**Assignment 3**

Programing techniques

**Order Management**

*Student:Gombos Kriszta*

*Group:* 30424

Content

*1. The purpose of the topic .............................................. .............1*

*2. Problem analysis, modeling, scenarios, cases of use .................... 1*

*2.1. Problem analysis ................................................ ................ 1*

*2.2. Modeling ................................................. ............................. 2*

*2.3. Scenarios ................................................. ........................... 5*

*3. Design ............................................... ................................................. 5*

*3.1. UML diagram ................................................ ......................... 6*

*3.2. Designing classes ................................................ .................. 7*

*3.3. Algorithms ................................................. .......................... 10*

*3.4. User Interface ................................................ ................... 11*

*4. Implementation and testing ............................................. ................ 12*

*5. Results, Conclusions and Further Developments ...................................... 14*

*6. Bibliography ................................... ..........................................14*

1. **Purpose of the subject**

The project requirement is: "We consider a OrderManagement application to process customer orders. The application will use (minimum) the following classes: Order, OPDept (Order Processing Department), Customer, Product, and Warehouse. The OPDept and Warehouse classes will use a Binary Search Tree for storing commands and products. "

a) Analyzes the field of application, determines the structure and behavior of its classes and identifies the cases of use

b) Generates case usage graphs, UML diagram, two sequence diagrams, and an activity diagram

c) Implements the application classes. Use Javadoc to document the classes

d) Implements a system of utility programs for reporting: substock, superstock, totals, filters, etc.

e) Write the appropriate test drivers.

The objective of the theme is to implement a graphical interface application to be used for order management. The application allows viewing, modifying, deleting and adding products, orders, and clients from the perspective of an administrator. Also, for storing products, orders and clients, we used a database with a structure of tables similar to the class structure implemented in Java.

**2. Problem analysis, modeling, scenarios, cases of use**

2.1. Problem analysis

To implement the application we chose the Model-View-Controller (MVC) design pattern. The Model-View-Controller design concept, in the specialized and used MVC documentation, emerged as a need to transpose traditional data management methods into the "virtual" environment, more specifically to ease user mode of operation respecting the same basic principles and differentiating through the tools used. However, the basic principles of the concept are easy to understand, but the details are complex enough to launch many debates and meet contradictory implementations.

The principle underpinning the Model-View-Controller concept is the sharing of responsibilities. In a based application designed to meet this concept, the model part will only work with the state of the application and its logic, will not count how this state is or will be represented to the user, or how it interacts with the application. Similarly, the "view" part is concerned only with creating the user interface based on the data and, in particular, the changes of its status received from the "model". It does not matter the logic of the application or how the "input" process takes place, but only the correct representation of the current state of the model. Finally, the "controller" deals with the translation of user actions into "updates" to the model, and it is not important what the "update" model will use.

The model represents the hard-programming part, the logic of the application. He is responsible for actions such as data operations and database connections. The model thus represents a collection of classes that will perform certain functions, unrelated to the actions of the user. They will be the basis for any function that your application will need to fulfill. Thus, in the model package we included the following classes: Product, Customer, Warehouse, Order, OPDept and DBConnection.

The view looks at the data display. Once the functions are executed by the model, the view is given the results, and it will display them to the user. In this package we included some classes expanding JFrame and JPanel supercars. They display the data according to the user's option. The classes included in the view package are: MainFrame, ProductsPanel, CustomersPanel, OrdersPanel, CartPanel and OrderFrame.

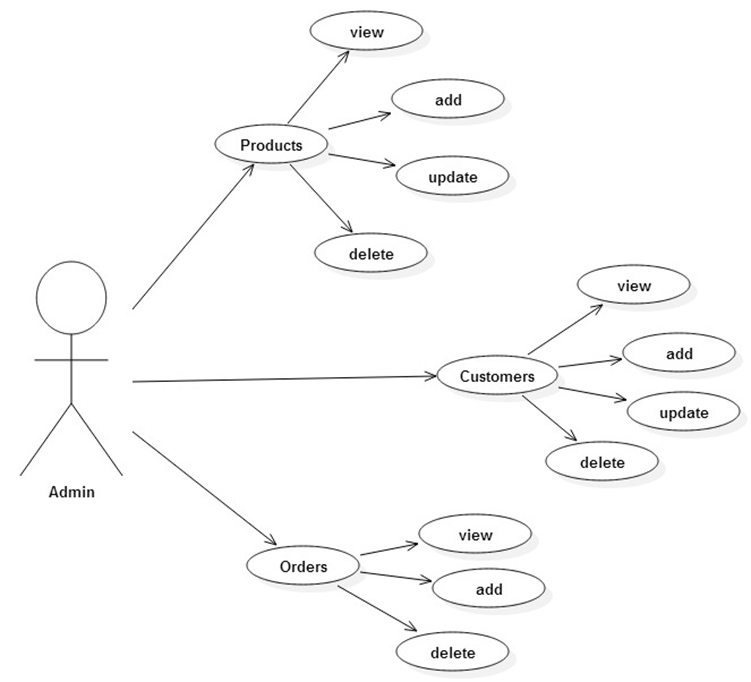
The controller is the link between the model and the view, between the logic of the application and the actions of the user. Depending on the user's requirements, the controller will perform a specific function specifically defined for the user application section. The function will use the model to process the data and send the results to the view, which will update the data, according to the actions of the user. In this package we included the classes: ClickListener, Controller and TableCellListener.

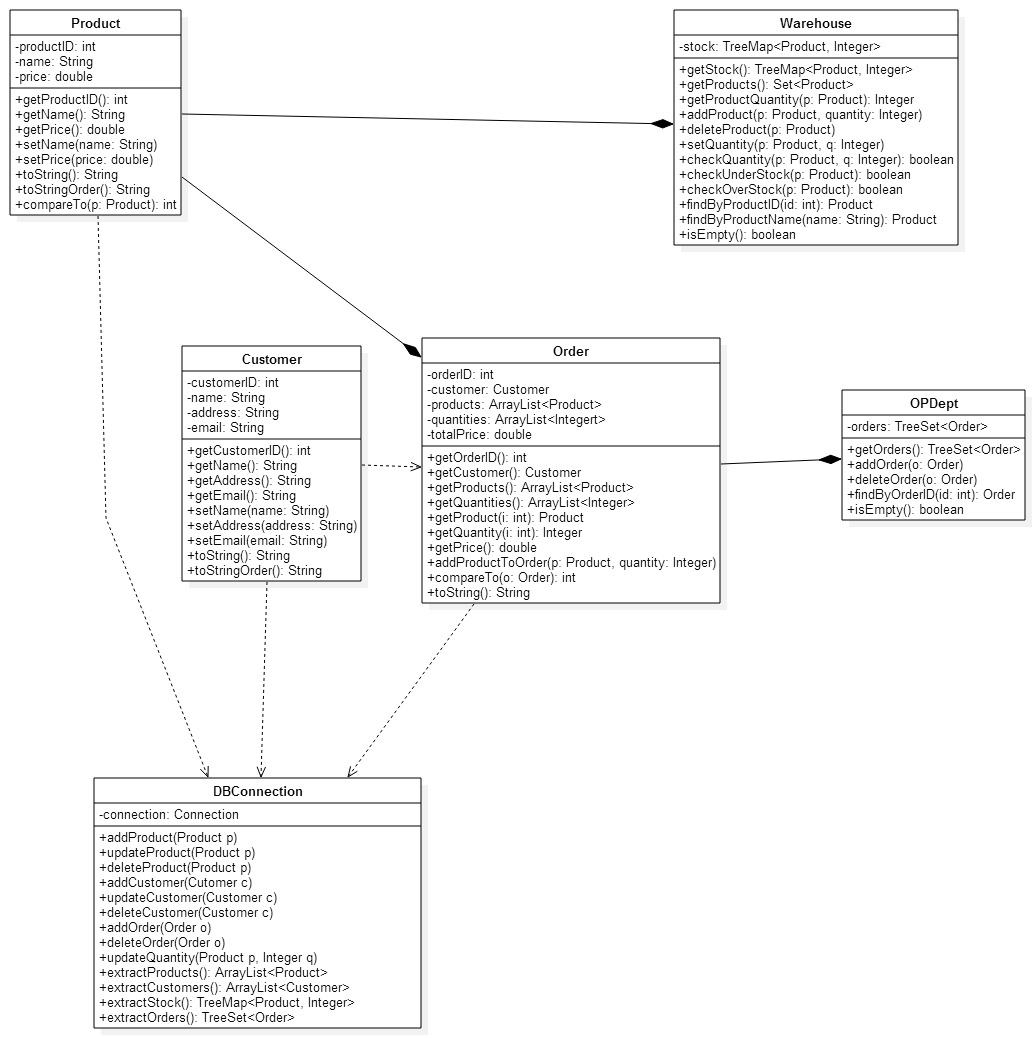
2.2. Modeling

When implementing the Warehouse and OPDept classes, we studied Java structures to create a Binary Search Tree. So, we used TreeSet <Order> to implement a Binary Search Tree Array, which is used in the OPDept class. To order orders in the search tree, we used the compareTo (Order o) method implemented in the Order class to sort the commands according to their id.

In order to create the Warehouse class, it was necessary to specify a pair of elements of the search tree, namely the Product object and its quantity (Integer). To do this, we chose to use a TreeMap <Product, Integer> structure which has as its key the Product object, and its quantity refers to the second position of the structure (Integer). Also in this case it was necessary to use a comparator for the introduction of the products into the search binary tree, which is implemented by the compareTo (Product p) method of the product.2.3 class. ScenariosIf it deserves particular attention is the addition of a product to an order if there is not enough quantity of the product. So, when adding a product to your shopping cart, first check if there is enough of that product and then add it to your shopping cart. Also, the data of any changes made to the database or to the object structures implemented in Java are then checked. DesigningTo implement the project, we used the Eclipse development environment and the Java programming language. The problem has been divided into classes that will be presented below. The steps to solve the problem were to create the UML diagram underlying the project, and then to build the required classes and methods.

For a better understanding of the class structure we chose to divide them into several packages, these being: model, view, controller and main. Because the class structure includes a total of 16 classes, we have made the UML diagram first for the classes in the model, along with the relationships between them, and then for the rest of the classes, specifying the relationships between the classes in the model and these. These charts can be seen next to the use case diagram:



**3.1 UML** **Diagramm**

Package *model*

3.2. Class Designing

Splitting classes into packages is as follows:

Model Package:

- Product

- Customer

- Warehouse

- Order

- OPDept

- DBConnection

View Package:

- mainframe

- ProductsPanel

- CustomersPanel

- OrdersPanel

- CartPanel

- OrderFrame

Controller Package:

- Controller

- ClickListener

- TableCellListener

Main Package:

- Main

Next, void details each of these classes, specifying their functionality.

3.2.1. The model package

3.2.1.1. Product class

The Product class shapes the features and functionality of a product. The attributes included the unique ID that identifies each product, the name of the product, and its price. In the class constructor, these three parameters are received, and the identification of these parameters with the attributes of the class is made. This class implements the Comparable <Product> interface, because for the construction of the binary search tree, it was necessary to order the products (we chose the ordering according to the product id), using the compareTo method.

The methods of this class are as follows:

- getProductID () - the method that returns the unique product id to be accessed by other classes of the application

- getName () - method that returns a string of characters represented by the name of the referenced product to be accessed in other classes of the application

- getPrice () - the method by which a double value is returned, represented by the price of the product

- setName (String name) - the method called when it is desired to change the name of a product, and the product-specific attribute is set to the value referenced by the parameter sent to the method call

- set Price (double price) - the method called when it is desired to change the price of a product to the value referenced by the parameter

- toString () - the method by which a character string is returned representing the attributes of a product in a form that can be displayed in the interface.

- toStringOrder () - similar to the toString () method with the difference that this display is a little more detailed and is used to display a client's order.

- compareTo (Product p) - the method declared in the Comparable <Product> interface and implemented in such a way that a negative number returns if the product referenced by this has a unique id lower than the given product as a parameter and a positive number otherwise. This method is used for ordering products in the binary search tree (TreeMap)

3.2.1.2. Customer Class

The Customer Class shapes the features and actions of each created customer. The attributes include the unique ID that identifies the client's name, address, and email to contact the customer with the order details. By constructor, these four attributes are initialized to the values ​​referenced as parameters.

The methods specific to this class are as follows:

- getCustomerID () - the method by which it returns the unique id identifying each client so that it can be accessed in the rest of the classes

- getName () - the method by which a string of characters is returned by the customer's name so that it can be accessed by other classes

- getAddress () - the method by which a character string represented by the client address is returned, so that it can be accessed by other classes

- getEmail () - the method by which a string of characters is returned by the customer's email so that it can be accessed by other classes

- setName (String name) - the method by which the name of the client is set to the value referenced as a parameter when an edit occurs

- setAddress (String address) - the method by which the client's address is set to the value referenced as a parameter when a change occurs

- setEmail (String email) - the method by which the client's email is set to the value referenced as a parameter, when a change occurs

- toString () - the method by which a string is returned to display a client in the interface. The attributes shown are the name and address of the client

- toStringOrder () - the method by which it returns a string to display the client in the detailed order display.

3.2.1.3. Order Class

The Order class shapes the properties and functionality of an order. As attributes, we included the unique id that identifies each command, the client who made the command, the products from which the command is composed (included in an ArrayList <Product>), the ordered quantities included in an ArrayList <Integer> and the total price of the order. In the class constructor, the first four parameters are received (without the total price), and the identification of these parameters with the attributes of the class and the calculation of the total price of the order according to the price of the products and the quantities ordered are made. Also, the class has a second constructor that only receives the order id and the customer, and the products and quantities are added after the order. This class implements the Comparable <Order> interface, because for ordering the binary search tree it was necessary to sort the orders (I chose ordering according to the order ID), using the compareTo method.

The methods of these classes are as follows:

"Get" methods by which the specific attribute is returned so that it can be accessed in other classes. These are: getOrderID (), getCustomer (), getProducts (), getQuantities (), getProduct (int i), getQuantity (int i), getTotalPrice ().

The addProductToOrder method (Product p, Integer q) adds a product to the product list of each order, and its quantity to the list of quantities. Also, the total order price is increased to the old value plus the new value added (quantity \* unit price)

The compareTo method, declared in the Comparable <Order> interface, is implemented in such a way that a negative number returns if the command referenced by this has a unique lower id than the command date as a parameter and a positive number otherwise. This method is used to sort orders in the binary search tree (TreeSet)

The toString () method returns a string so that all the available command data is displayed. This is called from a separate frame when the user wants to view all the details of the order.

***3.2.1.4Warehouse Class***

The Warehouse class is the object containing the products and quantities available. To implement this class we used a search binary tree, defined as a TreeMap <Product, Integer>, which is the only attribute of this class.

As methods we implemented the following:

"Get" methods used to return certain parameters of this class. Among these, we included the following: getStock (), getProducts (), getProductQuantity ().

The setQuantity (Product p) method is used when changing the quantity of a particular product.

The checkQuantity (Product p, Integer q) checks if there is enough quantity in a product if it is wanted to buy it.

CheckUnderStock (Product p) and checkOverStock (Product p) check if the stock in a particular product is too small or too large. As thresholds we considered that all products with a quantity of less than 100 pieces are considered "understock", and over 200 pieces are considered "overstock".

The decreaseQuantities method (ArrayList <Product> p, ArrayList <Integer> q) decreases existing quantities in the range for the entire list of products given as a parameter. The final quantity is determined as the difference between the quantity before the method call and the quantity that is desired to be low, given as a parameter in the quantity list. The method is used in case of an order, in order to decrease the quantity of the entire list of purchased products.

The findByProductID (int id) method returns the Product object that has the specified id as a parameter. If no product has the specified id, the null value is returned.

The isEmpty () method returns the false value if there is at least one product and value in the store if the warehouse contains no product.

***3.2.1.5 Class OPDept***

The OPDept class assumes the command processing department and has the functionality and maintenance of command-specific operations. To implement this we used TreeSet <Order>, which is a predefined Java structure for implementing a Binary Search Tree. In the class constructor, a TreeSet <Order> is specified as a parameter, representing all orders made at the building of the department (these are taken from the database)

The methods implemented in this class are as follows:

- getOrders () - the method by which all commands existing in the department are returned

- addOrder (Order o) - method by which the date command as parameter is added to the command processing department

- deleteOrder (Order o) - the method by which the date command is deleted as a parameter in the department

- findByOrderID (int id) - the method by which the command is searched that has the id identical to the id given as a parameter, and if it exists returns the command and otherwise returns null

- isEmpty () - returns true if there is no order in the department and false otherwise.

**3.2.1.6. DBConnection class**

The DBConnection class connects to the application's database. For this we have declared statically the components to which it connects (URL, User, Password), and the attribute of this class is the object of Connection that makes the connection to the database. In the class builder the connection to the database is made.

The methods implemented in this class refer to operations that enter, modify, delete or extract data from the database. These are: insertProduct (Product p), updateProduct (Product p), deleteProduct (Product p), extractProducts (), insertOrder (Order o), deleteOrder (Order o), extractOrders (), insertCustomer c), deleteCustomer (Customer c), extractCustomers (), extractStock ().

**3.2.2. The controller package**

3.2.2.1. Class Controller

Controller class performs application control. As attributes we have the model (represented by the connection to the database, the warehouse, the command department, and the interface referenced by the MainFrame object.) The constructor of the class receives as parameters the connection to the data base and the view (MainFrame), followed by the rest the objects are created based on the connection to the database. Also in the constructor is added ClickListener for the buttons in the interface classes and for the tables in them.

The methods implemented in this class refer to the actions available to the user. These are the following:

- getView () - returns the interface to be accessed in other classes of the application (used in the ClickListener class

- addProduct () - adds a product both to the database and to the Warehouse object as defined by the controller. The attributes of the added product are retrieved from the interface using the getPanel class available. After the change is made, the interface is updated to see the changes.

- deleteProduct () - method that deletes a product from both the database and the warehouse object. The product is retrieved from the interface (via a JComboBox)

- updateProducts () - the method by which user changes are made both in the database and in the warehouse structure. Check that the changes made by it are valid, and in this case update the interface with the new values

- the same methods performed for the Customer

- addToOrder () - the method by which the product selected by the user and the desired quantity is added and added to the product basket to be purchased

- addOrder () - the method by which an order is added both in the database and in the order department depending on the selected customer and the products added previously.

- removeProducts () - the method by which the product basket is empty. This is called when you click the "Remove All" button in the product basket.

3.2.2.2. ClickListener class

The class by which the actions of the buttons are treated. It implements the MouseListener interface and its specific methods. I chose to specify the actions only when clicked (mouseClicked) and it checks which button has been pressed and, depending on it, calls the specific methods in the controller (this is given as a parameter in the constructor)

3.2.2.3. The TableCellListener class

This class is used to treat the JTable components. In this way, you can access the row, column, new value and old value of the modified cell, for them to be checked and modified in the database and in the Java structure (ArrayList for clients, Warehouse for products and OPDept for commands)

3.2.3. The viewer package

This package contains the interface of the application and will be explained in detail at the User Interface section

3.2.4. The main package

3.2.4.1. The Main class - contains the main function main (String [] args) running at the start of the application. This creates a new MainFrame object, as well as the DBConnection object and the Controller object that receives the view and model parameters (MainFrame and DBConnection).

3.3. User Interface

The user interface consists of the interaction between the user and the order management system. The interface consists of a total of 6 classes, these being: 4 JPanels and 2 JFrames.

3.3.1. MainFrame class

 This is the main window that contains the rest of the interface components. This creates the menu bar required to change between the panels and the panels needed to use the application: ProductsPanel, CustomersPanel, OrdersPanel, CartPanel. These are set to be viewed only when they are viewed. In this class are implemented the methods that display one of these panels, called methods from the controller package, depending on the user's options. There are also implemented methods for returning the 4 panels to be accessed from the Controller class.

3.3.2. ProductPanel class

This class displays everything related to viewing, modifying, adding or deleting a product. The products are set by the controller when it is created (in the builder) and are displayed in a JTable. Also, some JTextFields are required to filter and add a product. In addition, we have a JComboBox that selects the product to be deleted and the buttons that confirm these actions.

3.3.3. CustomersPanel class

It is similar to the ProductsPanel class with the difference that this refers to clients extracted from the database.

3.3.4. OrdersPanel class

Assemble the two classes with the difference that the displayed orders are displayed. Also, for detailed order viewing, when a user wants this, an OrderFrame object is created that provides more details about the command in a separate frame.

3.3.5. CartPanel class

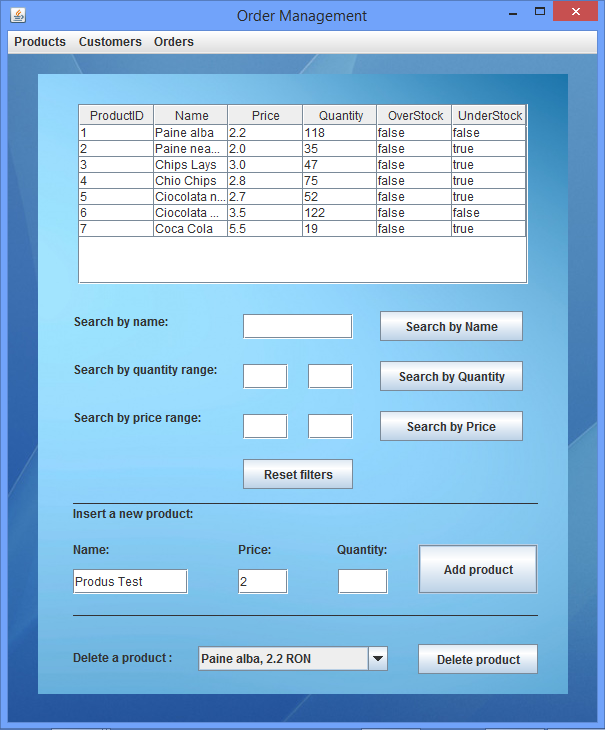
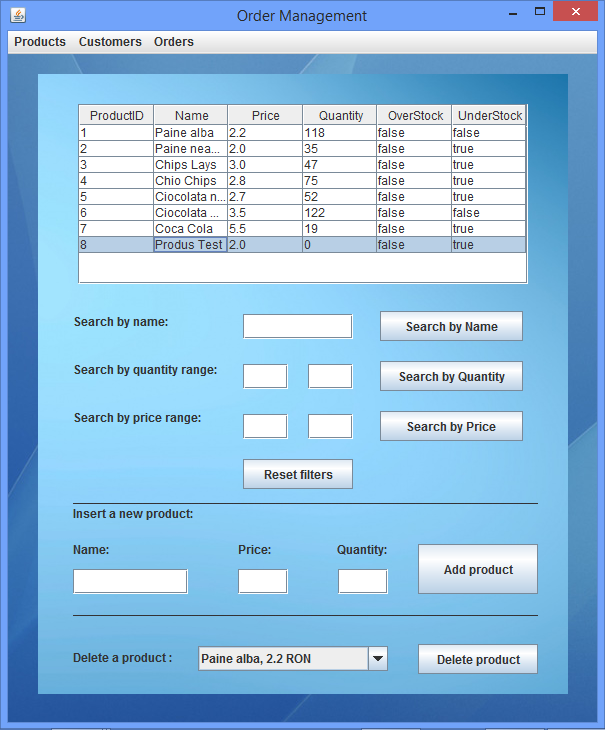
It is used to view the products and quantities to be added to an order

3.3.6. OrderFrame class

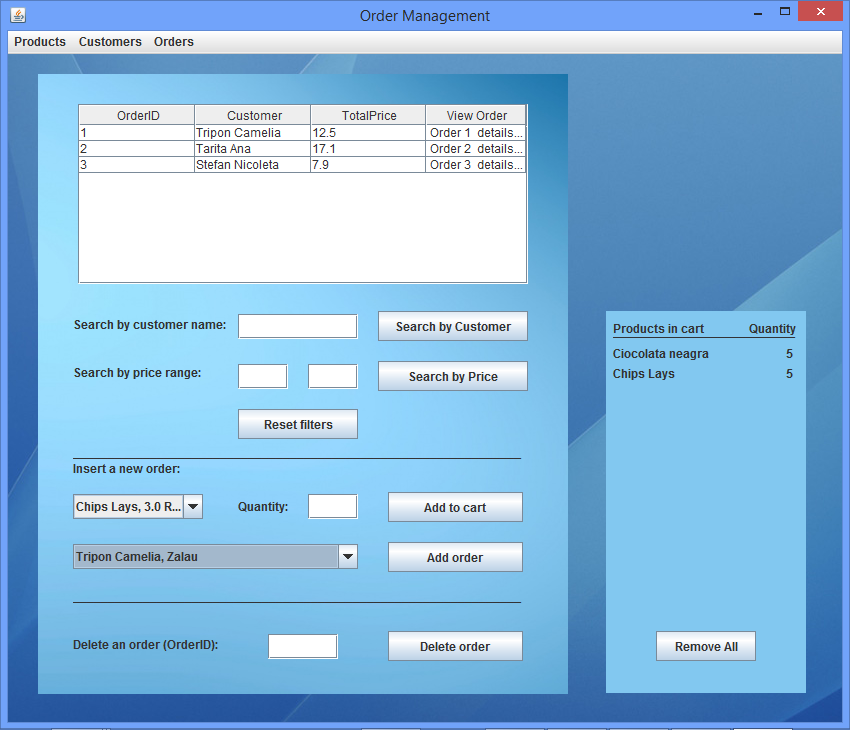
In this class you can view a detailed display of a command received as a parameter when building the object.

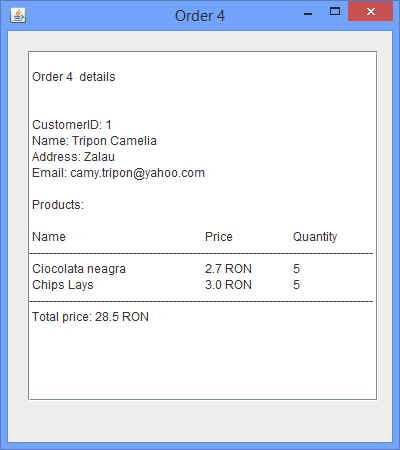
1. **Implementing and testing**

For testing I chose to put some pictures









**5. Results, Conclusions and Further Developments**

As a conclusion, this project has helped me to consolidate my object-oriented programming skills acquired in the first semester and to organize my work based on POO paradigms. At the same time, the way I structured the project and created the classes and methods helped me to make the code more efficient in terms of length and ease of understanding, vital for complex programs requiring more classes or involving a larger number of programmers.

Subsequent developments include the following: Improving the graphical interface, including several user options such as deleting commodities or modifying them, changing the order after it has been introduced, including multiple attributes for table data, and splitting program in two sections: administrator: who can access all the operations offered by the application and the client: who can view the available products and can submit new orders.

**6. References**

1. Object Oriented Programming Courses - Marius Joldos

2. www.stackoverflow.com

3. http://docs.oracle.com/javase/tutorial/

4. Thinking In Java - Bruce Eckel