Assignment 3 - Warehouse System

# 1.Objectives

The purpose of this assignment was to design an order management application for processing customer orders for a warehouse. In order to store all the necessary / needed data for the whole system, relational databases are used – by data meaning products, customers and clients. The whole design should include the domain specific classes, being the Order, Customer and Product classes; the whole business logic which consists of classes for order processing, warehouse administration and client administration; it is also necessary to provide presentation classes for the graphical user interface (GUI) and the data access classes for extracting, inserting, modifying and deleting data from the database.

# 2.Analyzing the problem

First of all, in order to reach a desired system construction, the main problem to be solved is creating the database, which in our case is the basic component of the whole application. Since there are three main domain specific classes, the database should contain at least these three: Order, Customer, Product (other classes to be added if necessary); these classes must have the fields exactly the same with the columns of the respective fields. This is one important step in the development that should be considered. Another criteria which should be respected in case of the database-application relationship is the importance of introducing (or trying to introduce) valid data into the database. For this reason we obviously need to implement some kind of validation for the information that is desired to be entered.

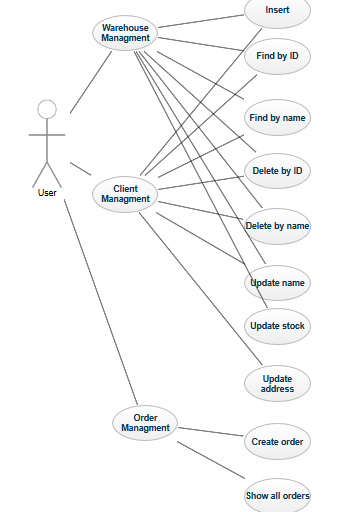
The other part is to create a (relatively) user-friendly application which allows people to enter new products (warehouse administration), add new customers to the system (client administration) and make it possible for customers to order items from the store.

In order to be able to work with real data that is actually stored in the database and to interact with it, some additional classes need to be added, others than the presentation and entity classes. This part should make a connection between the database and our application by sending queries to the server and receiving data in form a result set. The classes making the “connection” are called Data Access Object classes (actually data access object is an object or an interface that allows accessing the underlying database, provides kind of an interaction with it), providing methods for searching in the database, inserting elements into it, deleting entries or updating some data (this whole procedure basically lies on creating queries and executing them). The result obtained should from the query (statement) execution should also be processed somehow, because in the way they are returned is not possible to create a user-friendly output; these being stored in a result set the respective values for each object of the given type could be obtained by extracting them from the result set if each of the column names are known. This extraction could also be done (and should be done) by using reflection techniques. This technique is particularly useful when inspecting a piece of code in an application (in our case the result set content). These techniques allow to modify the runtime behavior of an application.

# 3.Modelling

As I mentioned previously, there are a few design steps that should be followed and may more problems that should be solved, respecting these steps. First of all, in the modelling stage, it is necessary to create the classes corresponding the database tables, the ones serving the presentation (graphical user interface classes), the DAO classes and the ones that control the logic of the whole system. Respecting the OOP paradigm, each component should enclose its own properties (encapsulation). In our case, the entity classes should enclose their own properties, which is actually obligatory, since they should have the same fields as the database tables. The business logic classes (this is the naming that I used for the classes representing the logic of the system) should include all the properties that allow the control of the system (basically the data manipulation), while the presentation classes (the ones that represent the GUI) encapsulate their properties in order to create a connection between the logical representation/control of the system and the outer world (the user).

Since this whole system is considered to be a complex one we should apply a layered architecture on it. Designing the mentioned architecture, I respected mainly the structure provided by the laboratory indications, meaning I have mostly the same packages. These layers include the following ones: the lowest level is the Data Access Level, which one contains the classes are the ones that contain the queries and the database connections; the second one, one level above the Data Access Level, is the Business Logic, which contains the warehouse management, client management and order management, so basically encapsulates the application logic; the third one, which could be considered the even the “top level”, is represented by the Presentation Layer and as I mentioned it previously a several times, it includes the classes defining the user interface.



# 4.Use Cases

In this project I developed an application that provides quite a few operations that the user can perform since actually the purpose of it is to offer the opportunity of controlling an online shop-like environment which is based mostly on the data entered by himself/herself. There are three main groups of use cases out of which two are basically admin-type operations and one is a customer-type operation. The first two groups are the Warehouse Management and Client Management. Warehouse Management includes the following sub-cases: insert product, delete product if given ID is found, delete product(s) if given name is found, update product name, update product stock, find product based on the given id, find product(s) based on a given name; Client Management has the following sub-cases: insert customer (add new customer to the system), delete customer if entered ID is found, delete customer(s) if entered name is found, update customer name, update customer address, find customer based on the entered ID, find customer(s) based on the entered name. The third use case is the Order Management which actually represents the act of creating a new order (this is rather the client side than the admin side); even the Order Management has sub-cases: one can enter a new order by giving the application the ID of an existing customer (basically his/her own identifier which should be available if he/she has registered himself/herself), the ID of a needed product and a desired quantity, also the user can display all the successful orders.

# 5.Scenarios – Insert new product

## Success scenario

1. The user runs the application.

2. If the application was successfully run, three windows appear on the screen and the user can decide, which part of the system he/she wants to manage – in this case the “Warehouse” window should be selected.

3.Once the user selected the desired window, which displays all the currently available products and the available operations (insert, delete, update, find), the user has to click on the desired operation’s button – in this case the insert button.

4.If the user successfully selected the desired operation, three text fields and a button appear underneath the operation buttons. It says next to each text field, what kind of information and with which purpose has to be entered- in this case product name, stock and price has to be entered.

5.After entering all the necessary data, the user has to press the “INSERT” button next to the last text field.

6.If the data entered was considered valid, a new entry appears in the displayed table containing the data provided by the user.

7.The user may continue this by selecting any other operation, even from other windows or just insert another item in the product table (adding a new available product to the system).

## Alternative scenario

8. The problem that may occur is that the data entered by the user may be considered invalid by the application – the user enters incorrect data (ex: string instead of a number as stock or price)

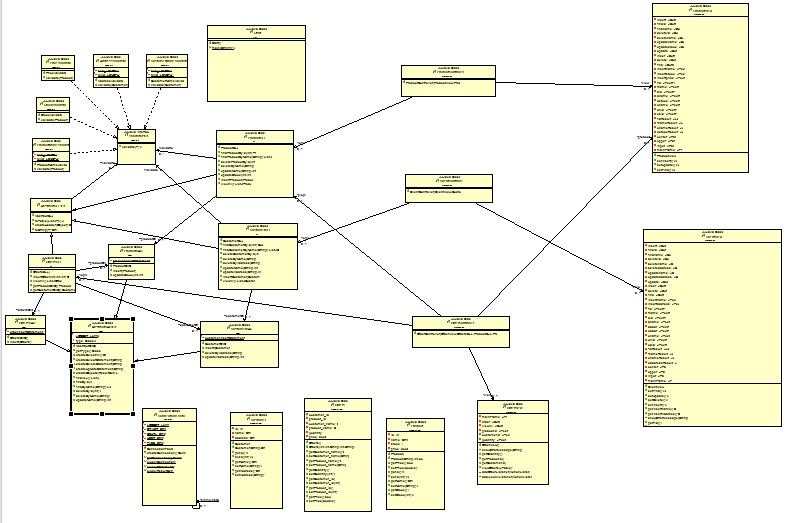
9. A pop-up window appears on the screen, telling the user which piece of data given does not satisfy the condition – the user has the opportunity to correct it

10.If the data is considered to be valid (this time) the user presses the “INSERT” button and now we are at step 5.

## Error Sequence

In case of this application all the unexpected events that can happen are mentioned in the previous part ( Alternative Scenario ) and they are also handled, so hopefully nothing else can go wrong.

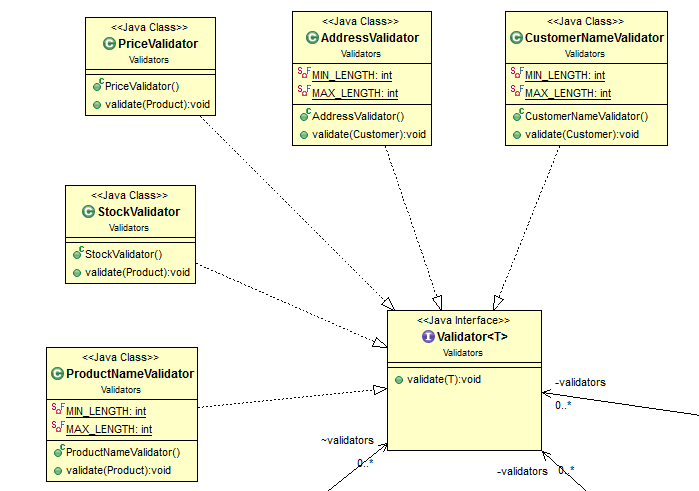
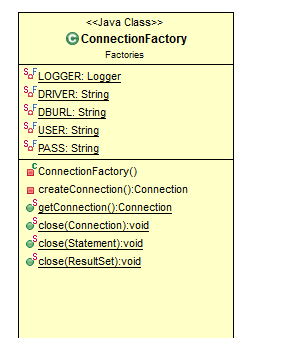
# 6.Design



## Model

The model includes three classes that are basically the entity classes of the system. One of them is the Customer class, which has as fields the same with the columns of the **Customer** table in the database, these being the id, name and address. It has a parameterless and a parametrized constructor (the first one is used with reflection) moreover a set of methods which are getters and setters. The second class out of this package is the **Product** class; from structural point of view it is quite similar with the previously mentioned Customer: it also has the fields identical with the corresponding database table columns, these being the id, name, stock and price and it also has two constructors besides a set of getters and setters. The last class that needs to be mentioned in this package is the **Orders** class. This is similar to the previous two classes, its fields are the following: customer\_id, product\_id, customer\_name, product\_name, quantity and the total price (the methods are getters and setter to this fields). In the database there is a one-to-many relationship both between the Customer-Orders table and the Product-Orders table.

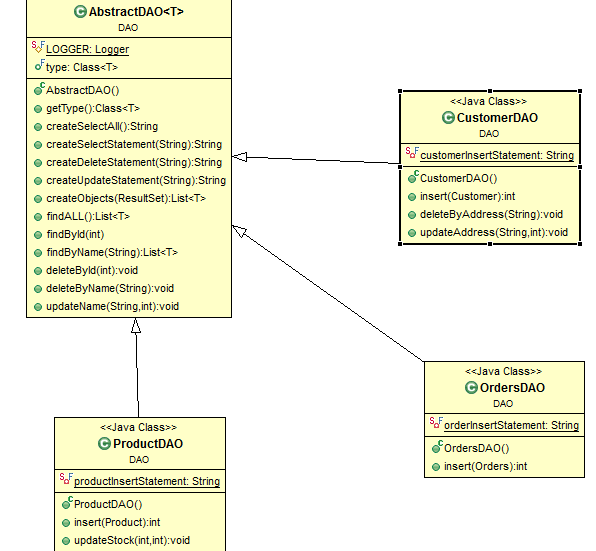
## Validators

The next package to be mentioned / described is the one that holds the validator classes. Since the data that is entered / going to be entered by the user in the application must correspond the type and the length of the database entry type, before taking the attempt to insert anything in the database, each piece of data has to be verified. For this first of all an **interface Validator** was created, which is also generic and the classes of this package implement this interface, which has one unimplemented method **validator(T).** The first ones to mention are the **ProductNameValidator** and the **CustomeNameValidator** (I am secribing them together because they are mostly the same with the difference that the first one in applied on product validation while the second one on customer validation). They both have the validator() method implemented in the same way : it verifies whether the name’s length is in the interval (5,45) or not; if not, the method throws an IllegalArgument exception that is handled on higher levels. The **AddressValidator** basically does the same thing, it implements the method in the way to check the length of the entered address. There are two more validator classes, both for Product class instances verifying numerical data; one of them (**StockValidator**) validates the entered data as stock if it is a positive number, while **PriceValidator** also checks if the double number introduced as price is positive or not.

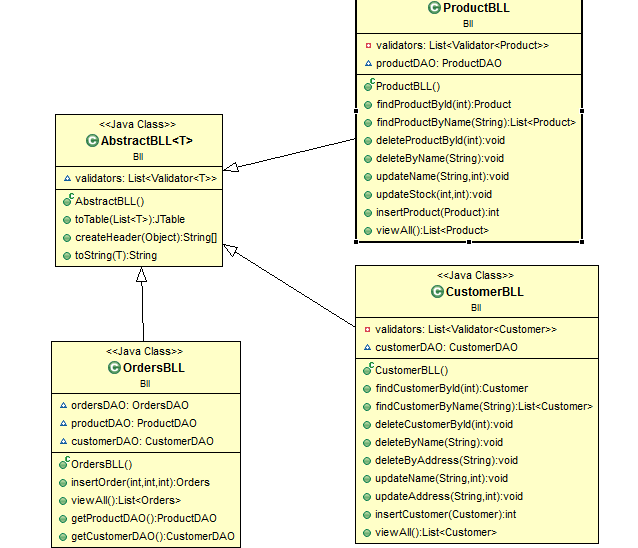
## Factories

The **Factories** package holds only a single class, this being the **ConnectionFactory** class. It has mostly static methods, except one, the **createConnection()** which tries to connect to the database each time it is necessary based on the final fields of the class (database url, username, password). The **getConnection()** returns the created connection belonging to the single instance from the class. There are also three closing methods: the first one, **close(Connection),** closes the previously created connection after there is no intention to extract more data from the database; the second one, **close(Statement)** closes a statement after it is no longer used – statements are used to execute queries, to set values to a previously returned string that is intended to have the structure of a query; the last one, **close(ResultSet)** closes the result set, that initially, after executing the query, holds the extracted data.

## DAO

This package is one important part of the application because it provides access, through connections of course, to the underlying database structure, it allows the user to and the application designer to access data in the database and manipulate it. First of all, there is a generic class, called **AbstractDAO<T>**  which provides a series of methods that all the subclasses use. This class implements methods, common operations for accessing a table and manipulating the data inside; these operations are the following: insertion, deletion, update, finding one element, finding all the elements. This is the part of the project which uses reflection techniques (allows the application to change behavior during runtime). This class has a field called **type** through which the class of the generic type is obtained. In order to perform any operation on the database, first a generic query has to be generated, in our case **createSelectAll()** returns a string that holds the query for extracting all the data from a table, **createSelectSatement()** retruns a string that allows accessing data based on a specific column (in this application based on name and id), **createDeleteStatement()** returns a string that holds the query for deleting an entry from a data table, **createUpdateStatement()** retruns the query for updating a row in a data table. All of these methods build a generic query – the name of the table is obtained by using the previously obtained type (this is the reason why the database table name most coincide with the class name). The methods executing the queries are the following: **findALL(), findById(), findByName(), deleteByName(), deleteById(), updateName().**  In all these methods first a connection has to be created with the database, then a statement is created based on the string returned by the query creation methods and the arguments of it are set (corresponding to the string based on which the query was created) and the returned result is obtained in the result set. Finally the result set, the statement and the connection are all closed independently if the data was correctly extracted or not. The next problem that has to be solved here is to obtain the model objects from the result set. In order to get the desired objects, for each result from the result set a new instance is created and corresponding to the number of fields of the type, these are set with the respective setter methods as attributes of the new instance. Each newly created instance is added to the result list which is returned when there are no more results in the result set. In this package there are three more classes that extend this generic class and access its methods directly through inheritance, moreover, here are some more specific methods implemented. The first one of these classes is the **ProductDAO**  class that adds two extra methods to the implementation: **insert()** and **updateStock()** ( the first one allows executing a query for inserting an entry to the database, the second one allows updating the stock of a product). The **CustomerDAO**  class implements methods for inserting new entry in the Customer table (**insert()**)**,** deleting an entry if the address field of it corresponds to the one given by the user (**deleteByAddress()**) and one method for updating the address of a client (**updateeAddress()**)**.** The last class in this package is the **OrdersDAO**  which only adds a method for inserting a new order in the Orders table (**insert()**).All these methods are written in the above described way.

## BLL

This package has quite a similar structure to the DAO one, since it also has a generic superclass which is extended by other three subclasses. This generic superclass is called **AbstractBLL<T>.** It holds as field a list of validators and three methods. The reason why I created this class is, to hold some generic methods of the application logic. These methods are the following: **toString()**, which generates a human readable string from any object, separating its fields; **createHeader(),** which returns a string with all the field names of a class; **toTable()**  which receives a list of objects (of any class) and using the createHeader() method it returns a new JTable with the content of the list – first it transfroms the list into a multi-dimensional array, than creates a new JTable with the header and the multi-dimensional array.

The other three classes extend this generic one and each adds up to the inherited methods its specific behavior. The first one, **ProductBLL,** using its DAO class implements its own behavior. **FindProductById()** returns the product from the table that has the entered id otherwise throws and exception, **findProductByName()**  retruns the found products if they have the entered name, **deleteProductById()** deletes a product based on the id of it, **deleteProductByName()** deletes products based on the name they have, **updateName()** updates the name of a given product (the user eneteres the id and the new name), **updatesStock()** updates the stock of a product, **insertProduct()**  allows the user to insert a new product and **viewAll()** returns a list of products composed of all the existent entries from the Product table.

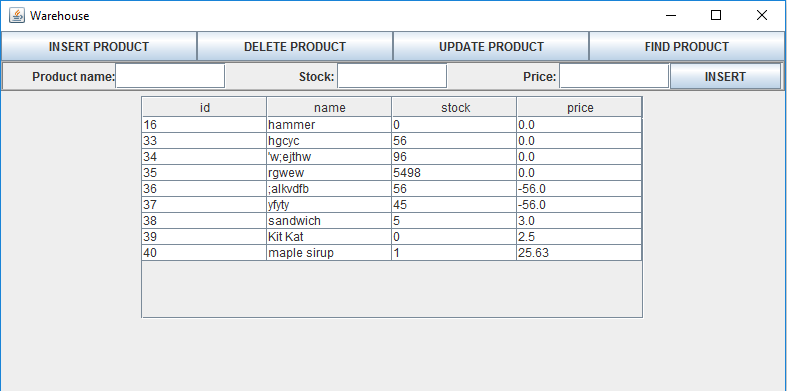
**CustomerBLL** is very similar to the previously described class. It has methods for finding customers (**findCustomerById(), findCustomerByName()**), for deleting customers (**deleteCustomerByName(), deleteCustomerById()**), updating customer data (**updateName(), updateAddress()**) and inserting a new customer in the data table (**insertCustomer()**).

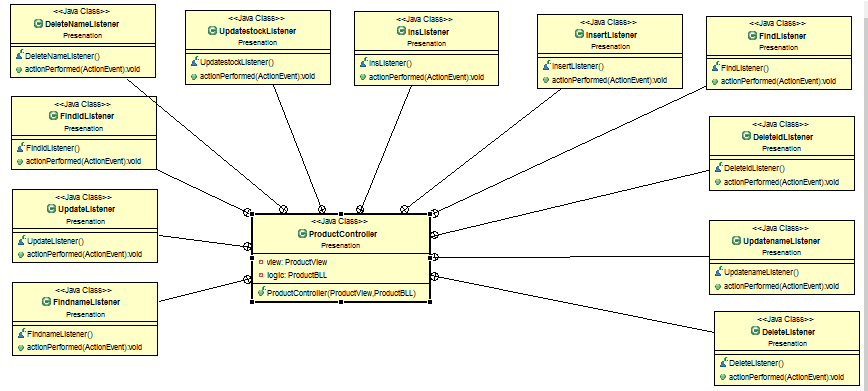
The last class to mention in this package is the **OrdersBLL** class. It has methods for inserting a new order (**insertOrder()**), viewing all the orders (returns a list – **viewAll()** )**,** plus getters and setters.

## Presentation

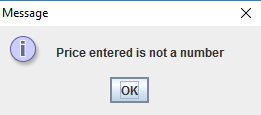
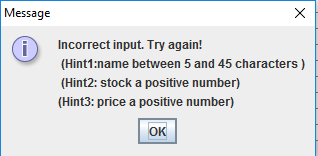
The graphical user interface part of the application represent the presenation package, which holds three views and three controllers, since once the applcation is run, three windows appear on the screen. The view of the Customer Service window ( that’ how I call the one in which th user can add customers to the system) is very similar to the Warehouse window (the one which allows adding new products to the system). The third one is the Online Shop (the user can order a product).

### ProductView, ProductController - Warehouse





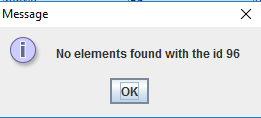
First, when the application starts, a new window appears for the Warehouse; it displays in the lower part of it a table with all the previously entered products, above it there are four buttons; if one click on any of them, between the table and the buttons the respective text fields appear where the user can enter all the necessary information that is needed to create a valid entry. In the case above, for insertion, a text field for product name, stock and price has to be completed (the id is automatically generated in the database – the user doesn’t have to bother with it). If he/she considers the input correct, presses the “INSERT” button. If any problems occur (too long/short name, not a number as stock or price, negative number as stock or price) a pop-up window appear, telling the user, what problem is met.



The user has the opportunity to retry entering the input. As the operation was successful, the table is immediately refreshed with the **setTable()**  method found in the ProductView class, which deletes al the contents of the panel, recreated a table, revalidates and repaints the panel than. In the ProductController class all the listener classes are present for the buttons and text fields present.

There are three more scenarios besides the insertion: DELETE, UPDATE and FIND.

****

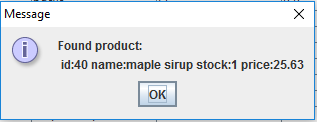
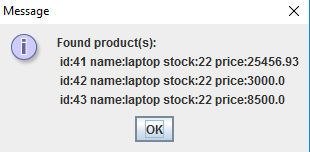
When deleting a product, the user can decide whether he/she would like to delete a product based on a selected name or on an id (based on the name even more than one entries may be deleted, when deleting base on the id, only one entry is deleted since the id is unique). If some of the data is not valid, or there are no entries with the respective id or name, a pop-up window appears, telling the user, what problem is met. The user has the opportunity reenter the data that was mistaken; as the data is correct, the table is redrawn, but this time without the deleted data.

In case of updating a product, the user has to separately update the name and the stock of a product – this is useful in those cases when not the whole product data has to be modified.

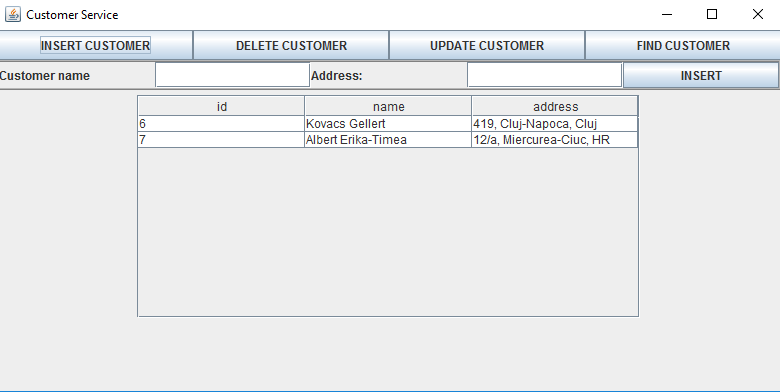


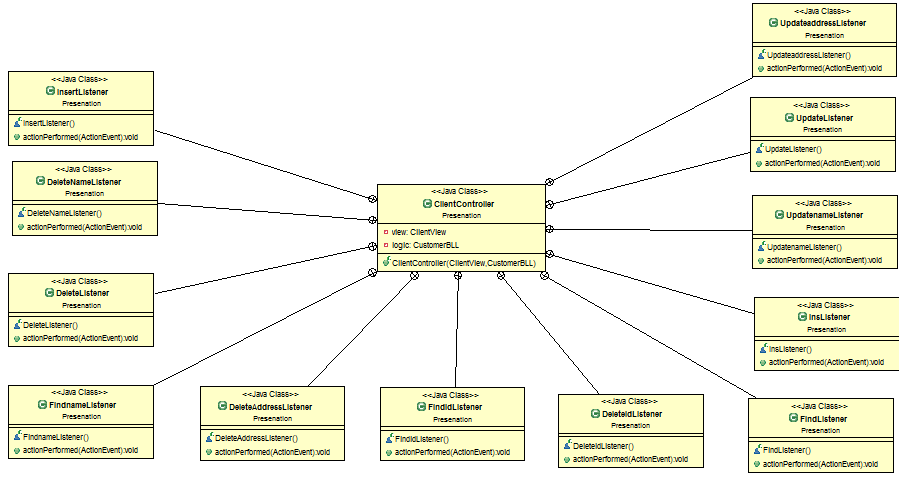
First the new name/stock number has to be entered and then the ID of the item that needs to be updated. If no item with the respective ID is found, an error message is displayed in a pop-up window. In case all the data is correct, the entry is updated and the table is redrawn.

The last part of the presentation that has to be described is the FIND part. Here the user has again multiple opportunities, since he/she can find a product based on id or name (when finding by name, even multiple results may be found). If the id or name entered cannot be found, an error message is displayed in a pop-up window; if the search was successful, also in this case a pop-up window appears, showing the results of the search.

****

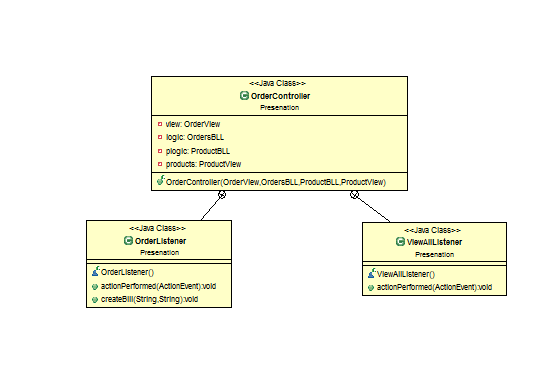
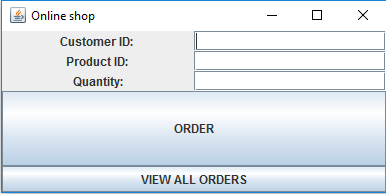
### ClientView, ClientController – Customer Service





From structural point of view, the Customer service is quite identical with the Warehouse; it has the same components (the view), the same functionalities and features – INSERT, DELETE, UPDATE, FIND. There are very few differences: in case of insertion, name and address have to be entered; user can delete by id, customer name and address; name and address can be updated; a customer can be found based on id and name. The application behaves the same way in case of success and failure like in the previous part.

### OrderView, OrderController – Online Shop



In case of the Online Shop, the structure and the functionalities are a little different than in the previous two cases. In order to finalize an order, the user has to type in first the user ID (let’s say it’s his/her own identifier), then the product ID and finally the quantity that he/she would like to order. If the quantity is greater than the current stock, an error message is displayed in a pop-up window, telling the user that there are not enough products on stock. In case the ids are not found, an error message is also displayed. If the order was successfully finalized, a pop-up window appears with the product name, client name, identifiers and total price besides a generated bill (a text file created by the **createBill()** method). In case the user would like to see all the successful orders, he/she has to click on the VIEW ALL ORDERS button; a new window is immediately created showing a table with all the previously sent orders.

## Start

This package contains only a single class, called Start, that holds the main method (creates an instance for all the 3 main views and logical controllers and runs the whole application).

# 7.Implementation and testing

## Model

The first classes to be implemented were the entity classes – first I designed the database and then, corresponding to the tables’ columns I wrote the classes with these fields.

## Logic

Looking through the presentation of the homework and the indication slides, I tried to understand the generic and reflection concepts while recreating the code. I looked up the query structures, how they should be constructed and I wrote the methods for executing them. I tested if I could connect to the database and insert/update/delete data. After the DAO methods seemed to be correct, I implemented the logic above them and I tested those again.

## Presentation

I have an overview of the GUI in my head before implementing it and I tried to decode it. I made many modifications until I reached a final form by continuously testing – this way I could also identify which are the error cases that should be handled.

# 8.Results

After many hours of coding and debugging, I think I reached a final solution of the problem which allows any user to efficiently and easily administrate the functioning of a warehouse (an online shop). Also the result are displayed in a quite straightforward way so he/she can easily understand what was mistaken and if the operation was successful it can also be followed.

I consider that the application is quite straightforward for every user and it is not difficult at all to use it. After a few minutes anyone can easily familiarize himself/herself with it.

# 9.Conclusions

## What I learnt

This assignment was very useful, because I could familiarize myself with workig with a database, connecting to it, extracting information and processing the result; I also learnt how one can change the behavior of a program during runtime. I had to clearify the concepts of generics and reflection but I also had to think about the user’s point of view to create a user-friendly application, if possible.

## Further implementations

The first improvmet I can think of is to make the GUI even more straightforward by separating clearly the administrator (Warehouse, Customer Service) and the user part (Online shop). I think it would be also a great improvment if some kind of identification should be needed before using the application (for the administrator rather). I could also think about creating cathergories for the products or giving a promotion to those customers who have ordered products above a given amount or a a given number of times.

## 10.Bibliography

<http://www.coned.utcluj.ro/~salomie/PT_Lic/3_Lab/HW3_Tema3/HW3_Indications.pdf> - code for the DAO, BLL and database connection

<http://stackoverflow.com> – when dealing with bugs

<https://docs.oracle.com/javase/tutorial/reflect/> - Reflection API

<http://www.javatips.net-> class diagram

<https://app.genmymodel.com-> use case diagram

<https://www.tutorialspoint.com/java/>