Programming Techniques

Homework 3

Database Application

1. **Objective:**

Problem Specification:

Design and implement an order management application for processing customer orders for a warehouse.

The objective of this program is to implement a database management application that is capable of manipulating the tables by inserting, deleteing, updating and searching, together with a simple graphic user interface and some additional functionalities (filtering, bill creation, etc).

1. **Analysis, Modelling, Use Cases, Scenarios:**

2.1. Analysis:

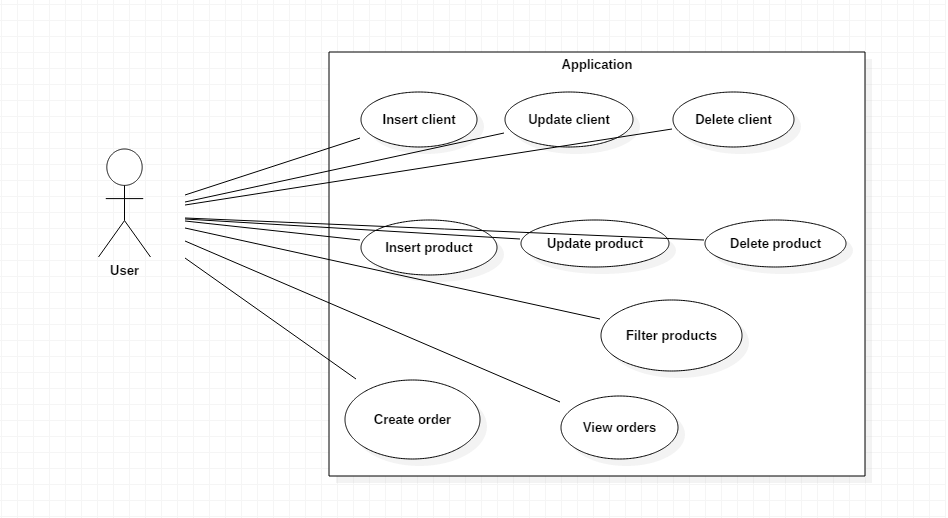
The program is a basic database administration application. It allows the user to manage the clients (add, update, delete), the products (add, update, delete) and the orders (create new orders). Everything is stored in a database and the data is available even after the program is restarted. The data is displayed neatly in tables and it can be easily accessed and modified. The application can be useful for warehouse managers and administrators.

2.2. Modelling:

The approach I have chosen for modelling this problem involves the creation and use of a layered architecture. This involves the use of three layers: the data access layer, which works directly with the database, the bussiness logic layer, which is concerned with manipulating the data from the database and from the user, and the presentation layer, which is used for communicating with the user. Additionally, all the classes in these layers use some classes that represent the entities from the database, classes which are used to simplify the process of retrieving data and updating the database.

2.3. Use cases:

The main (and only) actor in the for this program is the user, as per the definition of the use case. The user has all the capabilities of the program at hand, being able to work with the clients, the products or the orders. For clients and products, the user can perform insertion, deletion and update (and also filtering the products). For orders, the user can create new orders and see the existing ones. The use cases are better illustrated by the following use case diagram:



2.4. Scenarios:

Scenarios are sequences of steps that represent the interaction between the system and an actor.

**Scenario 1**

1. Identification Summary

Title: Insert product.

Summary: This use case allows the user to insert a new product in the database.

1. Flow of events
2. The user starts the application
3. The user pushes the „Manage Products” button
4. The user inputs in the provided text fields the data related to the new product, also selecting the desired option from the combo box
5. The user pushes the „Insert new” button
6. The application checks the corectness of the input data
7. The application inserts the new product in the database
8. The user pushes the „Refresh table” button in order to see the changes in the table

Alternative scenarios:

1. The input data is not correct or there is no input: could appear at step 5
2. The application notifies the user of the incorrect input with a message.

The scenario reverts to step 3

1. A database error occurs: could appear at step 6
2. The application notifies the user of the incorrect input with a message.

The scenario reverts to step 3

**Scenario 2**

1. Identification Summary

Title: Update product.

Summary: This use case allows the user to update a product in the database.

1. Flow of events
2. The user starts the application
3. The user pushes the „Manage Products” button
4. The user selects a row from the table
5. The user changes in the provided text fields the data related to the new product, also selecting the desired option from the combo box
6. The user pushes the „Update” button
7. The application checks the corectness of the input data
8. The application inserts the new product in the database
9. The user pushes the „Refresh table” button in order to see the changes in the table

Alternative scenarios:

1. The input data is not correct or there is no input: could appear at step 6

7. The application notifies the user of the incorrect input with a message.

The scenario reverts to step 4

1. A database error occurs: could appear at step 7

8. The application notifies the user of the incorrect input with a message.

The scenario reverts to step 3

**Scenario 3**

1. Identification Summary

Title: Filter products.

Summary: This use case allows the user to filter the products shown in the table.

1. Flow of events
2. The user starts the application
3. The user pushes the „Manage Products” button
4. The user selects a radio button
5. The user inputs in the provided text fields the data related to the filtering criteria if this is the case
6. The user pushes the „Refresh table” button in order to see the filtered content in the table

Alternative scenarios:

i. A database error occurs: could appear at step 5

6. The application notifies the user of the incorrect input with a message.

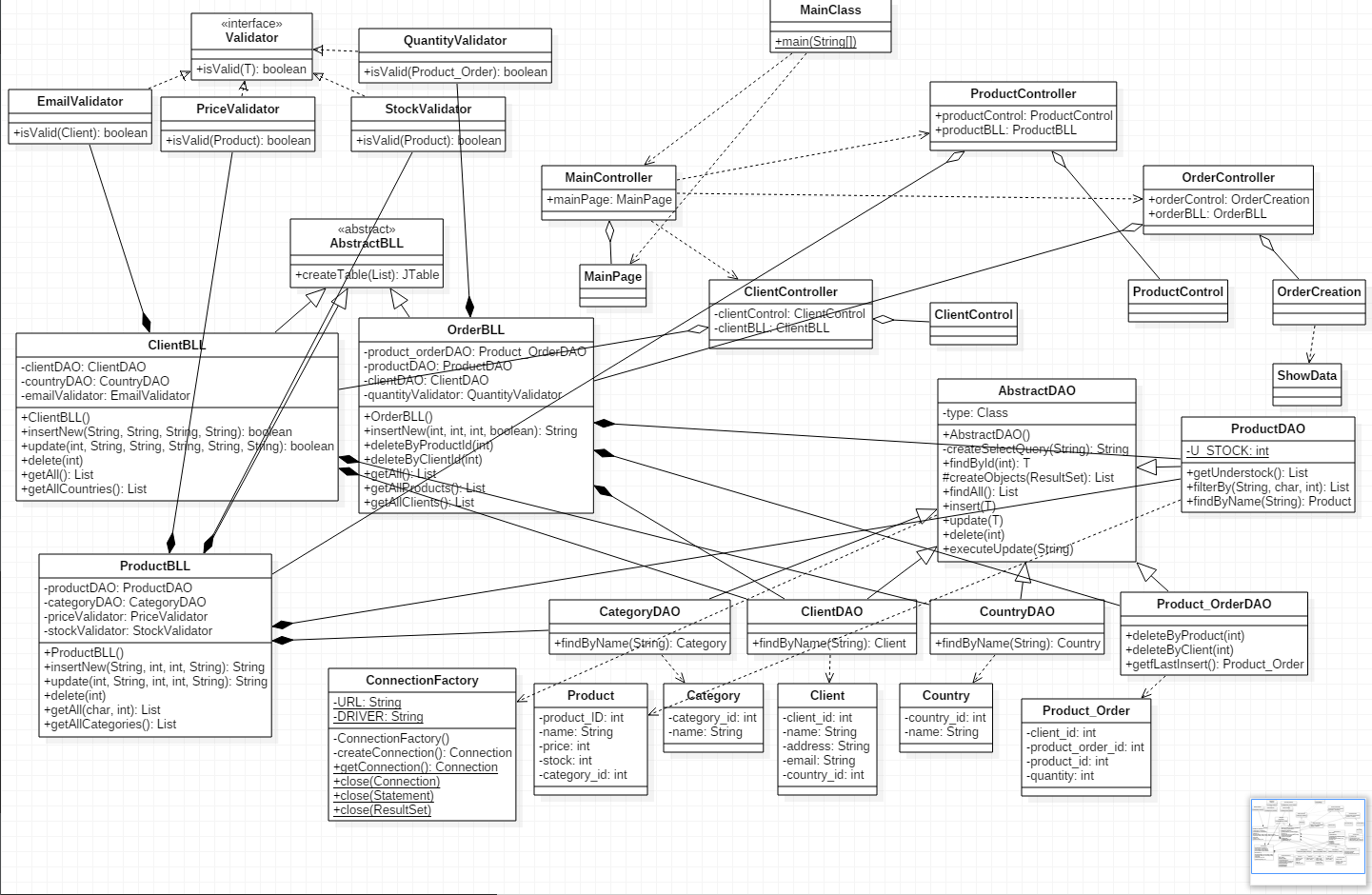
The scenario reverts to step 4

1. **Design**

2.1. Design decisions

The application uses both the Model-View-Controller architecture and the 3-tier layered architecture. Thus, from the MVC point of view, there is a separation between the problem Model (the classes concerned with solving the problem in a corect manner), the View (the classes concerned with the GUI: the parts which must be displayed on the screen, the interface between the user and the applications: text fields, text areas, buttons) and the Controller (the class(es) which make possible for the View classes to send messages to the Model classes and by this allow the user to control the the application). From the layered architecture point of view, the Controller and the View are part of the top layer, while the Model actually contains the other two layers: bussiness logic and data acess and also the actual model of the database.

2.2. UML Class Diagram



2.3. Data Structures

The two main data structures that are used in the application are *arrays* and *lists*:

1. ***Lists*** and namely the *Vector* and *ArrayList* implementation of the interface are used for arrays of variable (and often changing) length, because they are easier to use and have already-implemented methods for modifying the size and the required memory, thus being more efficient. They are used to model the lists of objects returned from the database by various queries.

2.4. Class Design

The application consists of the following classes:

* **AbstractDAO:** this abstract class is concerned with accessing the database. Through its method it can perform data retrievals, updates, insertions and deletions. Its methods work for any type of data because it uses reflection techniques. This class is extended by classes specific for each table, which implement other methods which require database access, like search for a specific name. These classes are: CategoryDAO, ClientDAO, CountryDAO, Product\_OrderDAO, ProductDAO.
* **AbstractBLL:** this abstract class is used as a base for the business logic level classes specific for each operation. It contains a common method that creates a JTable starting from a list of object by using reflection techniques. This class is extended by ClientBLL, ProductBLL, OrderBLL, which all have specific methods for dealing with their tables in the database. These subclasses work with the DAO classes, having methods to insert, update, delete and retrieve data.
* **ConnectionFactory:** this class is built in accordance with the Singleton design pattern. The idea is that only one object of this class can exist at any moment, and the constructor is private so the object can only be instantiated by calling static methods of this class. This class is used to open and close the database connections and return them to be used for accessing the database.
* **View** classes**:** these classes represent the graphical interface between the user and the application. The classes are MainPage (the starting window), ClientControl (the client administration window), ProductControl (the product administration window), OrderCreation (the window used to create new orders) and ShowData (a window used to show a table on screen).
* **Controller** classes**:** theses classes are parte of the MVC architecture. Through the use of inner classes, they provide the graphic interface with their functionalities, by matching the ActionListeners with methods from the BLL classes. These classes are: MainController (controls MainPage), ClientController (controls ClientControl with ClientBLL), ProductController (controls ProductControl with ProductBLL) and OrderController (controls OrderCreation with OrderBLL).
* **Validator** classes**:** objects of this class are used to validate the data before it is inserted in the database. They implement the **Validator** interface. Each BLL class has some Validator classes associated with it.
* **Category, Client, Country, Product\_Order, Product:** these classes match perfectly the entities from the database. They are used to facilitate data manipulation: objects of this classes are returned by methods that query the database, and also objects of theses classes are used by the methods which update the database. These classes do not offer other functionalities apart from the part of the Model-View-Controller architecture, this class is used to create the Queue objects, assign a QueueStatistics object to each one, create the ClientProducer object, assign a ShopStatistics object to it, start all the threads and after their execution finishes, send the resulting information to be printed.
* **MainClass:** this class is used to start the application through its main() method.

2.5. Interfaces

* **Validator:** this interface is used to validate the objects to be inserted in the database so that they do not create database errors.

2.6. Relationships

There are of course a lot of relationships between the classes. The following section presents the most important ones:

* (Main/Proudct/Order/Client)Controller (aggregation) MainPage/ ProductControl/OrderCreation/ClientControl, /ProductBLL/OrderBLL/ ClientBLL: The ()Controller receives its View and its BLL in the constructor: the latter is used to implement the action listeners for the former
* BLL (composition) DAO: Objects of the BLL classes have DAO objects as their instance variables but the objects are created inside the constructor, not outside it. The DAO objects are used to implement the BLL methods.
* AbstractDAO (dependency) ConnectionFactory: The AbstractDAO class uses methods from the ConnectionFactory class, but not objects.
* BLL (composition) Validator: The BLL classes have Validator objects as their instance variables but the objects are created inside the constructor, not outside it.

2.7. Packages

The classes are organized in eight packages: **controller** pack (the Controller objects), **main** pack (the MainClass), **model** pack (the classes which model the database), **view** pack (the GUI classes), **dao** pack (the Data Acess classes), **connection** pack (the ConnectionFactory), **bll** (the Business Logic Level classes), **bll.validators** (the Validator interface and its implementations).

2.8. Algorithms

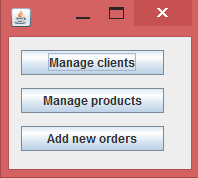
At the database access level, the results of each query are created using reflection techniques, creating objects by using their Class objects and building them piece by piece by setting each field. This is also how the insert and update queries are built, by iterating through the fields of a given object. Deletion is done based on the an integer id. Reflection is also used to build table from lists of objects: their fields become the table’s fields, each object becomes a row in the table.

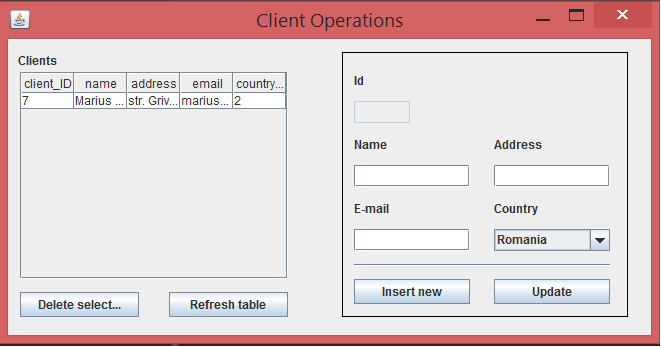
At the business logic layer, each class uses different DAO objects to retrieve the needed data or to insert/update/delete. At this level the objects to be inserted/updated are created and sent to the DAO objects.

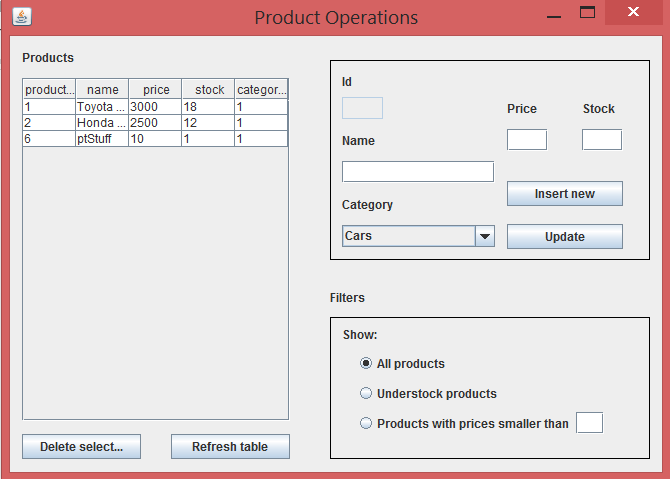
The program is pretty straightforward from the algorithmic point of view.

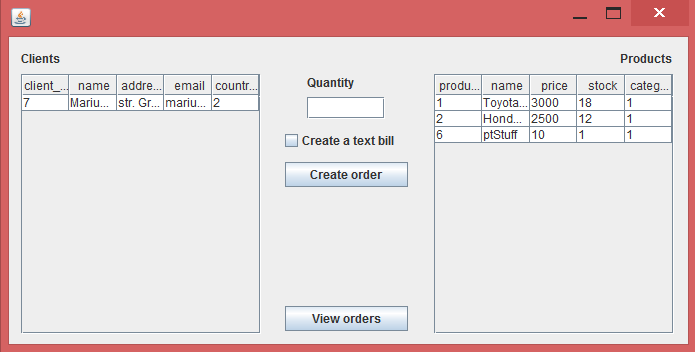
2.9. User Interface

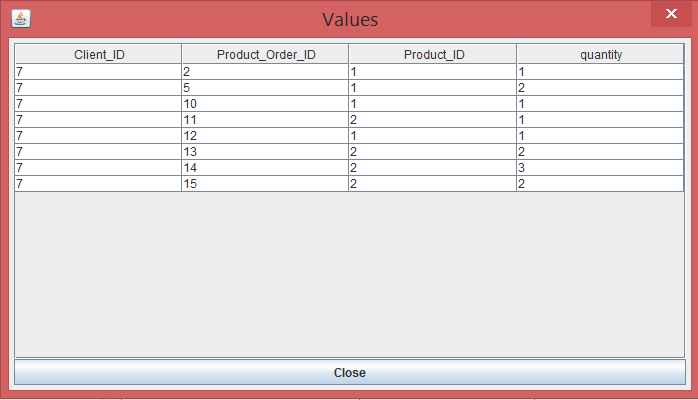
The windows of the user interface are built using the WindowBuilder Eclipse plugin. The use of the interface is pretty self-explanatory. The main window:



The Client management window, together with the table showing the contents of the database.

The Product management window, together with the table showing the contents of the database.

The Order insertion window, together with two tables showing the contents of the database.

The window which shows the Orders:

1. **Implementation and testing**

4.1 Implementation (also includes some of the more interesting code segments)

* The **AbstractDAO<T>** abstract class: has a type (Class<T>) as instance variable. Methods:
  + Constructor: assigns to the type a Class coresponding to the generic class used
  + createSelectQuery: has a String as parametera, it creates a query to be used on the database based on the String given as parameter which correesponds to a field in the database, if the input parameter is null, it creates a query to retrieve everything from the table;
  + findByID: has an int as parameter, it returns the object corresponding to the value from the database with the given input id;

**public** **T** **findByID**(**int** id) **throws** **Exception**{

Connection **con**=**null**;

PreparedStatement **pS**=**null**;

ResultSet **rS**=**null**;

**String** **query**=createSelectQuery(type.getSimpleName()+"\_ID");

con=**ConnectionFactory**.*getConnection*();

**try** {

pS = con.prepareStatement(query);

pS.setInt(1, id);

rS = pS.executeQuery();

**return** createObjects(rS).get(0);

} **catch** (**Exception** **e**) {

**throw** e;

} **finally**{

**ConnectionFactory**.*close*(con);

**ConnectionFactory**.*close*(pS);

**ConnectionFactory**.*close*(rS);

}

}

* + createObjects: has a ResultSet as parameter, it uses reflection techniques to create a List of objects based on the type instace variable and on the ResultSet;

**protected** List<**T**> **createObjects**(ResultSet rS) **throws** **Exception**{

List<**T**> **list**=**new** ArrayList<**T**>();

**try** {

**while**(rS.next()){

**T** **instance**=type.newInstance();

**for**(**Field** **field**:type.getDeclaredFields()){

**Object** **value**=rS.getObject(field.getName());

**PropertyDescriptor** **pD**=**new** PropertyDescriptor(field.getName(), type);

**Method** **method**=pD.getWriteMethod();

method.invoke(instance, value);

}

list.add(instance);

}

**return** list;

} **catch** (**Exception** **e**) {

**throw** e;

}

}

* + findAll: it returns a List of objects corresponding to all the rows from a table from the database;
  + insert: has a object T as parameters, it inserts the given object in the database through the use of reflection techniques;
  + update: has a object T as parameters, it updates the given object in the database through the use of reflection techniques;

**public** **void** **insert**(**T** t) **throws** **Exception**{

**String** **insertQuery**="Insert into "+type.getSimpleName()+" values (";

**try** {

**Object** **value**=**null**;

**for** (**Field** **field** : type.getDeclaredFields()) {

**if**(field.getName().equalsIgnoreCase(type.getSimpleName()+"\_ID")){

**continue**;

}

**PropertyDescriptor** **pD** = **new** PropertyDescriptor(field.getName(), type);

**Method** **method** = pD.getReadMethod();

value=method.invoke(t);

insertQuery += "\'" + value + "\',";

}

insertQuery = insertQuery.substring(0, insertQuery.length() - 1);

insertQuery += ")";

executeUpdate(insertQuery);

} **catch** (**Exception** **e**) {

**throw** e;

}

}

* + delete: has an id as parameter (int), it deletes from the database the entry with the given id;
  + executeUpdate: has a String as parameter, it executes the query given as parameter on the database (insert/update/delete)
* The **DAO** classes: extend the **AbstractDAO** class by adding specific methods:

**public** **Country** **findByName**(**String** name) **throws** **Exception**{

**String** **query**="Select \* from Country where name=\'"+name+"\'";

Connection **con**=**null**;

PreparedStatement **pS**=**null**;

ResultSet **rS**=**null**;

con=**ConnectionFactory**.*getConnection*();

**try** {

pS = con.prepareStatement(query);

rS = pS.executeQuery();

**return** createObjects(rS).get(0);

} **catch** (**Exception** **e**) {

**throw** e;

} **finally**{

**ConnectionFactory**.*close*(con);

**ConnectionFactory**.*close*(pS);

**ConnectionFactory**.*close*(rS);

}

}

* + findByName: has a String as parameter, it returns the entry in the database with the given name;
  + getUnderstock: it returns a list of Product objects that have their stock instance variable lower than the static field of the class;
* The Model classes **Client, Category, Country, Product, Product\_Order**: have different fields corresponding to the database fields and mehods for accessing and mutating these fields. Also have both a parameterless constructor and one with parameters.
* The **AbstractBLL** abstract class: contains a single method:
  + createJTable: has a List of objects as parameter, it returns a Jtable containing the given objects by iterating through their type using reflection techniques.

**public** **JTable** **createTable**(List<Object> objects){

**Class**<?> **type**=objects.get(0).getClass();

**Vector**<Object> **columnNames**=**new** Vector<Object>();

**for**(**Field** **field**:type.getDeclaredFields()){

columnNames.add(field.getName());

}

**Vector**<**Vector**<Object>> **data**=**new** Vector<**Vector**<Object>>();

**for**(**Object** **entry**:objects){

**Vector**<Object> **row**= **new** Vector<Object>();

**for**(**Field** **field**:type.getDeclaredFields()){

**try** {

**PropertyDescriptor** **pD**=**new** PropertyDescriptor(field.getName(), type);

**Method** **method**=pD.getReadMethod();

**Object** **value**=method.invoke(entry);

row.add(value);

} **catch** (**Exception** **e**) {

e.printStackTrace();

}

}

data.add(row);

}

**return** **new** JTable(data, columnNames);

}

* The **BLL** classes extend the **AbstractBLL** class by adding business specific methods. They have various DAO objects as instance variables and also Validator objects. Methods include:
  + Constructor: parameterless, each instance variable receives a new object created inside this method;
  + insertNew: has various parameters depending on the specifics of the BLL class. It is used to insert new objects in the database. The object is created and validated inside the method. If the object is not valid, the method returns an error message to be sent to the user. For foreign keys, this method does not use the foreign table id, but the name of the row, obtaining the name by using the findByName methods in the DAO objects;
  + update: has various parameters depending on the specifics of the BLL class. It is used to update objects in the database. The object is created and validated inside the method. If the object is not valid, the method returns an error message to be sent to the user;

**public** **boolean** **update**(**int** id, **String** name, **String** address, **String** email, **String** country) **throws** **Exception**{

countryDAO=**new** CountryDAO();

**try** {

**Country** **countryObj**=countryDAO.findByName(country);

**Client** **client**=**new** Client(id, name, address, email, countryObj.getCountry\_id());

**if**(!emailValidator.isValid(client)){

**return** **false**;

}

clientDAO.update(client);

} **catch** (**Exception** **e**) {

// **TODO** Auto-generated catch block

e.printStackTrace();

**throw** e;

}

**return** **true**;

}

* + delete: has an integer id as parameter, it is used to delete an entry from the database;
  + getAll: returns a List of all the entries from a table in the database;
* The **ConnectionFactory** class: Built in accordance with the Singleton design pattern: only one object of this class can exist at any given moment. Methods:
  + Constructor: is private, it opens the database connection;
  + createConnection: is private, it returns an object of type Connection;
  + getConnection: static, it returns the Connection created by createConnection;
  + close: can have a Connection, a ResultSet or a Statement as parameter, it closes the given object;
* The **MainPage, ClientControl, OrderCreation, ProductControl, ShowData** GUI classes were built with **WindowBuilder**, but they were heavily modified. The text fields, buttons and combo boxes were transformed in instance variables so that their values can be accessed from outside. Also, additional method were added:
  + Add()ActionListener: adds an action listener to a button;
  + Getters for the instance variables;
  + showData: has a String as parameter, it outputs a message dialog containing the String;
  + setTable: has a JTable as input parameter, it sets the table on the window to the input table, also adding a MouseAdapter to the table.

**public** **void** **setTable**(**JTable** newTable){

table=newTable;

**JScrollPane** **scrollPane** = **new** JScrollPane();

scrollPane.setBounds(12, 33, 268, 207);

contentPane.add(scrollPane);

scrollPane.setViewportView(table);

table.addMouseListener(**new** MouseAdapter(){

***@Override***

**public** **void** **mouseClicked**(**MouseEvent** evt) {

**int** **row** = table.rowAtPoint(evt.getPoint());

**if** (row >= 0) {

textField\_3.setText(table.getValueAt(row, 0).toString());

textField.setText((**String**)table.getValueAt(row, 1));

textField\_1.setText((**String**)table.getValueAt(row, 2));

textField\_2.setText((**String**)table.getValueAt(row, 3));

comboBox.setSelectedIndex((**Integer**)table.getValueAt(row, 4)-1);

}

}

});

}

* The **Controller** classes: have a BLL and a GUI object as instance parameters. Methods:
  + Constructor: sets the corresponding fields.
  + **ActionListener**(s): inner classes which implements the ActionListener interface. Methods:

**class** **UpdateActionListener** **implements** ActionListener{

**public** **void** **actionPerformed**(**ActionEvent** arg0) {

**try** {

**if**(!clientBLL.update(**Integer**.*parseInt*(clientControl.getTextField\_3().getText()),

clientControl.getTextField().getText(), clientControl.getTextField\_1().getText(),

clientControl.getTextField\_2().getText(), (**String**)clientControl.getComboBox().getSelectedItem())){

clientControl.showMessage("Invalid email address format!");

}

} **catch** (**Exception** **e**) {

clientControl.showMessage("Could not update, a database error ocurred!");

}

}

}

* + - actionPerformed: has an ActionEvent as parameter. The method retrieves data from the GUI and calls the methods from the BLL objects whenever the user pushes a button.
* The **Validator** classes: implement the *Validator* interface, are used to validate objects before they are inserted in the database. Method:
  + isValid: has an object as parameter, returns true if the object is valid and false otherwise.

4.2. Testing

Testing was done by checking the corectness of the output and of the database based on the inputs.

1. **Results**

The resulting application is a pretty good database management application which illustrates how we can work with reflection and database access. Although the number of options is not very large, the application could be easily extended to provide support for larger databases.

1. **Conclusions, Lessons learned, lmprovements**

6.1. Conclusions

In building this program I have understood the statement that „The Devil is in the details”. While the program was not difficult to implement from the algorithmic point of view, its difficulty lies within its large number of classes that have to work together without error. Although I could have chosen a different approach, the resulting application is indeed very elegant and seems easy to extend.

6.2. Lessons learned

During the developement of this application I have learned how to work with reflection techniques and database connection and access, and also how to implement the Singleton design pattern..

6.3. Improvements

* + Enhance the graphic user interface in order to show the database evolution in a graphical manner
  + Allow the user to make changes in the Category and Country entities as well
  + Implement an authentification method, maybe with different priviledges for different types of users.
  + Extend the database with more tables
  + Eliminate the „Refresh tables” buttons and make the refreshing happen automatically after some events.

1. **Bibliography**
   1. Project skeleton:

https://bitbucket.org/utcn\_dsrl/pt-layered-architecture.gitRunnable vs Thread:

* 1. Reflection techniques:

https://bitbucket.org/utcn\_dsrl/pt-reflection-example.git

* 1. Implementing MouseActionListener for JTable:

http://stackoverflow.com/questions/1378096/actionlistener-on-jlabel-or-jtable-cell