Student: Ardelean Eugen-Richard

Group: 30423

Programming Techniques

Homework 3

Contents:

1. Problem Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2. Analyzing the problem, modelling, scenarios, use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3. Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
4. Implementation and testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
5. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
6. Conclusion and future developments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
7. Bibliography . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .10
8. **Problem Specification**

Consider an order management application for processing customer orders for a warehouse.

Relational databases are used to store the products, the clients and the orders. Furthermore, the application uses (minimally) the following classes:

1. Domain specific classes: Order, Customer and Product

2. Business Logic (warehouse-specific processing) classes: OrderProcessing, WarehouseAdministration, ClientAdministration

3. Presentation classes: GUI related classes

4. Data access classes: Database access related classes

Other classes and packages can be added to implement the full functionality of the application.

Requirements:

a. Analyze the proposed application, determine the structure and behavior of its classes and draw an extended UML class diagram.

b. Design, implement and test the application classes. Use javadoc for documenting classes.

c. Define, design and implement a system of utility programs (examples: reports for under-stock, totals, filters, etc.).

d. Design and implement a comprehensive demo driver for the order management application.

1. **Analyzing the problem, modelling, scenarios, use cases** 
   1. **Analyzing the problem**

This application should be able to allow the user to insert (add a new customer or product with all the fields), delete (remove a customer or product with all the information about it) and update (change any of fields of the customer or product) the data from the database. The products, orders and customers are stored in a relational MySQL database. The user should also be able to add orders by choosing the customer, the product and the quantity of the product. The application should be able to generate text or pdf files, in this case text files, with each order given.

* 1. **Modelling**

Modelling is the activity to make an abstract concept easier to understand by finding its main characteristics and defining some laws which make the given phenomenon quantifiable. This process also includes the decomposition of a complex problem into smaller and simpler problems which will be easier to implement.

The modelling of the problem was done in the indications, by separating the problem into packages. Each package having its own part of the whole problem.

* 1. **Scenarios**

A success scenario looks like this:

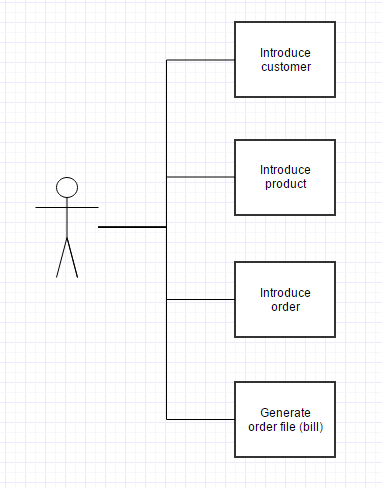
* Introduce an order by customer x with the payment method p.
* Introduce one or more products (keeping in mind that quantity has to be smaller than the stock).
* The order will appear in the order table.

While a failure scenario looks like this:

* Introduce an order by pressing a button and introducing the id of a customer and the payment method.
* Introduce a product id and the quantity, and press the add product button.
* Quantity introduced is bigger than the available stock.
* Generate an appropriate error.
  1. **Use Cases**

Use case diagrams are usually referred to as behavior diagrams used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors).

The use case diagram of the application should look something like this:



1. **Design (design decisions, UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface)**
   1. **Design decisions**

There have been many decisions about how to create the application along the way because of the freedom of choice of how to implement it. For example, when choosing how to implement the database, beside the 3 necessary tables (customers, products and orders), I chose to add a fourth table, order details, that basically links the orders and products tables so that they can have a many to many relationship.

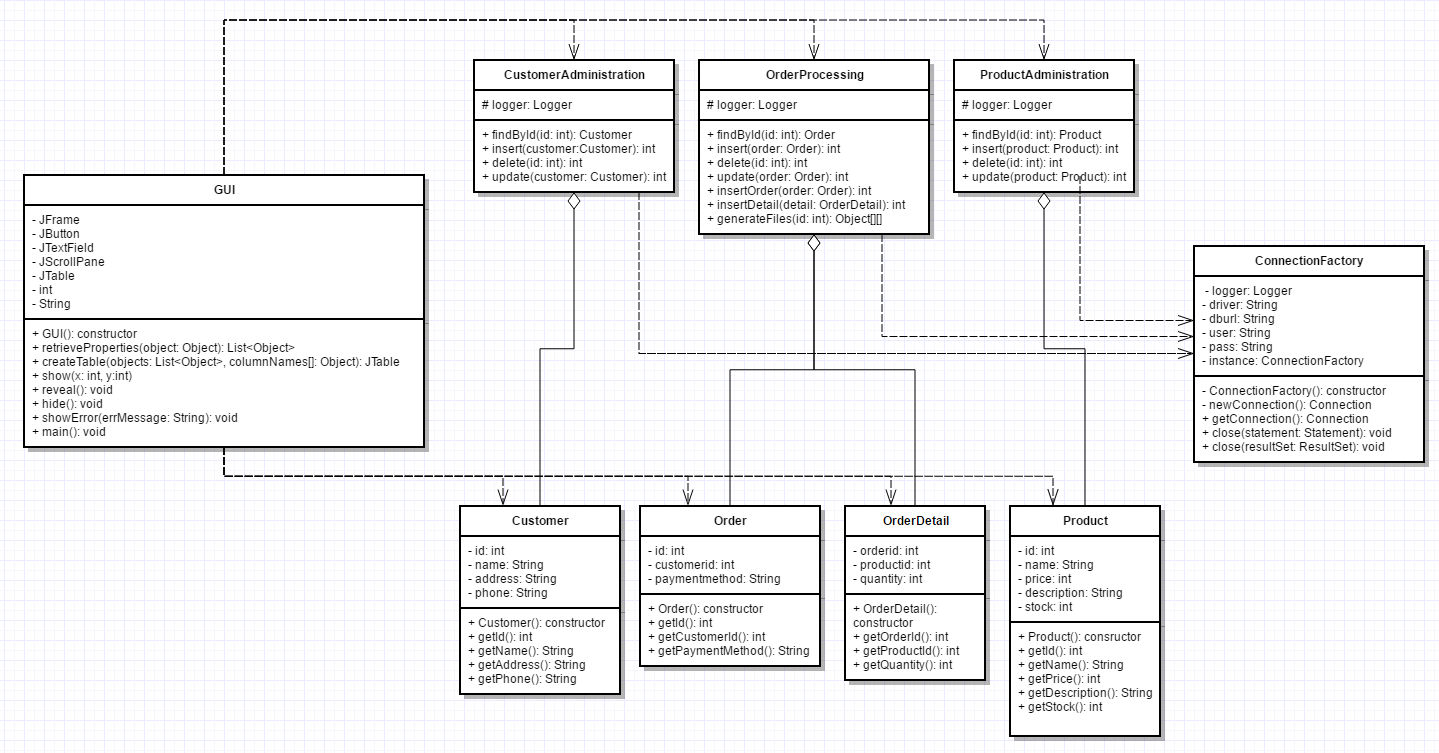
Another decision choice was the implementation of adding new orders to the database. I chose to do this by creating a button for adding orders that will read from the text fields the customer and the payment method. After pressing this button, all of these will become invisible, in their place there will be a button for adding a product (to the order) and the fields of the product, id and quantity. There will also be an add a new order button that the user can use to finish adding products to the current order and create a new one.

For generating the text file, I used a buffered write to generate the file and then write in it the data from the table. Getting the data is done using a method that uses a query to get the information needed and then puts the information in an object type matrix.

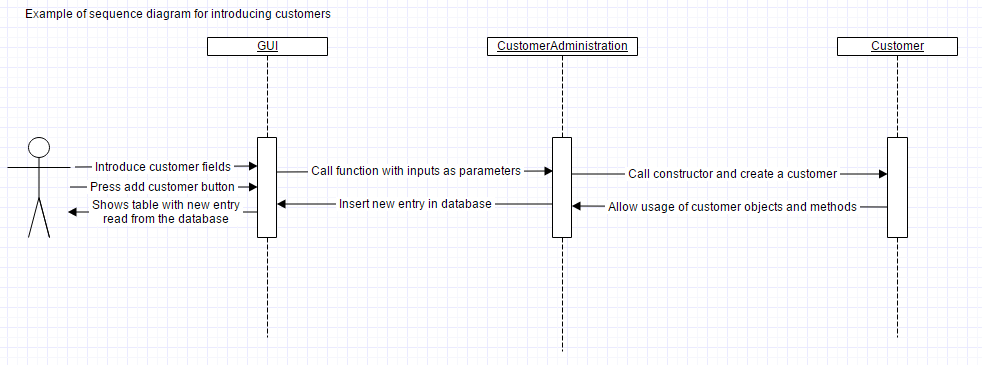
* 1. **UML diagrams**

We prepare UML diagrams to understand a system in better and simple way. A single diagram is not enough to cover all aspects of the system. So, UML defines various kinds of diagrams to cover most of the aspects of a system.

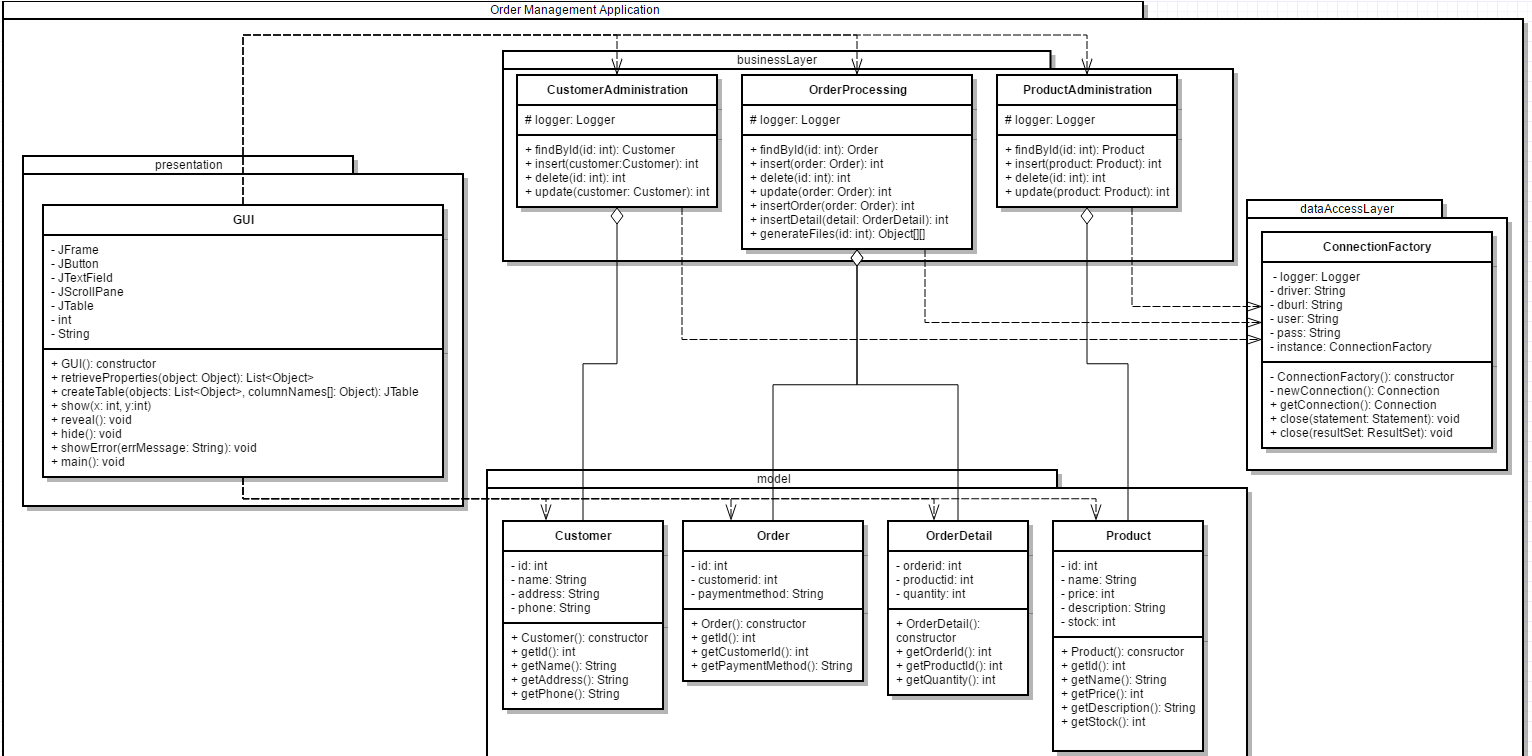
Class diagrams are the most common diagrams used in UML. Class diagram consists of classes, interfaces, associations and collaboration. Class diagrams basically represent the object-oriented view of a system which is static in nature.



A sequence diagram is a diagram that shows the interaction between the different objects of the application. For the order management application, it would look something like this:



A package diagram in the Unified Modeling Language depicts the dependencies between the packages that make up a model.



* 1. **Data Structures**

I used the ordered collection List for the reflection technique in the user interface class, to put the values introduced in the JTable. Also the interface List is used in the user interface class to remember the elements from the tables in the database and then put them in the JTable.

* 1. **Class Design**

For each table in the database, we need to have a class with the columns from the table as fields in the constructor. For this we have the classes: Customer, Product, Order and OrderDetail. Customer represents the customer table from the database it has the fields: id, name, address and phone. Product has the fields: id, name, price, description and stock. The order class has the fields: id, customerid and payment method. The order details class is an auxiliary created to simulate the many to many between Product and Order so that a product can be part of many orders and an order can have many products.

The administration classes implement the ability to modify the database with the application with the methods: insert, delete and update. For the case of the order class it is also used to update the value of the stock when an order is introduced.

For simplicity, instead of having the usual MVC (model, view, controller) implementation of the graphical user interface, I chose to implement it all in one class called GUI.

The ConnectionFactory class makes the connection between the database and the application.

* 1. **Interfaces**

For the implementation of this project, there were no interfaces used as there wasn’t any need to implement them. All classes have different methods, even if they have the same name, the way they operate or / and the parameters are different.

* 1. **Relationships**

Association represents the fact that objects of one thing are connected to objects of another thing, but there is no ownership.

Dependency represents the fact that one class uses operations from another class or it uses variables or arguments typed by the other class.

Aggregation and composition are particular cases of association, in aggregation all objects have their own lifecycle, but there exists ownership (that’s why it is called a “has - a” relationship), and if we destroy the parent object, the child still exists. In the case of composition (a specialized form of aggregation), when the parent object is destroyed so it the child object.

Between the business layer and the model there are many relationships. Each element from the business layer “has - a” associated element in the model. For order processing we have the class order, for warehouse administration we have the class product and for customer administration we have class customer. Between all of these, we have aggregations relationships.

There are many dependencies between these classes, because the classes are very tightly linked. The connection factory class has dependencies with most of the other classes because any method that operates on the database has to have an object of that class. Also there are lots of dependencies between the graphical user interface class and most of the other classes, because it uses objects from those classes to store values that we put in the tables.

* 1. **Packages**

As required, the application contains the four packages: dataAccessLayer, businessLayer, model and presentation. Each package represents a part of the application. The data access layer represents the connection between the application and the database, the business layer represents the administrative part of the application, here you can add, remove or change the data from the database by using the appropriately-named methods: insert, delete and update. The modification of the database can be done on any of the tables. The model contains a model of each of the tables, with the columns as fields, so in this package we have a class for each table: customer, order, product and order detail. And the presentation package has the graphical interface, which will be explained in depth in section 3.9.

* 1. **Algorithms**

Java Reflection makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using reflection.

A good example of how java reflection techniques work is given in the indications of this project, it looks like this:

**public** **static** **void** retrieveProperties(Object object) // unknown object

{

**for** (Field field : object.getClass().getDeclaredFields()) // field takes the form of one of the fields of the object

{

field.setAccessible(**true**); // set modifier to public

Object value;

**try**

{

value = field.get(object); // value takes the value in the field "field" from object "object"

System.***out***.println(field.getName() + "=" + value); // print the name of the field and its value

}

**catch** (IllegalArgumentException e)

{

e.printStackTrace();

}

**catch** (IllegalAccessException e)

{

e.printStackTrace();

}

}

System.***out***.println();

}

* 1. **User Interface**

The User Interface was built using Java Swing elements like: buttons, frames, labels, text fields, scroll panes and tables. The interface can be called user-friendly because it does not require knowledge in computer science to be able to use it. The inputs of each operation (insertion, deletion or update) will be introduced in text fields that have the initial value as “Type Field” so the user knows what kind of information to type. The operation itself will be done by pressing an appropriately named button.

There were many methods implemented for the esthetics of the graphical user interface, like methods used for hiding some buttons or text fields or making them appear when clicking certain buttons, this is not used only for esthetics but also for making the user not be able to add empty orders by hiding the “add order” button in the action listener so that when the user clicks it, it disappears.

1. **Implementation and testing** 
   1. **Implementation**

The application has been implemented in Java Programming language, using Eclipse. For the GUI, all the buttons, textfields and other components were added by code without any “drag and drop” plugins. Listeners are placed for specific component to catch events (usually a simple button press), and respond accordingly.

All the implemented classes, including their methods and attributes were documented with comments to be easier to understand.

One of the most difficult parts to understand and to implement was using the reflection techniques properly, to be able to create abstract methods and classes, to work with fields of objects that you don’t know the type of or the number of. All of this information is not known until compilation.

An example of how the reflection technique works is the following method used for creating a jtable:

// method for creating a JTable

JTable createTable(List<Object> objects, Object columnNames[])

{

JTable table = **null**;

List<Object> properties = *retrieveProperties*(objects.get(0)); // properties is the list of fields of an object from the list of objects

Object rowData[][] = **new** Object[objects.size()][properties.size()]; // matrix rowData has object size rows and properties size columns

**int** i = 0; // used as row iterator in the matrix

**for**(Object obj: objects)

{

properties = *retrieveProperties*(objects.get(i)); //gets the value of the fields of the next object

**int** j = 0; // used as column iterator in the matrix

**for**(Object prop: properties)

rowData[i][j++] = prop; // put the data in the matrix

i++;

}

table = **new** JTable(rowData,columnNames); // create the table using the matrix and a parameter

**return** table;

}

* 1. **Testing**

The testing in this homework is quite different from the first homework where it was required to do the testing using JUnit. There is no mandatory way of doing the tests, so I chose to do them using try-catch structures and by printing errors if any occur during the input or execution part of the application.

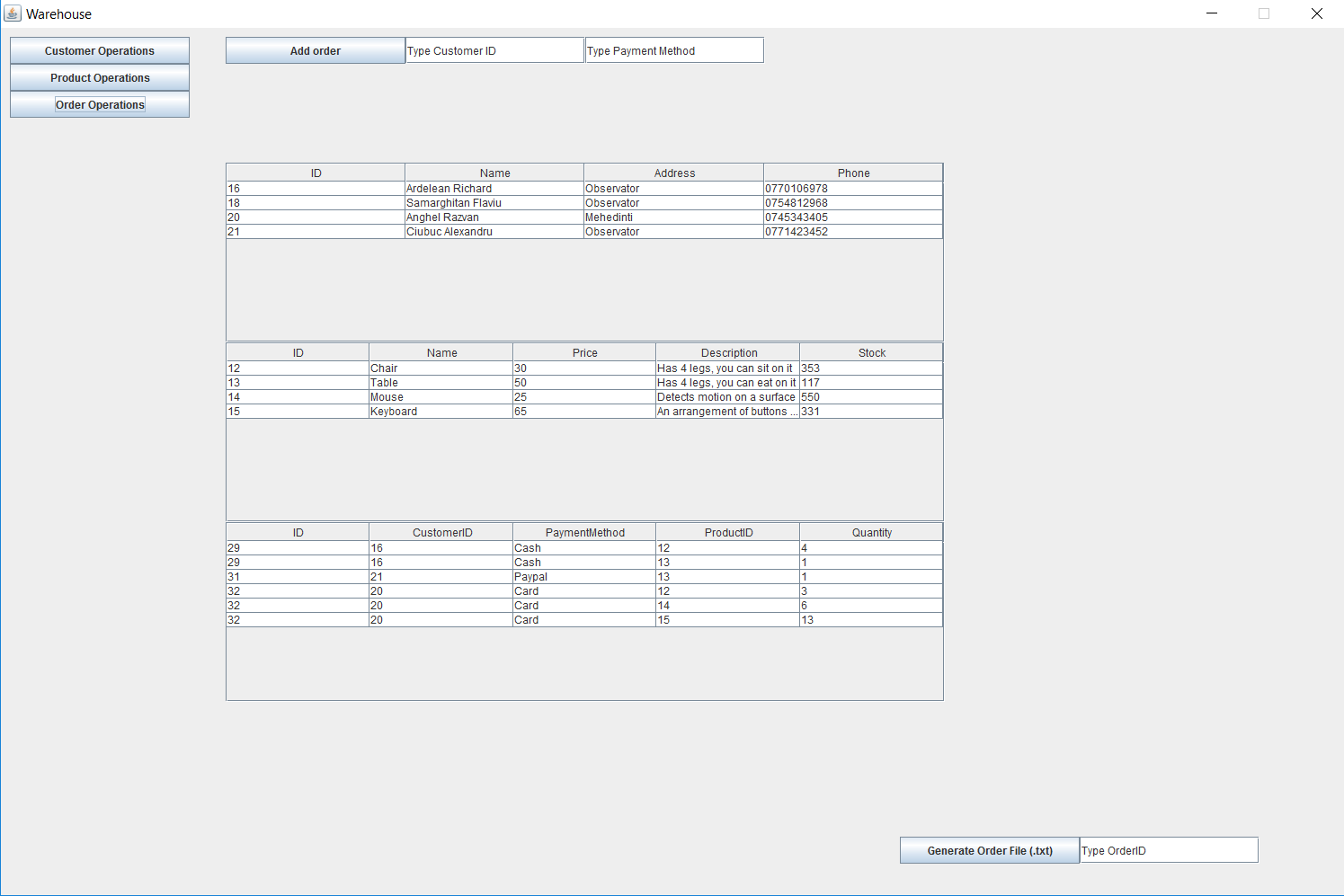
Certain errors will appear for introducing bad inputs, for example for introducing a string in a text field from where the user interface expects to read an integer will generate an error and it will reset the text field to the initial text.

Also for making the application more realistic, you cannot make an order for a product if the quantity you ask for is bigger than the stock of that product, this will generate an error that tells you that you have asked for too many and it will also reset the quantity text field.

1. **Results**

Through hard work and intensive testing, I have been able to create an application for ordering products from a warehouse. It allows the user to introduce customers and products in the tables. And generate orders. As expected it also allows the user to introduce an order to a chosen customer with as many types of products as he wants as long as the number of that type of product is in stock.

The final result looks something like this:

****

1. **Conclusion and future developments**
   1. **Conclusion**

As for the previous homework, our knowledge in this section (connecting databases to application and managing them was practically zero). But when you start learning about them and understanding how they work, everything becomes clear. Of great help was the example given with the explanation of how to link a database to an application.

Java reflection is hard to comprehend because everything you do is for general objects that most of the time you know nothing about. In this project, for example, when creating the jtable, you have no idea about any of the fields of the table (or what kind of table it is, as in what it contains), you don’t know the type of the fields or the number of them. It is difficult to understand how to write the code without first looking over examples.

* 1. **Future developments**

There are a lot of future developments that could be introduced into this project. Like expanding the fields for each table and adding new tables like: suppliers and shipping.

Also in the case of this application, the user is both the manager of the warehouse and the customer. A future development could be to differentiate the two roles, maybe by adding accounts and giving them certain permissions, ex. Customers to place orders and managers to modify database.

1. **Bibliography**

* Example and usage of vectors in java:

<https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html>

* To draw the diagrams:

<https://www.gliffy.com/>

* Answers to questions regarding JAVA syntax:

<https://stackoverflow.com>

* Explanations for diagrams:

<https://www.tutorialspoint.com/uml/uml_standard_diagrams.htm>

* Read a tutorial to better understand and be able to use the reflection technique in java:

<http://tutorials.jenkov.com/java-reflection/index.html>