*Technical University of Cluj-Napoca*

*Faculty of Automation and Computers*

*Department if Computer Science*

*2nd Semester 2015-2016*

***Programming Techniques***

***Homework 2***

Student: Hitter András

Group: 30424

Contents

[Problem Specification 3](#_Toc447641704)

[1. Example of working 6](#_Toc447641705)

[2. Design 8](#_Toc447641706)

[2.1. Relational Diagram 8](#_Toc447641707)

[2.2. Class Design 10](#_Toc447641708)

[Use Case Diagramm 23](#_Toc447641709)

[3.3 Packages and Interfaces 23](#_Toc447641710)

[3. Using and testing the application 25](#_Toc447641711)

[4. Conclusions 26](#_Toc447641712)

[5. References 27](#_Toc447641713)

Abstract

Formally, a "database" refers to a set of related data and the way it is organized. Access to these data is usually provided by a "database management system" (DBMS) consisting of an integrated set of computer software that allows [users](https://en.wikipedia.org/wiki/User_(computing)) to interact with one or more databases and provides access to all of the data contained in the database (although restrictions may exist that limit access to particular data). The DBMS provides various functions that allow entry, storage and retrieval of large quantities of information and provides ways to manage how that information is organized.

**Online shopping** (sometimes known as **e-tail** from "electronic retail" or **e-shopping**) is a form of electronic commerce which allows consumers to directly buy goods or services from a seller over the Internet using a web browser. Alternative names are: e-web-store, e-shop, e-store, Internet shop, web-shop, web-store, online store, online storefront and virtual store. Mobile commerce (or m-commerce) describes purchasing from an online retailer's mobile optimized online site or app.

An online shop evokes the physical analogy of buying products or services at a bricks-and-mortar retailer or shopping center; the process is called business-to-consumer (B2C) online shopping. In the case where a business buys from another business, the process is called business-to-business (B2B) online shopping. The largest of these online retailing corporations are Alibaba,Amazon.com, and eBay.

# Problem Specification

Homework 2

Consider an application **OrderManagement** for processing customer orders. The application uses (minimally) the following classes: **Order**, **OPDept** (Order Processing Department), **Customer**, **Product**, and **Warehouse**. The classes **OPDept** and **Warehouse** use a **BinarySearchTree** for storing orders.

**Cerinte pentru nota 5:**

- Formular pentru operatii pe client: introducere client nou, editare detalii client, stergere client, vizualizare clienti in format tabelar (JTable).

- Formular pentru operatii pe produs: introducere produs nou, editare detalii produs, stergere produs, vizualizare produse in format tabelar (JTable).

- Optiunea de cumparare produse de catre un client. Se va oferi posibilitatea selectarii unui **client existent** si a **unui produs existent**. Se va introduce **cantitatea** dorita din produsul selectat. In cazul in care nu exista suficiente produse (**UNDERSTOCK**) se va afisa un mesaj de eroare.

- Documentatie

**Cerinte nota 6:**

- Toate cerintele pentru nota 5

- Crearea unei facturi cu detaliile de cumparare a unui produs de catre un client intr-un fisier (.txt sau .pdf)

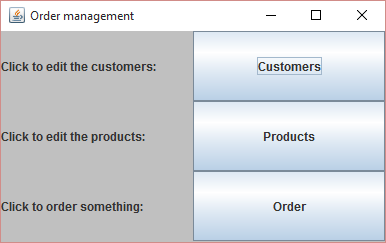
**Cerinte nota 7:**

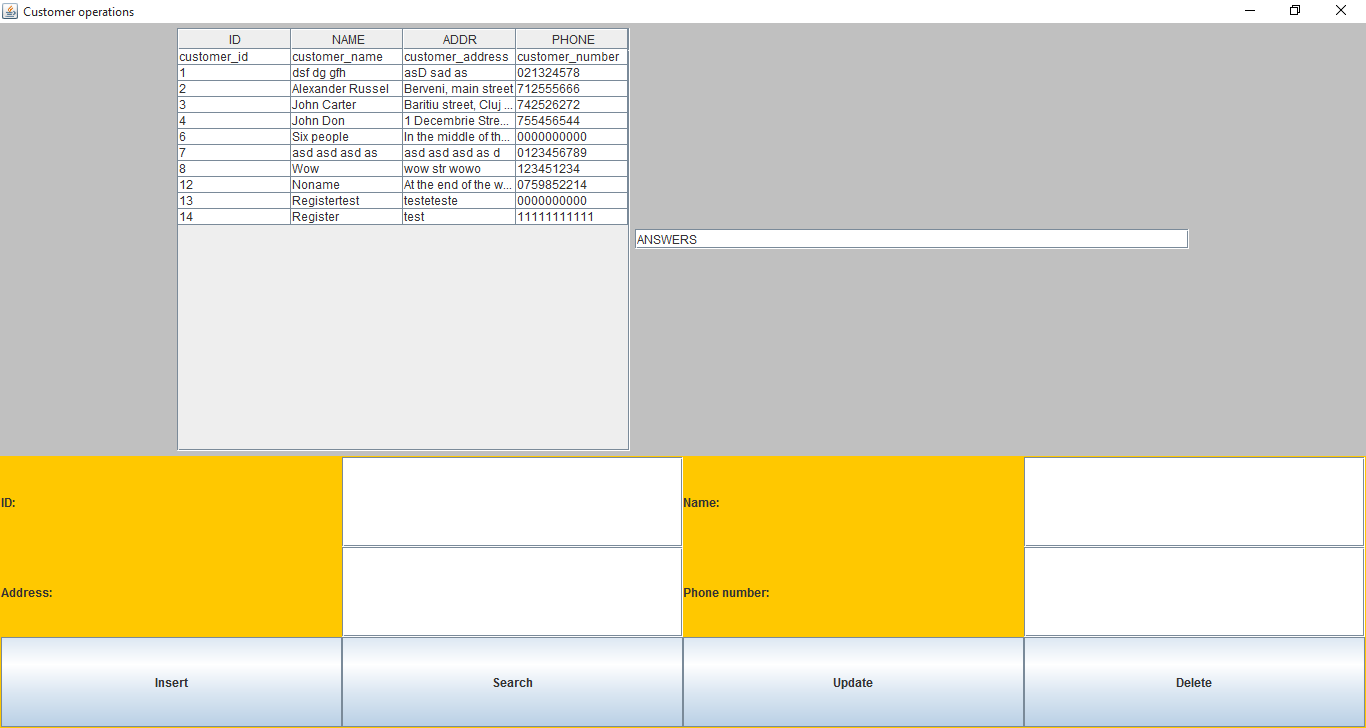
- Inlocuirea structurii de **BinarySearchTree** cu **baza de date relationala** cu minim 3 tabele.

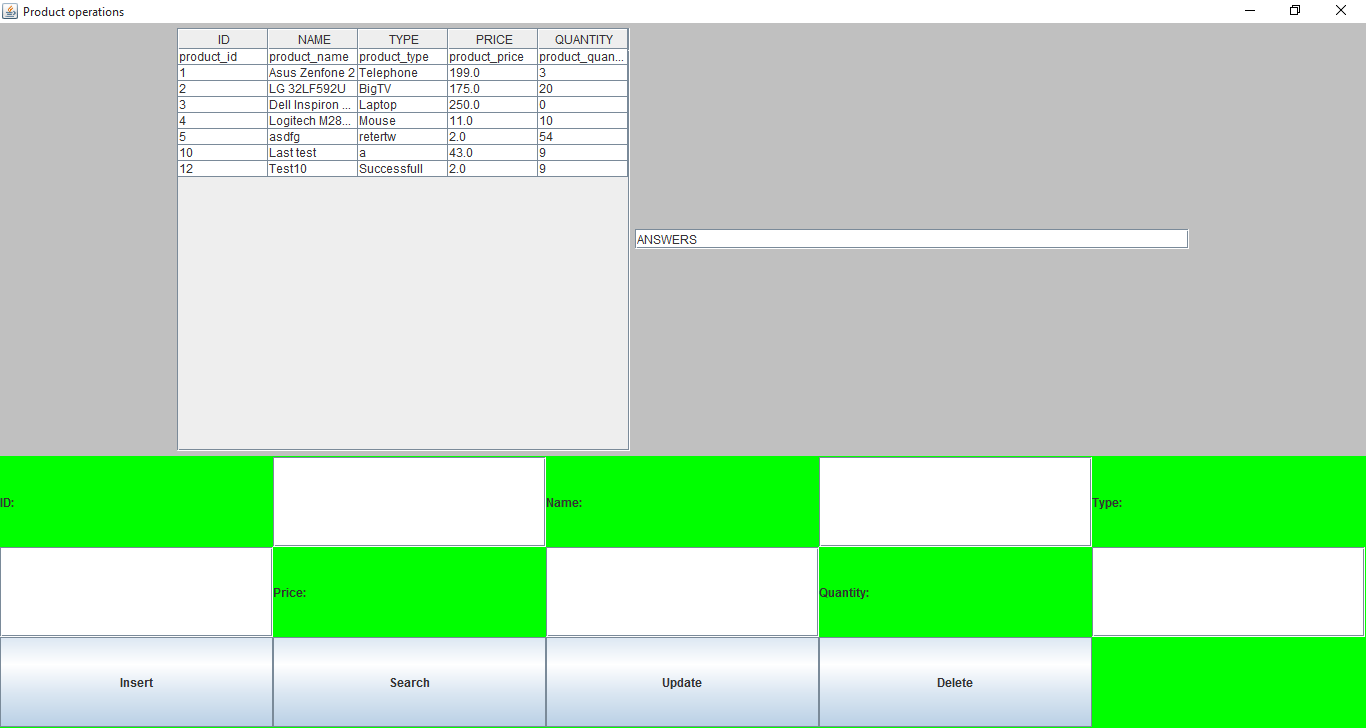
**Cerinte nota >7:**

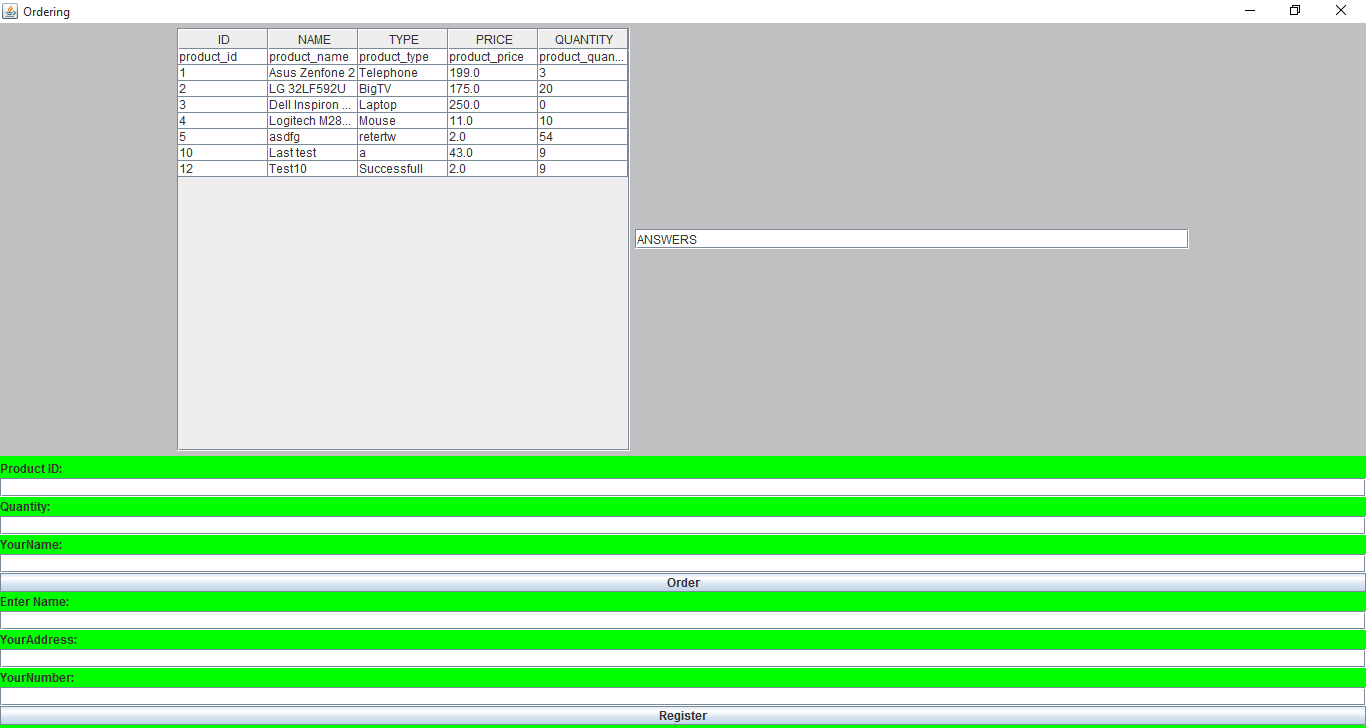
- Arhitectura **LAYERS** a aplicatiei discutata la laborator (existenta a minim 4 pachete: **DataLayer** cu clase DAO pentru fiecare tabel, **Model** cu clasele entitati corespunzatoare fiecarul tabel, **BusinessLogic** cu clasele de logica a aplicatiei si pachetul de **Presentation** pentru interfata grafica)

# Example of working









The program will simulate actually a web-shop. A client can launch orders, and the program stores data about it. The user can register. This is necessary because he/she needs to add some personal data.

# Design

## Relational Diagram

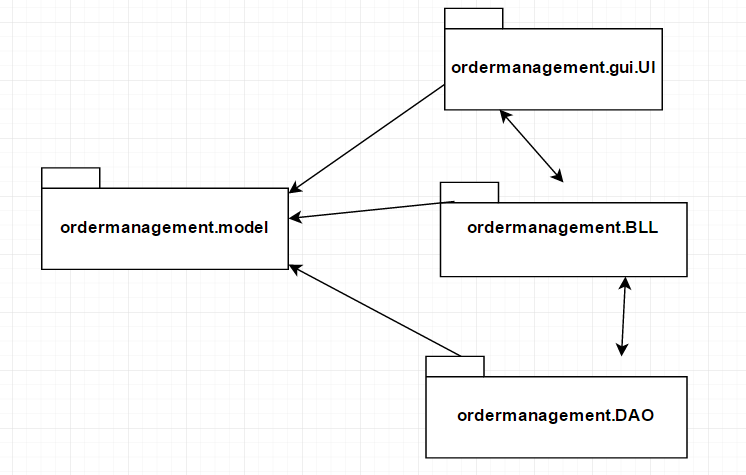
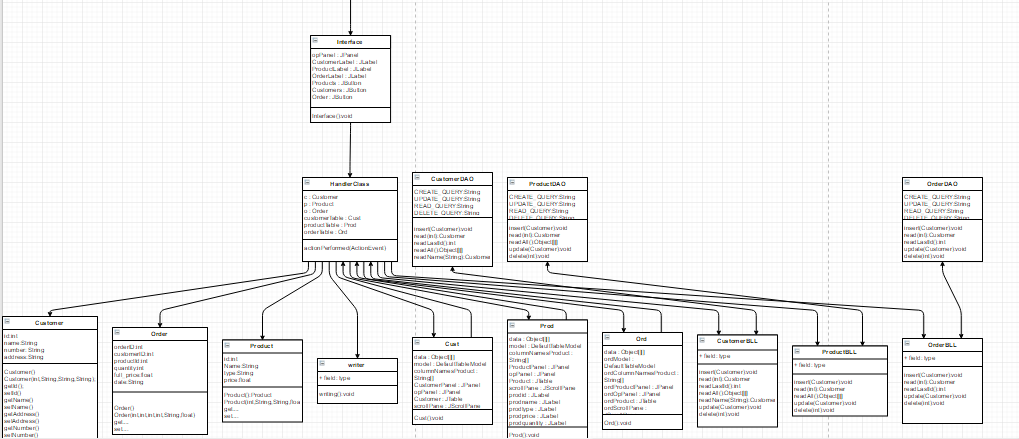
To resolve the problem, I’ve chosen to use four different packages, so that the design of application would be easier. Every package contains the classes which has strict semantical relation with it. For example the package: ordermanagement.DAO has inside it the classes *ProductDAO, CustomerDAO* and OrderDAOclasses. Their names and dependencies are presented below:

Figure 1 Package diagram

From the picture above you can see the relationships between pakcages: all packages have relations of "dependency" between each other, which means that one’s class uses another’s or uses an object or a method of another’s class. The packeges has the following classes each:

* ordermanagement.DAO: *ProductDAO, CustomerDAO* and OrderDAO
* ordermanagement.BLL: *ProductBLL, CustomerBLL* and OrderBLL
* ordermanagement.gui.UI: Cust, Ord, Prod, Interface, Writer, HandlerClass
* ordermanagement.model: Customer, Order, Product, OrderStart

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

## Class Design

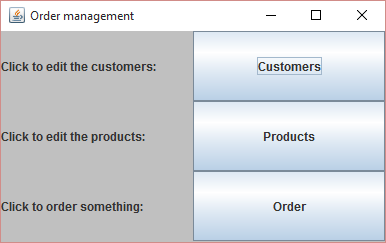
1. *Interface* Class : public class Interface extends JFrame

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

In this class there is only a constructor, by wich is realisied the creation of a container. To this container there are added more panels: opPanel. The previous one is responsible for containing -3- JTextButtons in which the user may use to achieve some actions, ( opening new windows, closing them, creating things ) The actions are done immediately, and with a good implementation of the graphical user interface, they can navigate through windows, and perform easily the desired actions. 3 JLabels are indexing the way! (If somebody reads them I’m sure he or she will find what he or she wants ).

The attributes if the Interface Class:

To achieve the desired Interface I needed many attributes of different types:

* A JFrame: it is created by extending the JFrame
* Panels:
  + Private opPanel - contains all the buttons necessary which by the user interacting with the program can understand what he or she wants and execute the corresponding operation(s).
* 3 Buttons for executing the desired operations by the user/admin:
  + Public static JButton Cusomers;
  + Public static JButton Products;
  + Public static JButton Order;

Constructor:

* + Public Interface() {…} – the initializations, coloring, placing(layout) are done here.

The following elements will be created: frame, textfields, buttons, labels.

* Creating a new HandlerClass object:
  + HandlerClass handler = new HandlerClass();

Every button is added to and ActionListener: Customers.addActionListener(handler);

Products.addActionListener(handler); etc.

1. *HandlerClass* class: public class HandlerClass implements ActionListener;

This class was created to keep track of the user activity, and do operations corresponding to the user will. This class implements ActionListener class: public class HandlerClass implements ActionListener; It was required that I import the following classes: Order, Product, Cusotmer, in order to be able execute the operations according to the user input.

This class contains:

**if** (event . getSource() == Interface . *Customers*)

**if** (event . getSource() == Interface . *Products*)

**if** (event . getSource() == Interface . *Order*)

**if** (event . getSource() == Cust . *insert*)

**if** (event . getSource() == Cust . *update*)

**if** (event . getSource() == Cust . *search*)

**if** (event . getSource() == Cust . *delete*)

**if** (event . getSource() == Prod . *insert*)

**if** (event . getSource() == Prod . *update*)

**if** (event . getSource() == Prod . *search*)

**if** (event . getSource() == Prod . *delete*)

**if** (event .getSource() == ordermanagement . gui . UI . Ord . *Order*)

**if** (event . getSource() == Ord . *Register*)

Every part is playing a very important role in realizing the graphical user interface, and it’s communication with the user.

* Products objects( attributes ):
  + Public Product p = new Product();
  + Public Product p = new Product(int,String,String,String);

The user input will be stored in one (mainly the second) of these two objects, so I the program can work with them.

1. *ProductDAO* class

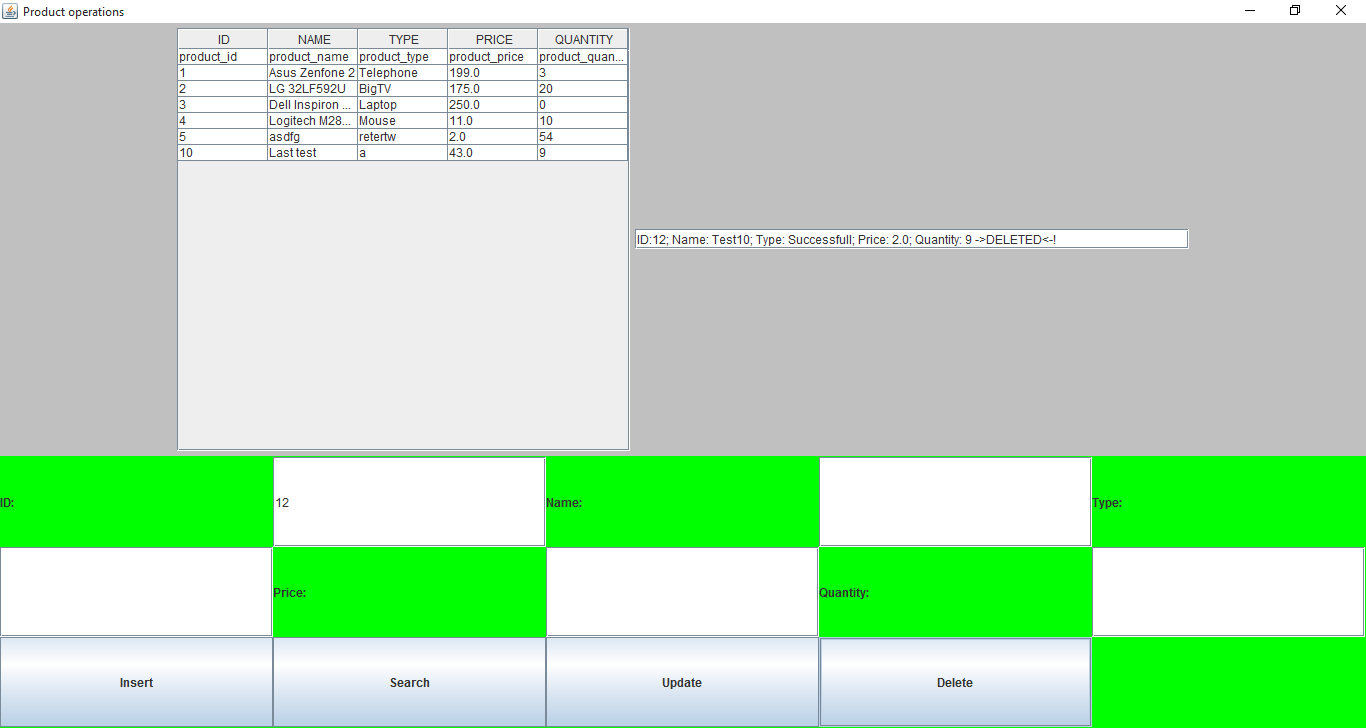
This class which communicates with the database. With more accuracy the Product table from the database, which has 3 tables! Product, Order, Customer. This and only this class is allowed to communicate with the Product table part of the database, it is more safe, encapsulated and well structured this way.

In this class there is:

* 2 CONSTRUCTORS
* Implemented CRUD operations:
  + Public static int readAll()

This method reads everything from the database Product table and gives this information further to the ProductBLL class, where it is checked if the data is correct, and it is checked that too if the input data and the readRequest from the user is all good! This method gets nothing as arguments, but returns a result set in which all the rows from the Product table is found. This data is given further to the BLL class from where it is given even further to the interface part. There all data is displayed in a JTable, where the user can finally see what is actually in the database Product table.

* + Public static boolean delete(int id)

This method gets one argument which is the ID. Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***DELETE\_QUERY*** = "DELETE FROM product WHERE product\_id = ?"; In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the ProductBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

It says “ID:12; Name: Test10; Type: Successfull; Price: 2.0; Quantity: 9 ->DELETED<-!”

* + Public static Product read(int id)

This method gets one integer type argument. It is again an ID, which represents a Product’s id from the database, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***READ\_QUERY*** = "SELECT \* FROM product WHERE product\_id = ?;"; In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the ProductBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

ID:10; Name: Last test; Type: a; Price: 43.0; Quantity: 9

Public static void insert (Product p)

This method gets one Product type argument. It is a product, which represents a Product from the Product class, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***CREATE\_QUERY*** = "INSERT INTO product (product\_id, product\_name,product\_type,product\_price,product\_quantity) "

+ "VALUES (?,?,?,?,?);";

In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the ProductBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

Inserted!!!

* + Public static boolean update(Product p)

This method gets one Product type argument. It is a product, which represents a Product from the Product class, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String **private** **static** **final** String ***UPDATE\_QUERY*** = "UPDATE product SET product\_name=?, product\_type=?,product\_price=?,product\_quantity=?" + " WHERE product\_id = ?;";

In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the ProductBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

* Inserted!!!

1. *CustomerDAO* class

This class which communicates with the database. With more accuracy the Product table from the database, which has 3 tables! Product, Order, Customer. This and only this class is allowed to communicate with the Product table part of the database, it is more safe, encapsulated and well structured this way.

In this class there is:

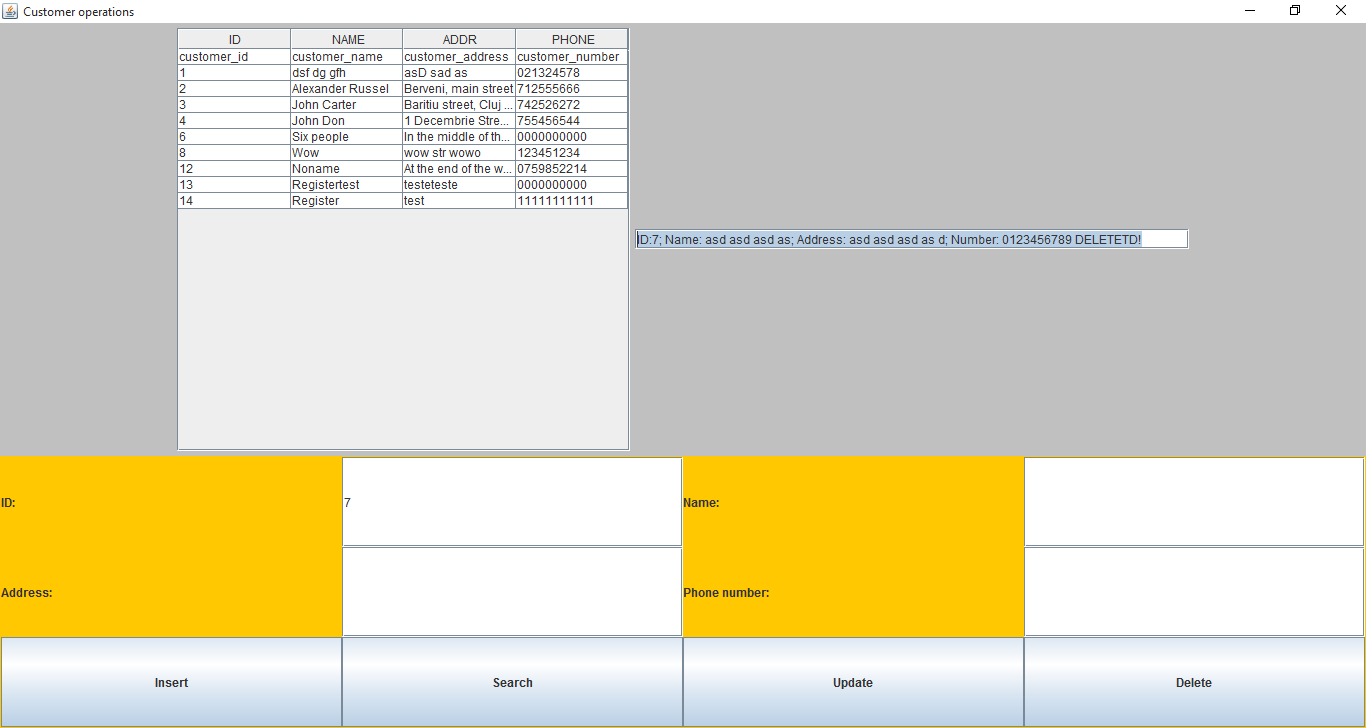
* 2 CONSTRUCTORS
* Implemented CRUD operations:
  + Public static int readAll()

This method reads everything from the database Product table and gives this information further to the CustomerBLL class, where it is checked if the data is correct, and it is checked that too if the input data and the readRequest from the user is all good! This method gets nothing as arguments, but returns a result set in which all the rows from the Customer table is found. This data is given further to the BLL class from where it is given even further to the interface part. There all data is displayed in a JTable, where the user can finally see what is actually in the database Product table.

* + Public static boolean delete(int id)

This method gets one argument which is the ID. Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***DELETE\_QUERY*** = "DELETE FROM customer WHERE customer\_id = ?"; In the database it executes the query, and returns a Customer type object, which contains one row’s data. This object is passed to the CustomerBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

ID:7; Name: asd asd asd as; Address: asd asd asd as d; Number: 0123456789 DELETETD!



* + Public static Customerread(int id)

This method gets one integer type argument. It is again an ID, which represents a Customer’s id from the database, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***READ\_QUERY*** = "SELECT \* FROM CustomerWHERE customer\_id = ?;"; In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the CustomerBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

ID:10; Name: Last test; Type: a; Price: 43.0; Quantity: 9

Public static void insert (Customer p)

This method gets one Product type argument. It is a product, which represents a Product from the Product class, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***CREATE\_QUERY*** = "INSERT INTO customer (customer\_id, customer\_name,customer\_type,customer\_price,customer\_quantity) "

+ "VALUES (?,?,?,?,?);";

In the database it executes the query, and returns a Customer type object, which contains one row’s data. This object is passed to the CustomerBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

Inserted!!!

* + Public static boolean update(Customer p)

This method gets one Product type argument. It is a product, which represents a Product from the Product class, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String **private** **static** **final** String ***UPDATE\_QUERY*** = "UPDATE customer SET customer\_name=?, cusotmer\_type=?,cusotmer\_price=?,customer\_quantity=?" + " WHERE cusotmer\_id = ?;";

In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the CustomerBLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

* Inserted!!!

1. *OrderDAO* class

This class which communicates with the database. With more accuracy the Product table from the database, which has 3 tables! Product, Order, Customer. This and only this class is allowed to communicate with the Product table part of the database, it is more safe, encapsulated and well structured this way.

In this class there is:

* 2 CONSTRUCTORS
* Implemented CRUD operations:
  + Public static int readAll()

This method reads everything from the database Product table and gives this information further to the *Order*BLL class, where it is checked if the data is correct, and it is checked that too if the input data and the readRequest from the user is all good! This method gets nothing as arguments, but returns a result set in which all the rows from the *Order* table is found. This data is given further to the *Order*BLL class from where it is given even further to the interface part. There all data is displayed in a JTable, where the user can finally see what is actually in the database Product table.

* + Public static boolean delete(int id)

This method gets one argument which is the ID. Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***DELETE\_QUERY*** = "DELETE FROM customer WHERE order\_id = ?"; In the database it executes the query, and returns a Customer type object, which contains one row’s data. This object is passed to the *Order*BLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

ID:7; Name: asd asd asd as; Address: asd asd asd as d; Number: 0123456789 DELETETD!

* + Public static *Order* read(int id)

This method gets one integer type argument. It is again an ID, which represents a Customer’s id from the database, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***READ\_QUERY*** = "SELECT \* FROM Order WHERE order\_id = ?;"; In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the *Order*BLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

Public static void insert (Order o)

This method gets one Product type argument. It is a product, which represents a Product from the Product class, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String ***CREATE\_QUERY*** = "INSERT INTO order (order\_id, order\_name,order\_type, order\_price, order\_quantity) "

+ "VALUES (?,?,?,?,?);";

In the database it executes the query, and returns a order type object, which contains one row’s data. This object is passed to the *Order*BLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

Inserted!!!

* + Public static boolean update(*Order* p)

This method gets one Product type argument. It is a product, which represents a Product from the Product class, Again the method estabilishes a connection and tries to connect to the database. If it’s successful that it launches a predefined query, which is **private** **static** **final** String **private** **static** **final** String ***UPDATE\_QUERY*** = "UPDATE order SET order\_name=?, order\_type=?,order\_price=?,order\_quantity=?" + " WHERE order\_id = ?;";

In the database it executes the query, and returns a Product type object, which contains one row’s data. This object is passed to the *Order*BLL class, where again the checking method runs through it, and if there is no problem, the data is feeded to the graphical user interface .

* Inserted!!!

1. *CustomerBLL, ProductBLL, OrderBLL*

*These classes are the middle area of the three level database access system. These classes, communicate with the graphical user interface and the DAO classes, and they are responsible for that no garbage data enters in the database.*

*All the three are the same with the exception that every BLL class communicates with its DAO class, and almost with every interface component.*

1. *Interface classe: Ord, Prod, Cust, write, HandlerClass, Interface*

These classes are responsible for the communication with the user. They have inside them Buttons, Panels, TextFields, JScrollPanes. There are a lot of them and Ord, Cust, Prod are the same, the just communicate with different BLL and DAO classes.

## Use Case Diagramm

## Packages and Interfaces

A Java package is a mechanism for organizing Java classes into namespaces. Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality. A package provides a unique namespace for the types it contains. Classes in the same package can access each other's package-access members.

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

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

* import java.awt: Contains all of the classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components.
  + java.awt.BorderLayout: A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center.
  + java.awt.Color: The Color class is used encapsulate colors in the default RGB color space or colors in arbitrary color spaces identified by a [ColorSpace](http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/color/ColorSpace.html).
  + java.awt.Dimension: This encapsulates the width and height of a component (in integer precision) in a single object.
  + java.awt.GridLayout: The GridLayout class is a layout manager that lays out a container's components in a rectangular grid for a better view of all the buttons and textfields which are added to the main panel.
* import java.awt.event
  + java.awt.event.ActionEvent;
  + java.awt.event.ActionListener;
* import javax.swing: Typical Swing applications do processing in response to an event generated from a user gesture. For example, clicking on a JButton notifies all ActionListeners added to the JButton. That’s why we use this package for creating the user interface Gui.
  + javax.swing.JButton;
  + javax.swing.JFrame; javax.swing.JLabel; javax.swing.JPanel;

## Using and testing the application

In order to use the application open Homework2. This will open a window which generates the Gui class. Thus the user can enter the desired values and selecting the operations by pressing one of the 6 buttons.

Rules for the input data:

You should Start the program and click on a button! After this it is possible to manipulate Customer or Product tables, or Order something from the products. It is important that one order contains only one product type, but the quantity can be more!

If you follow the isntructions you should be able to figure out everything how should be done!

* Before executing the insert, delete, update, search, searchName, searchAll operations there must be everything set, which means one should enter the correct data in every textField to obtain a good working process with this program.

# Conclusions

Achieving such a program may be hard both in terms of algorithms, graphical structure. Although some instructions were easy for implementation, like the CRUD ( Create , Remove, Update, Delete )operations were difficult and some problems in implementation occurred.

Moreover, to read and display an ordered result of the users actions took a many special function, actionListeners, Action handlers to transform the data type which in a way that the database can easily understand and operate with.

For a better performance there should be implemented all cases where exceptions can occur and the application stops working due to an error made ​​by the user. It could handle more operations, for example create a user account, get notifications for new products, delete user account, and implement Admin rights, which are separated from all users. Another thing that could be improved is the display so that it would be more elegant.

It was a really good practice, which taught me many things about this kind of programming.

# References

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

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

*Teacher(s)*