Order management application

~ Homework 3 ~

Papita Anda

Group 30421

Programming techniques

Table of Contents

1. Objective

The objective of this laboratory homework is the design and implementation of an order management application meant for processing customer orders for a warehouse. In other words, this application uses a relational database for storing the customers, the orders and the products. The problem is commonly met for online shops or simply whenever people need to keep track and update a larger set of objects. The application implements a system of utility programs such as calculating totals, filtering information from one or multiple tables, reporting under-stock products, inserting, deleting, updating new items and so on.

Hence, the purpose of this app is minimizing the complexity with which users can keep track of their orders, clients and products and manipulate large sets of data. From an educational perspective, this assignment was meant to help the student get familiar with the concept of reflection techniques and working with databases in Java.

2. Problem analysis, modeling, scenarios, use-cases

In order to be able to work with different types of tables and using a minimum number of methods for accessing the MySQL database, one must understand the concept of reflection. 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. The ability to examine and manipulate a Java class from within itself may not sound like very much, but in other programming languages this feature simply doesn't exist. For example, there is no way in a Pascal, C, or C++ program to obtain information about the functions defined within that program. The reflection classes, such as Method, are found in java.lang.reflect. However, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.

In this case, the application uses multiple reflection techniques, more specifically to create a generic class that contains the methods for accessing the DB: create object, edit object, delete object and find object. These methods can be accessed in the case of each table of the database and they can all benefit of their functionality.

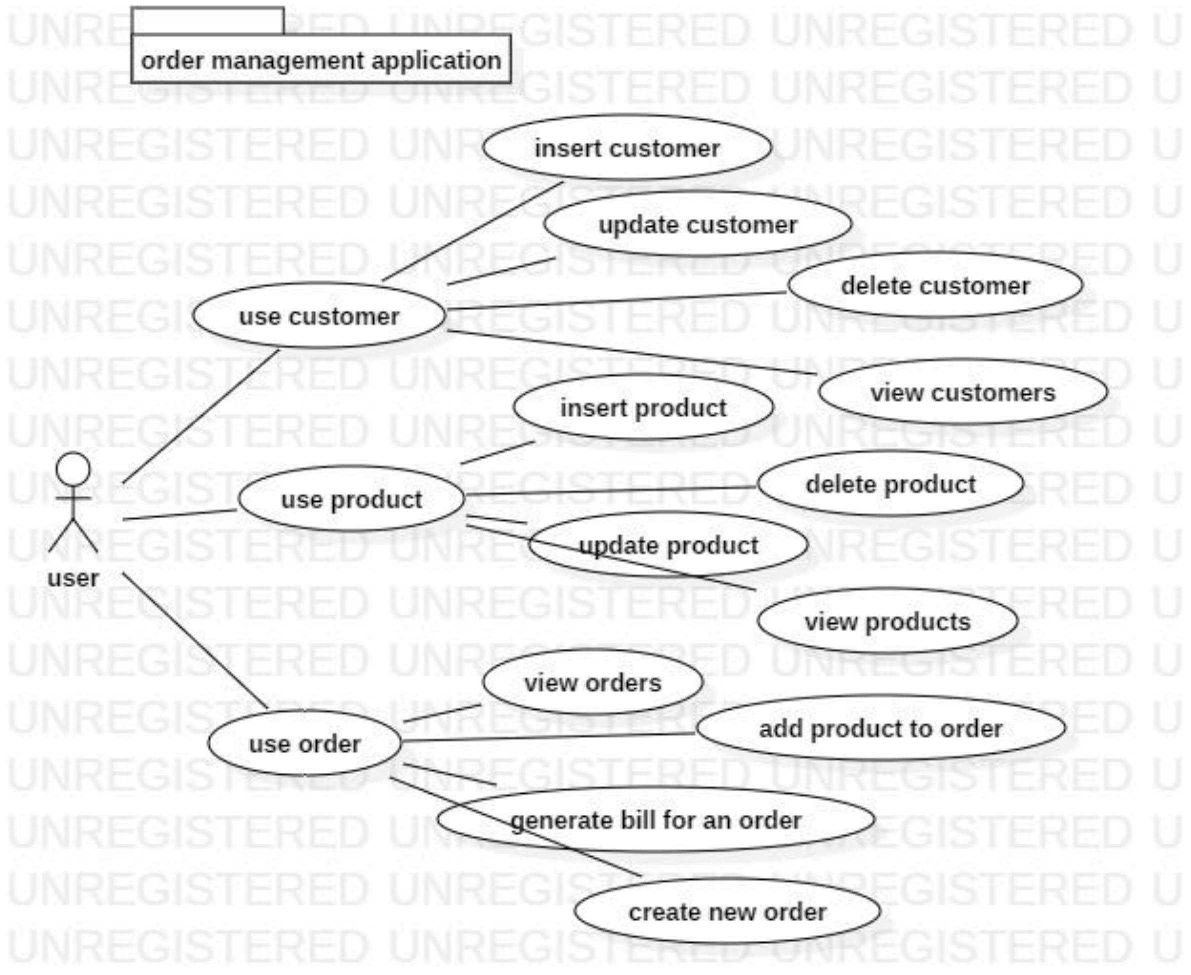
Example of use case scenario

Use case title: insert a customer

- **Summary: this use case allows the user to add a customer to the customers table**
- **Actor: the user of the application**
- **Main success scenario:**
    - The user introduces the id, first name, last name, email, phone, card no. and address of a customer
    - the validators check if the data introduced by the user is correct by some pre-defined standards for each field
    - the new customer is introduced in the database through connectivity
- **Alternative sequences:**
    - **a)** Incorrect first or last name
        1. A pop-up message will show and tell the user the first or last name is incorrect
        2. The app returns to the second step
    - **b)** Incorrect email
        1. A pop-up message will show and tell the user the email is not valid
        2. The app returns to the second step
    - **c)** Incorrect phone number
        1. A pop-up message will show and tell the user the phone number is not valid, i.e. not composed of 9 digits
        2. The app returns to the second step
    - **d)** Incorrect card number
        1. A pop-up message will show and tell the user the card number is not valid, i.e. not composed of 3 digits
        2. The app returns to the second step

All the other use cases are similar to this one.

The use case diagram is the following one:



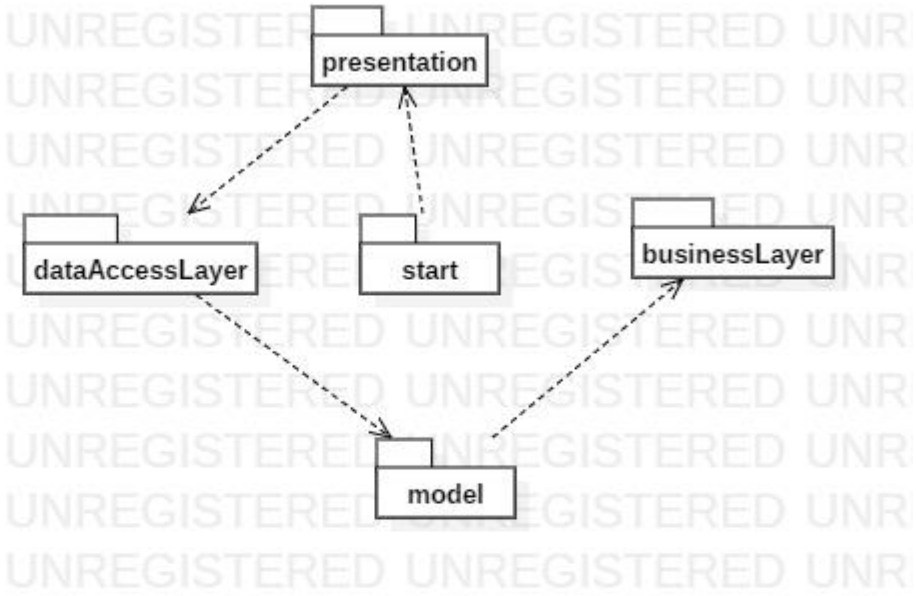
3. Design

The design of this app follows a layered design architecture. The 5 packages are: model, presentation, businessLayer, dataAccessLayer and start. Therefore, the model package is represented by the classes Customer, Product and Order. These classes are the ones which contain as attributes all the fields that can be found in the corresponding tables from the database. The dataAccessLayer is the package that contains the classes that handle the connectivity between the database and the model classes. It contains the ConnectionFactory class, the AbstractDAO, CustomerDAO, ProductDAO and OrderDAO classes. The presentation package, represented by the classes View and Controller, follows a bit the typical MVC pattern. Thus, The View class is responsible for the graphic user interface design. It extends the JFrame class. The Controller acts on both model and view, and in addition it also uses the dataAccessLayer classes. It also controls the data flow between what the database already has and what the user updates or introduces.

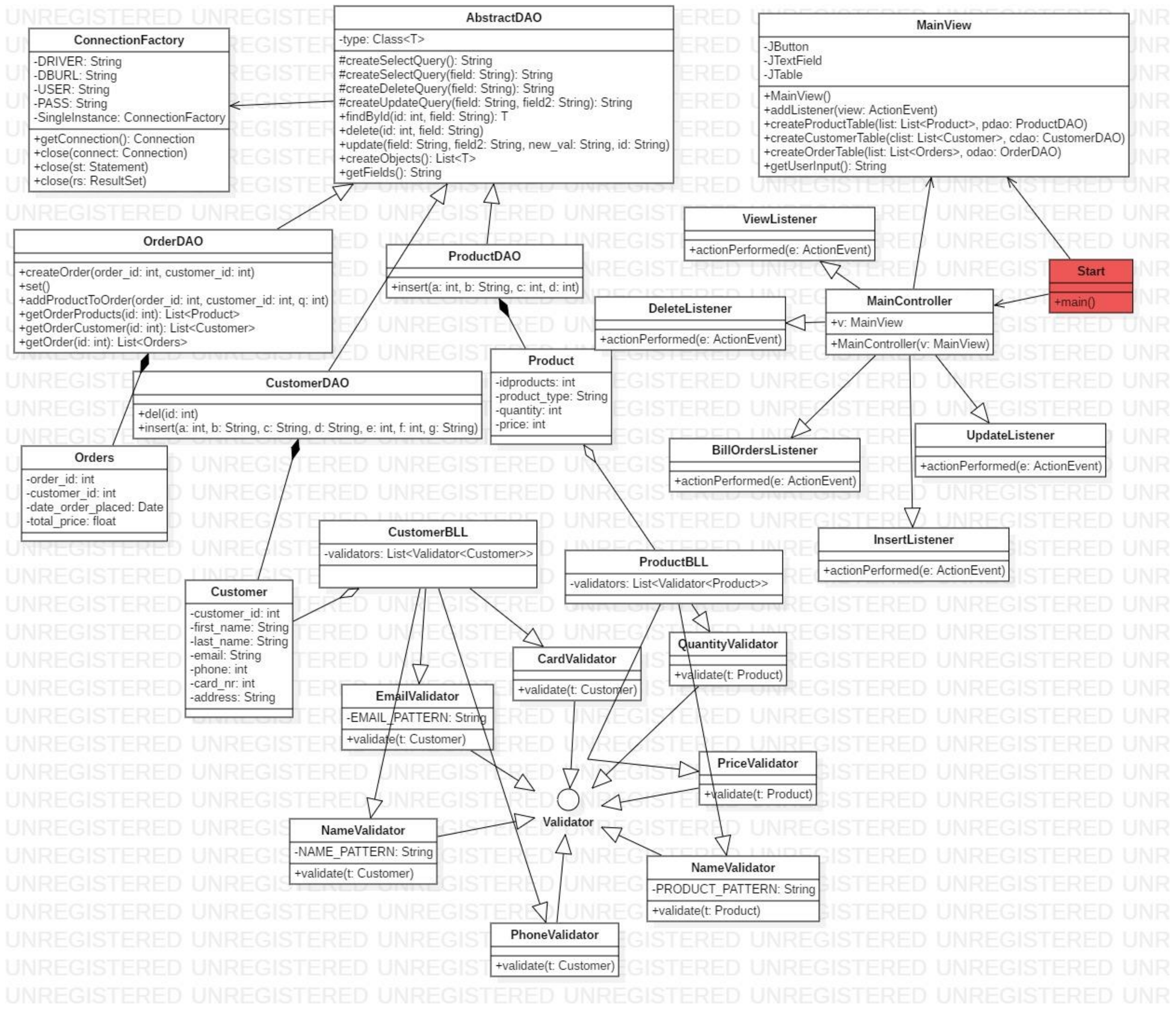
More precisely, it coordinates each button the user can press to perform a specific action and helps display the resulting consequences. Finally, the businessLayer package contains the logic of the application and that is what types of data can populate the tables. Its classes extend the Validator interface: CustomerBLL and ProductBLL. In addition to these elements, the app contains the Start class from the start package which links the dataAccessLayer, controller and view together and starts the execution, making the interface visible and giving the user the permission to interact with the database.

The connections between packages can be seen in the following diagram:

Package relationships



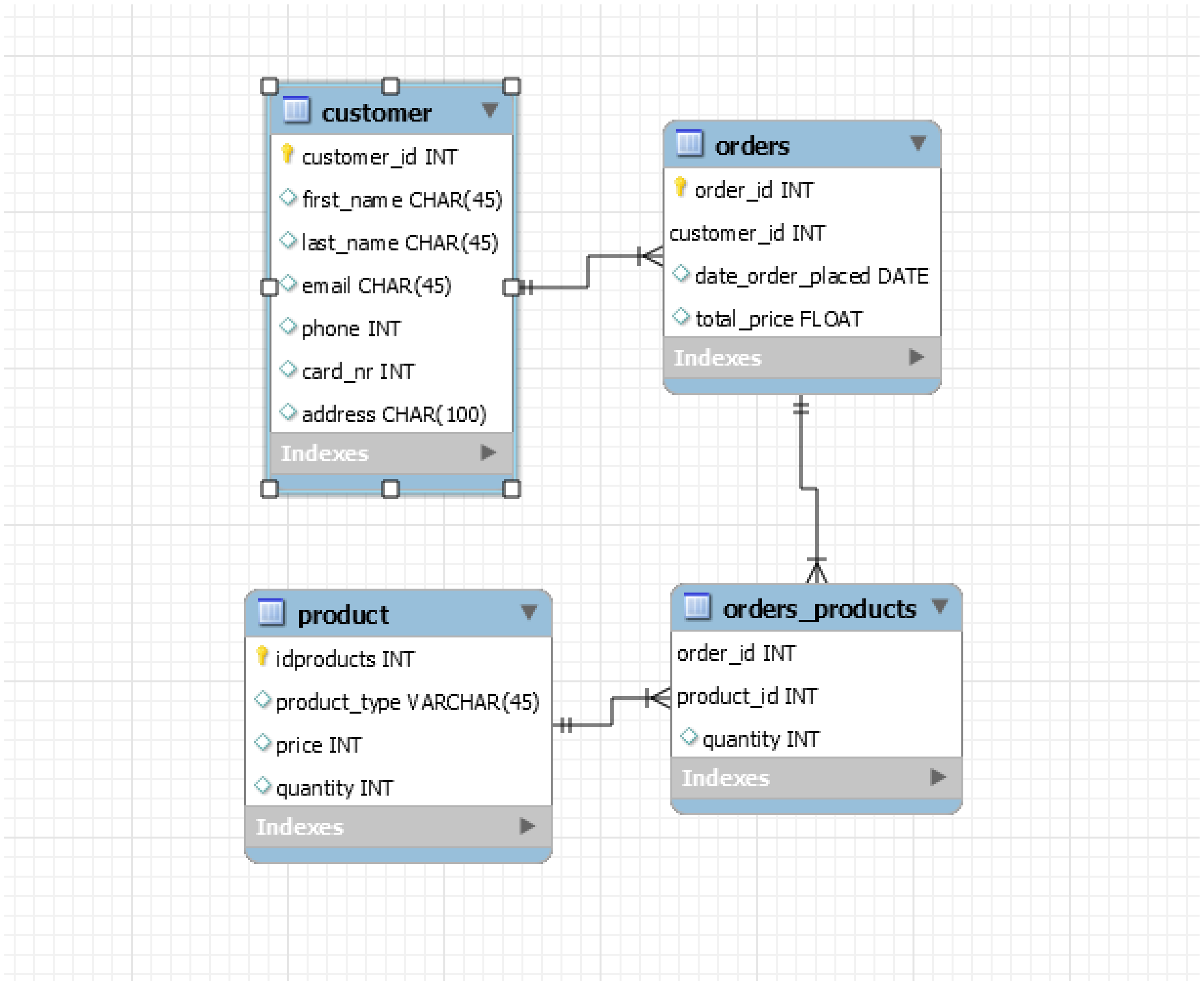
The UML diagram of the project is the following one:


AbstractDAO
-type: Class<T>
#createSelectQuery(): String
#createSelectQuery(field: String): String
#createDeleteQuery(field: String): String
#createUpdateQuery(field: String, field2: String): String
+findById(id: int, field: String): T
+delete(id: int, field: String)
+update(field: String, field2: String, new_val: String, id: String)
+createObjects(): List<T>
+getFields(): String
ConnectionFactory
-DRIVER: String
-DBURL: String
-USER: String
-PASS: String
-SingleInstance: ConnectionFactory
+getConnection(): Connection
+close(connect: Connection)
+close(st: Statement)
+close(rs: ResultSet)
MainView
-JButton
-JTextField
-JTable
+MainView()
+addListener(view: ActionEvent)
+createProductTable(list: List<Product>, pdao: ProductDAO)
+createCustomerTable(clist: List<Customer>, cdao: CustomerDAO)
+createOrderTable(list: List<Orders>, odao: OrderDAO)
+getUserInput(): String
OrderDAO
+createOrder(order_id: int, customer_id: int)
+set()
+addProductToOrder(order_id: int, customer_id: int, q: int)
+getOrderProducts(id: int): List<Product>
+getOrderCustomer(id: int): List<Customer>
+getOrder(id: int): List<Orders>
ProductDAO
+insert(a: int, b: String, c: int, d: int)
ViewListener
+actionPerformed(e: ActionEvent)
Start
+main()
MainController
+v: MainView
+MainController(v: MainView)
DeleteListener
+actionPerformed(e: ActionEvent)
Product
-idproducts: int
-product_type: String
-quantity: int
-price: int
CustomerDAO
+del(id: int)
+insert(a: int, b: String, c: String, d: String, e: int, f: int, g: String)
Orders
-order_id: int
-customer_id: int
-date_order_placed: Date
-total_price: float
BillOrdersListener
+actionPerformed(e: ActionEvent)
UpdateListener
+actionPerformed(e: ActionEvent)
InsertListener
+actionPerformed(e: ActionEvent)
CustomerBLL
-validators: List<Validator<Customer>>
ProductBLL
-validators: List<Validator<Product>>
Customer
-customer_id: int
-first_name: String
-last_name: String
-email: String
-phone: int
-card_nr: int
-address: String
CardValidator
+validate(t: Customer)
QuantityValidator
+validate(t: Product)
EmailValidator
-EMAIL_PATTERN: String
+validate(t: Customer)
PriceValidator
+validate(t: Product)
Validator
NameValidator
-NAME_PATTERN: String
+validate(t: Customer)
NameValidator
-PRODUCT_PATTERN: String
+validate(t: Product)
PhoneValidator
+validate(t: Customer)

Database design

Moreover, the design of this application also consisted in designing a MySQL database that the app accesses to get/modify data. The order_management database implemented for this purpose consists of 4 tables. Three of them represent the data from the three tables (customer, product and order) and the fourth one, orders_products, represents a helper table which represents a many-to-many relationship between the orders table and the product table. There is another one-to-many relationship between the customer and order tables, as a customer can have multiple orders.

The diagram of the database is the following one:



4. Implementation

Customer class: this class represents the format of a customer which can be introduced in the customers table. Each object of type Customer will have as attributes the customer_id, first_name, last_name, email, phone, card_nr, address, the attributes the user should know in order for an order to be processed. Its methods are only getters and setters, a constructor and a toString function.

Product class: this class represents the format of a product which can be introduced in the products table. Each object of type Product will have as attributes the idproducts, product_type, price, quantity, the attributes the user should know in order for an order to be processed. Its methods are only getters and setters, a constructor and a toString function. In other words, identical to the previous class.

Order class: this class is identical with the previous two, it contains the attributes of an order, as they are reflected in the orders table: order_id, customer_id, date_order_placed and total_price. It also consists of getter, setters and a toString method.

ConnectionFactory class: this class establishes the connection between the application and the MySQL database as follows: it needs a driver, the database url, the user and the password. It contains a simple Constructor, which `uses the forName() method in order to register the driver class.` Its method getConnection() is used to establish the connection with the database. It contains three more close() methods which close the connection, an object of type Statement and a ResultSet.

AbstractDAO class: this class implements the generic methods that can be implemented by all the other DAO classes. As attribute, in has a Class type, which represents a class in a program at runtime. Its constructor simply uses the getClass() method, which returns the reference to the Class object of the class of the object.

Methods:

- `protected String createSelectQuery()` this method creates a basic select query which selects all the fields of a table; it uses StringBuilder to build the desired query
- `protected String createSelectQuery(String field)` this method also creates a select query, but this one is specific to just one field, given as parameter
- `protected String createDeleteQuery(String field)` this method created a delete query with regard to a specific field
- `protected String createUpdateQuery(String field, String field2)` this method creates an update query. It sets the record of the String field according to the field2.
- `public T findById(int id, String field)` this generic method can find a row in a table by identifying one of its fields (specified by field) by the value of the given id. It uses the createSelectQuery(String field) to prepare a PreparedStatement.
- `public void delete(String field, int id)` this one, a generic method as well, deletes a record from a table identifying it by a field and the value in it, represented by the id parameter

- `public void update(String field, String field2, String new_val, String id)` this generic method works by the same principle as the previous ones: it updates a field and its new_val (new value) by identifying the desired row by any field (field2) and its id.
- `public List<T> createObjects()` this method creates a list of generic objects with all the contents in a table, including the fields' names and all the data recorded.
- `public String[] getFields()` this method creates a vector of String objects which holds the names of the fields of a table.

CustomerDAO class: this class extends the AbstractDAO class and can implement all of its methods. In addition to those methods, it has an insert one, which inserts a new record in the table. The generic delete method is also called in this class using super(), as it specifies to always delete a customer using the field customer_id.

ProductDAO class: this class also extends the AbstractDAO class, and just as CustomerDAO, it has its own insert method. No other special methods are implemented.

OrderDAO class: just like all the other dao classes, this one also extends the AbstractDAO class, but it has more independent methods which use queries that are have a higher level of complexity

Methods:

- `public void createOrder(int order_id, int customer_id )` this method inserts new values in the table orders, it basically creates a new order. The user needs to specify just the order and customer id (parameters) as the date field will be automatically completed with the current date and the total price field will be initialized with 0.
- `public void set()` this method implements a query that sets the safe-update preference to 0, which needs to be done every time an update is needed.
- `public void addProductToOrder(int order_id, int product_id, int q) throws Exception` this method uses a procedure, rather than a query, by which it adds a new product to an order. This is achieved by adding values to the orders_products table, setting the new total price in the order table and decrementing the quantity of a product in the products table, by the amount of products the user wishes to order. Moreover, this method looks for the desired product using the findbyId method and gets its quantity value in order to check if enough products, if at all, are on stock. In case no products,

are on stock, it will throw an Exception a pop-up message will appear. The procedure was created within MySQL, and has the following structure:

1. **CREATE DEFINER=`root`@`localhost` PROCEDURE `addprod`(in orderid int, in productid int, in q int)**
2. **BEGIN**
3. **insert into orders_products values(orderid, productid, q);**
4. **update product**
5. **set quantity = quantity-q where idproducts=productid;**
6. **update orders**
7. **set total_price=(total_price + (select price from product where idproducts=productid)*q) where order_id = orderid;**
8. **END**

- `public List<Product> getOrderProducts(int id)` this method creates a list with all the product that are in an order
- `public List<Customer> getOrderCustomer(int id)` this method creates a list which contains the necessary information about the customer of an order
- `public List<Orders> getOrder(int id)` this method creates a list which contains the necessary information about an order

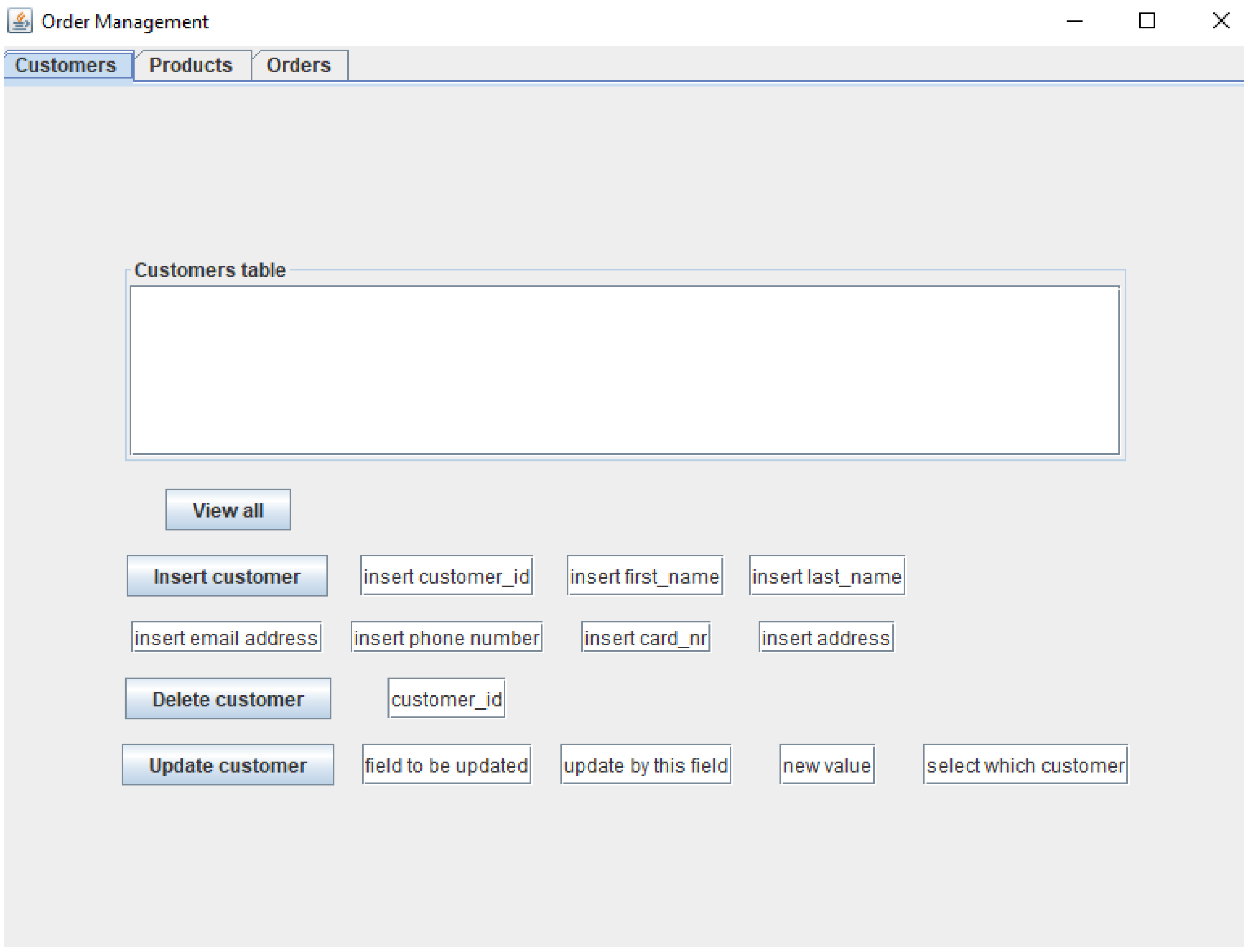
MainView class: this class creates the interface the user works with regarding to introducing the necessary and desired values in the database tables, and also showing the tables' records. It extends the JFrame class. Its components include JTables, text fields and buttons and tabbed panes. This time I found the most appropriate and perhaps convenient to use the GridBagLayout with respect to the panels organization. I set constraints to each component. They refer to the number of columns and lines of the panel. Thus, the panel can be seen as an imaginary matrix and each element is placed in one matrix element. Everything is arranged within the constructor, MainView().

Additional methods:

1. `public void createProductTable(List<Product> list, ProductDAO pdao)` -> this kind of methods creates a table based on a list of objects
2. `public void addViewListener(ActionListener view)` this kind of methods adds an action to a button, which will be implemented within the controller

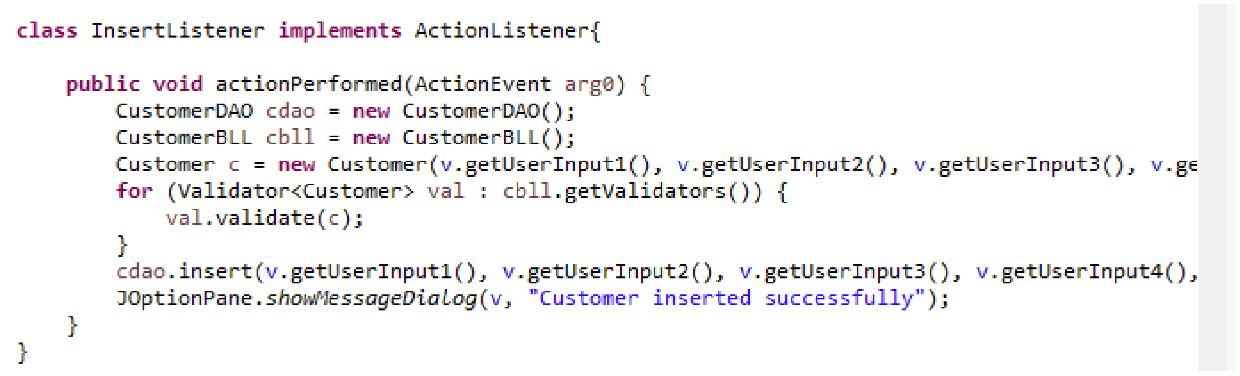
3. `public int getUserInput1()` this kind of methods simply gets the values from the text fields

The interface has the following structure:



MainController class: this class acts on dataAccessLayer, model and MainView classes. It controls the data flow that goes into the database and updates the View whenever data changes and the other way around. More precisely, it coordinates each button the user chooses to press. It consists of one constructor that implements view functions. The class also contains multiple listener classes for all the buttons.

For example:



```
class InsertListener implements ActionListener{

    public void actionPerformed(ActionEvent arg0) {
        CustomerDAO cdao = new CustomerDAO();
        CustomerBLL cbll = new CustomerBLL();
        Customer c = new Customer(v.getUserInput1(), v.getUserInput2(), v.getUserInput3(), v.ge
        for (Validator<Customer> val : cbll.getValidators()) {
            val.validate(c);
        }
        cdao.insert(v.getUserInput1(), v.getUserInput2(), v.getUserInput3(), v.getUserInput4(),
        JOptionPane.showMessageDialog(v, "Customer inserted successfully");
    }
}
```

Start class: this is the class that starts the application. It only contains a main() method.

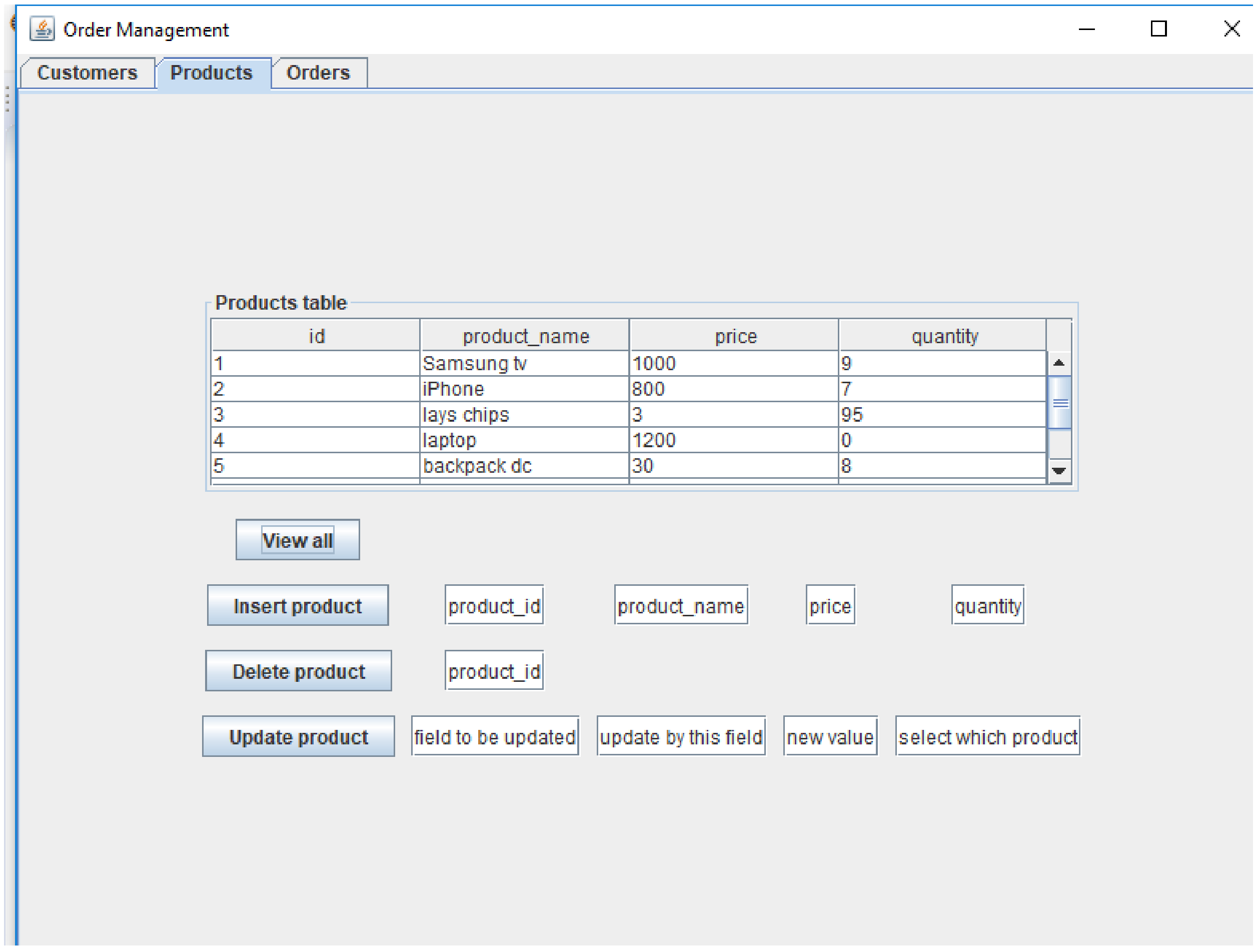
Validator interface: this is the interface implemented by all the other classes in the businessLayer package. It simply contains an unimplemented method validate(T t).

CustomerBLL class: this class implements the previously mentioned interface. It contains four validator classes that verify names, email, phone and card number.

ProductBLL class: this class also implements the previously mentioned interface. It contains three validator classes that verify the product name, quantity and price.

5. Results

As we can see in the following picture, this is how the interface displays data with various data from the existent tables:



Products table

id	product_name	price	quantity
1	Samsung tv	1000	9
2	iPhone	800	7
3	lays chips	3	95
4	laptop	1200	0
5	backpack dc	30	8

6. Conclusions

In conclusion, this type of applications is in high demand all over the commerce business, especially e-commerce and represents the basis of a web developed shopping site. It can further be improved to be safely used by the customer, in a different manner than it is used by the owner of the products.

7. Bibliography

1. https://www.geeksforgeeks.org/reflection-in-java/
2. https://www.javatpoint.com/example-to-connect-to-the-mysql-database
3. https://docs.oracle.com/javase/tutorial/uiswing/components/tabbedpane.html