface

private JTextArea username;

private JPasswordField password;

private JPanel mainView,userSection,passSection,loginSection;

private JTextField user,pass;

private JButton login;

The Constructor of the Gui Class: public Gui ()

It initializes all the attributes declared above; the window with the following elements will be constructed: frame, buttons, and text fields. Here there are the frame settings, such as the size, the visibility, title and some predefined operation such as:

this.setTitle("Program");

this.setSize(300,300);

this.setLocationRelativeTo(null);

initializeView();

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.setVisible(true);

In the customer and admin views the attributes and operations are a little bit different.

What is more, for every button (operation) there will be added an “ActionListener” which contains the instructions that are needed to be executed in the moment a certain operation button is clicked. For each button there is a different class declared inside the UserView class and CustomerView class which implements the “ActionListener” interface.

More information about the “ActionListener” implementation are in the chapter about the user interface where there are presented all the swing components used in the Gui constructor.

The Methods of the Gui,UserView and CustomerView Class:

Interfaces UserView and CustomerView contains methods for achieving operations when you press one of the two existing buttons in the interface.

In UserView

Methods for the five methods buttons:

* + The Add button:

It creates a product based on the fields that we entered. When the Add button is pressed by the user, the new product ,if valid is added to the warehouseNow the change becomes visible. The same is done for the other four panels.

For example: when adding a product ,the following operations are done:

* Creates a new product with the information inserted in the text fields;
* Adds the product to the existing tree of products in the warehouse
* Writes the information in the serialized file(so the next time the application is run to have the inserted information)
* Print the new table with the clients
* Add the table to the panel so it can be shown in the user interface

The same goes for delete and for the operations of adding or removing information from the other tree structure (OpDept and Warehouse).

* + The Remove Product button

It behaves like the Add, except that we don’t need to insert the quantity of the product.

* + View Orders

Has the role to print the table with the orders. The orders are sorted by a name which is a string with an auto-increment integer. Depending on the order, this table with contain the Order class attributes: client name, product name and quantity.

* + The logOut Button:

This button has the role to exit the application when it is pressed by the user and to give the possibility of relogging .

Client Class and the Product Class:

public class Customer {

public class Product implements Serializable, Comparable<Product> {

These classes are the representation of a client, respective a product which means that it contains as attributes information about a certain client: name, username and password or a product.

The methods of the Classes:

public Customer(String firstName, String lastName) {//Customer class

this.firstName = firstName;

this.lastName = lastName;

this.userName = "Customer";

this.password = "Customer1";

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

this.firstName = firstName;

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

this.lastName = lastName;

}

public String getUserName() {

return userName;

}

public String getPassword() {

return password;

}

//Product class

Here we override the hashcode and equals methods , so that to compare products based on their fields and not on their normal hashcode.

public Product(String name, String company, double price, int stock) {

this.name = name;

this.company = company;

setPrice(price);

setStock(stock);

}

public void setStock(int stock) {

this.stock = stock;

}

public int getStock() {

return stock;

}

public void setPrice(double price) {

this.price = price;

}

public double getPrice() {

return price;

}

public String getName() {

return name;

}

public String getCompany() {

return company;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime \* result + ((company == null) ? 0 : company.hashCode());

result = prime \* result + ((name == null) ? 0 : name.hashCode());

long temp;

temp = Double.doubleToLongBits(price);

result = prime \* result + (int) (temp ^ (temp >>> 32));

return result;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Product other = (Product) obj;

if (company == null) {

if (other.company != null)

return false;

} else if (!company.equals(other.company))

return false;

if (name == null) {

if (other.name != null)

return false;

} else if (!name.equals(other.name))

return false;

if (Double.doubleToLongBits(price) != Double.doubleToLongBits(other.price))

return false;

return true;

}

@Override

public int compareTo(Product p2) {

if (this.equals(p2)) {

return 0;

} else if (this.hashCode() > p2.hashCode()) {

return 1;

}

return -1;

}

We also implement getters and setters for some fields.

1. Warehouse Class: public class Warehouse {

Has the role of representing as a „database” structure the information about the products inserted in the file "products.ser". We can say that this class has an internal representation of the database, as a binary tree (Tree Set) and an external representation (for display in the user interface) as a JTable.

Also this class has a singleton pattern.

private Warehouse() {

inOut = new InputOutput();

warehouse = inOut.deserializeProducts();

}

public static Warehouse getInstance() {

if (instance == null) {

instance = new Warehouse();

}

return instance;

}

The Attributes of the Warehouse Class:

public static Warehouse instance;

private TreeSet<Product> warehouse;

private InputOutput inOut;

private Iterator<Product> iterator;//for iterating through the elements of the TreeSet

private static final int MAXIMUM\_STOCK = 1000;

private static final int MINIMUM\_STOCK = 100;

The Constructor of the Warehouse Class:

* Initialize a product's attributes (is called from the Gui class when adding a new product to the list).

The methods of the Warehouse Class:

Methods for working with the file are handled in the InputOutputClass.

Methods for working with the SetTree structure

public void addProduct(Product product)

public void removeProduct(Product product)

public int getTotalNumberOfProducts()

public Product[] getAllProducts()

public int getProductStock(Product product)

public void sellProduct(Product product)

public boolean searchForProduct(Product product)

Remark:

The class OpDept issimilar with the Warehouse class with the exception that the methods are adapted to the specific object of each of them.

**3.6 *Packages and Interfaces***

A Java package is a mechanism for organizing Java [classes](http://en.wikipedia.org/wiki/Class_%28computer_science%29) into [namespaces](http://en.wikipedia.org/wiki/Namespace_%28computer_science%29). Java packages can be stored in compressed files called [JAR files](http://en.wikipedia.org/wiki/JAR_file), allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality. A package provides a unique namespace for the types it contains. Classes in the same package can access each other's package-access members.

A package allows a developer to group classes (and interfaces) together. These classes will all be related in some way – they might all have to do with a specific application or perform a specific set of tasks.

For this application the following packages are imported, each of them having a certain role for the proper working of the application. We import them in the Gui Class (most of them relate to the user interface properties):

* import java.awt: Contains all of the classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components.
  + java.awt.BorderLayout: A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center.
  + java.awt.Dimension: This encapsulates the width and height of a component (in integer precision) in a single object.
* import java.awt.event
  + java.awt.event.ActionEvent;
  + java.awt.event.ActionListener;
* import javax.swing: Typical Swing applications do processing in response to an event generated from a user gesture. For example, clicking on a JButton notifies all ActionListeners added to the JButton. That’s why we use this package for creating the user interface Gui.
  + javax.swing.JButton;
  + JFrame; JLabel; JPanel; JTabel; JScrollPane

***3.7 Serializable Interface***

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable if this is not the case. The error will be detected at runtime.

During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

When traversing a graph, an object may be encountered that does not support the Serializable interface. In this case the NotSerializableException will be thrown and will identify the class of the non-serializable object.

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures.

In this case, we use serializable to read and write objects in a binary file so that the information inserted or removed by the user to be memorized from one run to another.

* 1. ***Comparable Interface***

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class C is said to be *consistent with equals* if and only if (e1.compareTo((Object)e2) == 0) has the same boolean value as e1.equals((Object)e2) for every e1 and e2 of class C. Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

We use this property for the trees declared in the Warehouse and OpDept Classes. This means that when we add a product to the tree, it will be put in its right position. We order the products in a lexicographic order, depending on their names.

For the orders in the OpDept class the rule is that they are ordered depending on the id, which is an “auto-increment” variable. The orders table will look like this:

***3.9 User Interface***

When running the application, the window will open and it will provide to the user the possibility of giving inputs and choosing what operation he likes to be executed. This window is constructed in the Gui class using some predefined classes and instructions.

The user interface is based on the properties of the above mentioned packages. All the objects we need are declared as attributes of the Gui class and they are initialized in the constructor of this class. For executing the operation commanded by the user we use the predefined functions from the ”ActionListener” interface.

* The ActionListener functions

The listener interface is for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked. In this case the only events that occur are when the user clicks on one of the operation buttons from the graphical interface.

We take as an example the instructions that need to be executed when clicking on the “Add” button after we filled in the fields for the product . For this we need another class which implements the ActionListener:

This class will contain the method which executes all the instruction needed in order to fulfill the selected operation. The structure of this method looks like in the next section :

public void actionPerformed (ActionEvent e)

if (event.getSource() == logOut){

***} else if (event.getSource() == add) {***

***} else if (event.getSource() == remove) {***

***} else if (event.getSource() == viewOrders) {***

***} else if (event.getSource() == search) {***

***} else {***

***JOptionPane.showMessageDialog(this, "The product doesn't exist in the warehouse.", "INFORMATION",***

***JOptionPane.INFORMATION\_MESSAGE);***

***}***

***4. Using and testing the application***

In order to use the application open Homework3.JAR. This will open a window which generates the Gui class. Thus the user can select one of the 4 options from the left side of the application and after that he can insert or remove objects from the “database”.

***5.Conclusions***

Achieving such a program may be hard both in terms of algorithms, graphical structure. The best is to represent the orders and products as a tree type structure because this kind of structure makes it easier for some operations to be done: add, remove or search for an element from the structure.

For a better performance there should be implemented all cases where exceptions can occur and the application stops working due to an error made ​​by the user. Also, the application can be changed so that it could deal with more operations and more clients . Another thing that could be improved is the display so that it would be more elegant.

***6.References***

[*http://users.utcluj.ro/~jim/OOPE/*](http://users.utcluj.ro/~jim/OOPE/)

[*http://docs.oracle.com/javase/7/docs/api/overview-summary.html*](http://docs.oracle.com/javase/7/docs/api/overview-summary.html)

[*http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html*](http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html)

[*http://stackoverflow.com/*](http://stackoverflow.com/)