Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

2nd Semester 2018-2019

Programming Techniques
Homework 3

Student: Elekes Lukacs-Roland

Group: 30424

Supervising Teacher: Viorica Chifu

Contents

1. Objectives

The main objective is the following:

Consider an application OrderManagement for processing customer orders for a warehouse.

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

- Model classes - represent the data models of the application
- Business Logic classes - contain the application logic
- Presentation classes – classes that contain the graphical user interface
- Data access classes - classes that contain the access to the database

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

a) Analyze the application domain, determine the structure and behavior of its classes and draw an extended UML class diagram.
b) Implement the application classes. Use javadoc for documenting classes.
c) Use reflection techniques to create a method createTable that receives a list of objects and generates the header of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list:
    1. JTable createTable(List<Object> objects);
d) Implement a system of utility programs for reporting such as: under-stock, totals, filters, etc.

As secondary objectives we can enumerate:

- Create connection between the MySQL database and the java application (3.1, 4)
- Create queries (2.2, 3.1, 3.3, 4)
- Design the tables from the database (2, 3.1)
- Design graphical user interfaces that take the commands from the user and acts with the help of a controller

2. Problem Analysis, Modelling, Scenarios, Use Cases

2.1. Problem Analysis

Order management systems are available and necessary in many fields, we can meet this type of systems almost everywhere in the world. One of the most important facility nowadays is to order a product online, which cannot be done without a system like the above mentioned one. Developing an order management system helps the seller, also helps the clients.

Such a system lets the administrator to insert, update, delete products and clients and from the client point of view, a client can search among the products, can filter them and an order can be made.

Using graphical user interfaces the visualization, the commands become easier for every user of the system.

2.2. Modelling

For such an application it is necessary to map the tables from the relational database, we need to model every table as a new entity. To create a connection to the database is also necessary to have a functioning system.

Queries that execute different models have to be modelled in order to be able to perform the basic CRUD (Create, Read, Update, Delete) operations, and additional queries need to be modelled to apply some filters and create new commands (for example searching for phone numbers, names, emails etc.).

A generic entity was modelled which allows to create the queries by reflection needed for common operations for every entity representing the model.

2.3. Scenarios, Use Cases

Scenarios includes maintaining different tables from the database and placing an order. Every entity from the database has its own GUI (except for the table containing the order requests), that allows the user to execute commands on each table.

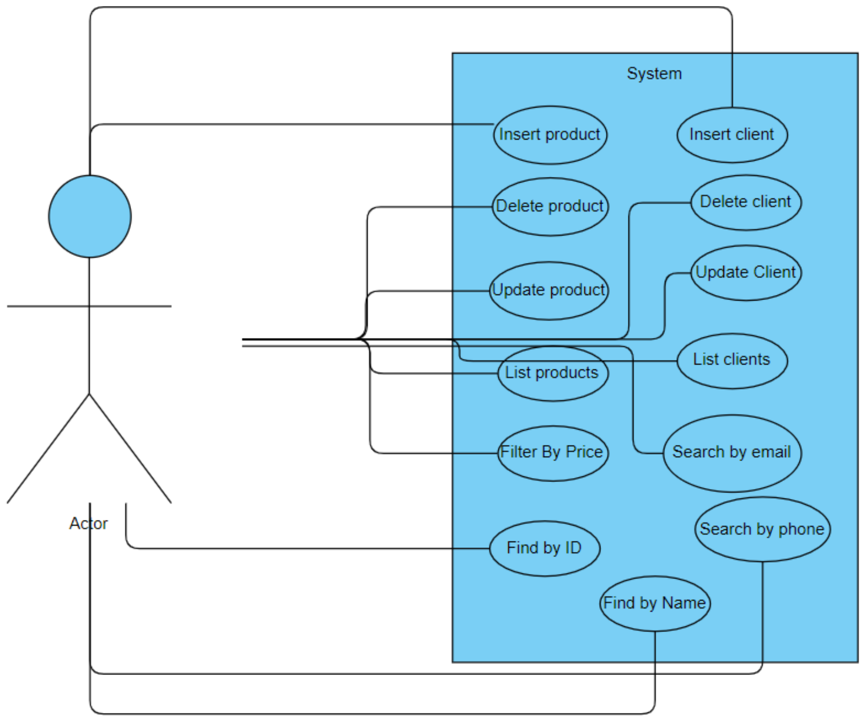
The user can insert and update clients or products by introducing new data, he can select a client or product from the tables and delete them. The user can also apply different filters (like filtering by price) or search for names, emails, phone numbers.

The user can also place an order, in this scenario both tables (clients and products) are displayed and the user have to select one from each entity and enter the amount needed. For every order, a bill is created as a .txt file.

When the application starts, the user can choose what he wants to do: maintain products or clients and place an order.

After every operation, the tables are updated or the desired results are displayed.

The use case diagram is the following:



Even though we can think about an Administrator and a Client as the user of the application, I chose to have only one actor on the system, because he is the one who maintains the whole system, even the orders too.

Use Case title: Insert

Main success scenario:

1. User introduces new data
2. User presses the “Insert” button
3. System takes information
4. New entity is inserted

Alternative sequence:

Data introduced by the user cannot be validated

1. A message is displayed
2. User starts again from the 1. step of the main success scenario

Use Case title: Update

Main success scenario:

1. User introduces new data
2. User presses the “Update” button
3. System takes information
4. New entity is inserted

Data introduced by the user cannot be validated

1. A message is displayed
2. User starts again from the 1. step of the main success scenario

Use Case title: Delete

Main success scenario:

1. User selects an object
2. User presses the “Delete” button
3. Object is deleted

Use Case title: List

Main success scenario:

1. User presses the “List” button
2. Objects are listed in the table

Use case title: Filter by Price

Main success scenario:

1. User introduces the price limits
2. User presses the “Filter” button
3. Products in the price range are listed

Use case title: Search by (id, name, phone, email)

Main success scenario:

1. User introduces the data corresponding to the field selected
2. User presses button
3. Objects are listed

3. Design

3.1. Class Design, Packages and UML Diagram

The project is built on a layered architecture (four important layers are: data Access Layer, business Layer, model and presentation). Furthermore, the design of the classes is based on the Model View Controller pattern. The total number of packages is 7, they are the following: bll, connection, dao, model, presentation, start, validator.

The connection package contains a class ConnectionFactory which is a singleton class, only one object exists and it creates the connection between the application and the relational database (in MySQL).

The model package contains the entities from the database, these map the tables in the database, thus the attributes are the column names, the class name is the table name and they are: Client, Product and OrderRequest.

The dao package contains the classes that contain the access to the database. There is a generic class AbstractDAO<T> where some common (for all models) queries are created and executed (like the CRUD operations, but also find by id or find by name appear there). ClientDAO, ProductDAO, OrderDAO classes extend the AbstractDAO<T> class and implement specific queries for the corresponding data models.

Package bll contains classes ClientBLL, ProductBLL and OrderBLL which contain the application logic. In these classes the objects to be used are validated and other logic can be found (for example in OrderBLL the logic of the order confirmation and bill creation).

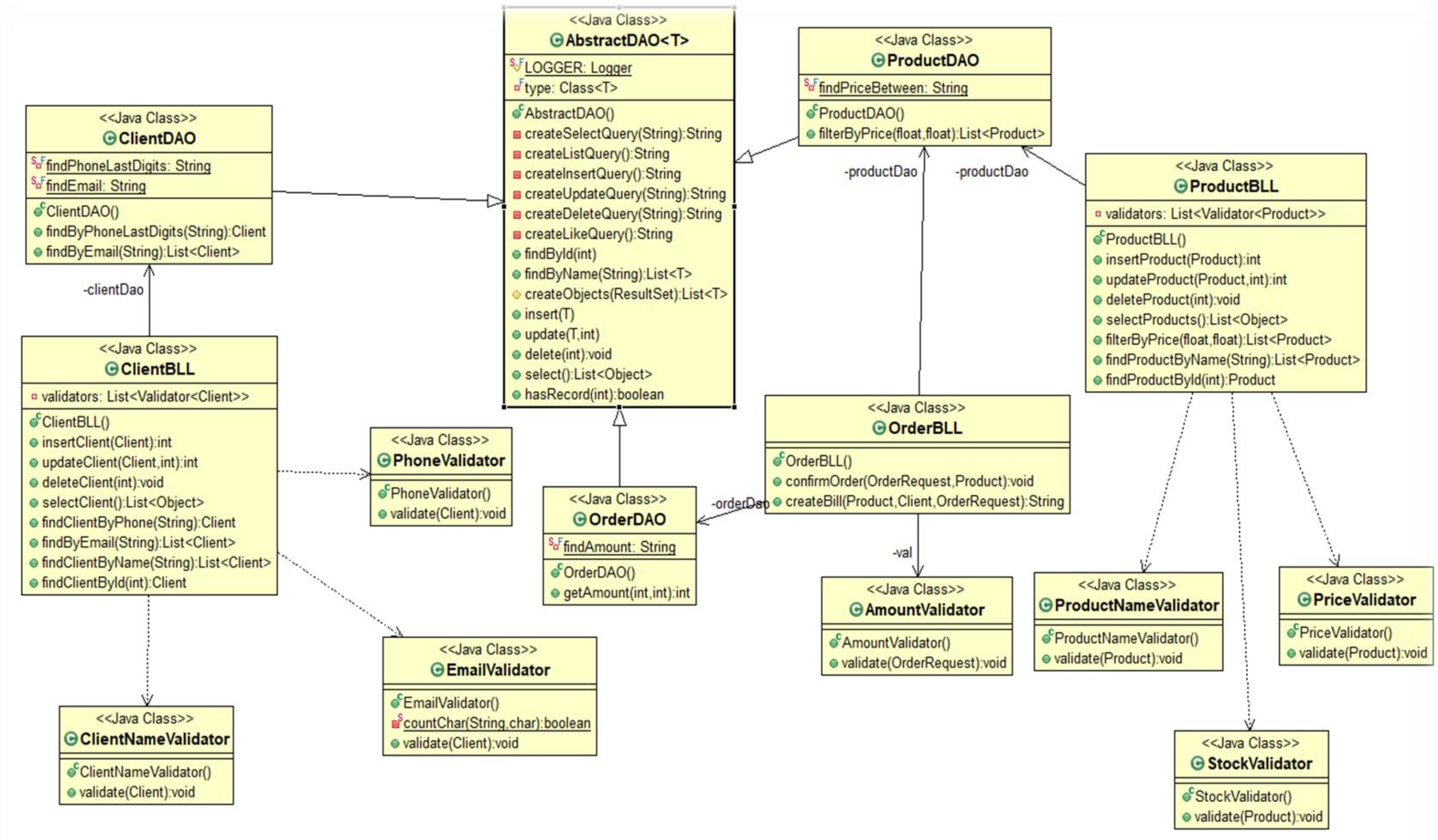
Package presentation contains all the classes that are parts of the graphical user interface and the Controller class which implements the buttons.

Package start contains a class to start the application.

Package validator contains an interface named Validator which has a method validate. The class contains classes that implement the Validator interface and its method, names, numbers and other data is validated there.

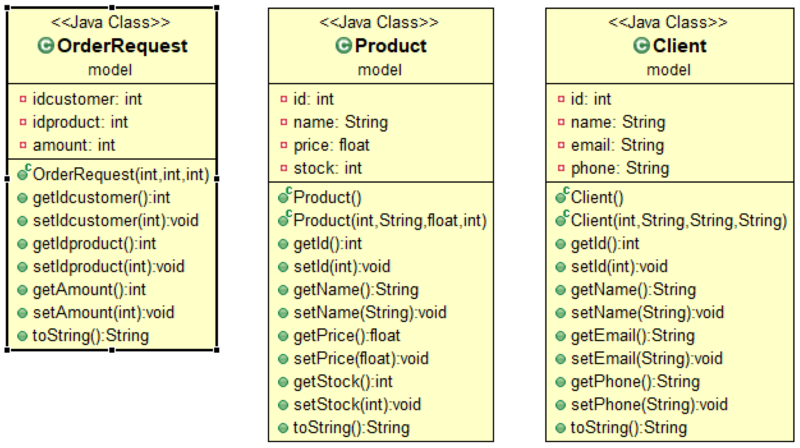
For a better understanding, the UML class diagram was divided in more parts, because the complexity of the problem makes full diagram unreadable.

The diagram that contains all the classes that access the database and the business logic of the application:

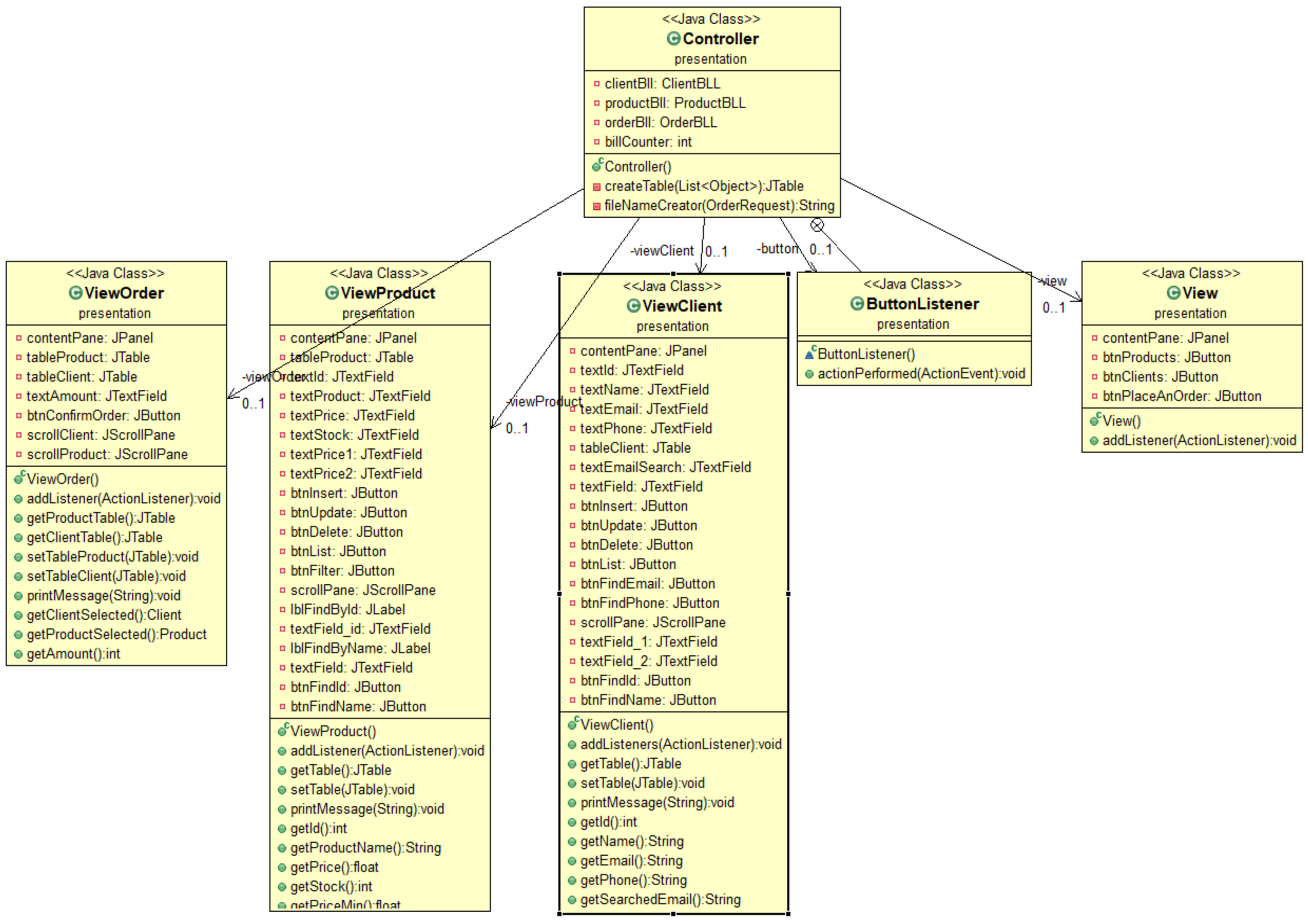


<<Java Class>> AbstractDAO<T>
LOGGER: Logger
type: Class<T>
AbstractDAO()
createSelectQuery(String):String
createListQuery():String
createInsertQuery():String
createUpdateQuery(String):String
createDeleteQuery(String):String
createLikeQuery():String
findById(int)
findByName(String):List<T>
createObjects(ResultSet):List<T>
insert(T)
update(T,int)
delete(int):void
select():List<Object>
hasRecord(int):boolean
<<Java Class>> ProductDAO
findPriceBetween: String
ProductDAO()
filterByPrice(float,float):List<Product>
<<Java Class>> ClientDAO
findPhoneLastDigits: String
findEmail: String
ClientDAO()
findByPhoneLastDigits(String):Client
findByEmail(String):List<Client>
-clientDao
<<Java Class>> ClientBLL
validators: List<Validator<Client>>
ClientBLL()
insertClient(Client):int
updateClient(Client,int):int
deleteClient(int):void
selectClient():List<Object>
findClientByPhone(String):Client
findByEmail(String):List<Client>
findClientByName(String):List<Client>
findClientById(int):Client
-productDao
-productDao
<<Java Class>> ProductBLL
validators: List<Validator<Product>>
ProductBLL()
insertProduct(Product):int
updateProduct(Product,int):int
deleteProduct(int):void
selectProducts():List<Object>
filterByPrice(float,float):List<Product>
findProductByName(String):List<Product>
findProductById(int):Product
<<Java Class>> PhoneValidator
PhoneValidator()
validate(Client):void
<<Java Class>> OrderDAO
findAmount: String
OrderDAO()
getAmount(int,int):int
-orderDao
<<Java Class>> OrderBLL
OrderBLL()
confirmOrder(OrderRequest,Product):void
createBill(Product,Client,OrderRequest):String
-val
<<Java Class>> AmountValidator
AmountValidator()
validate(OrderRequest):void
<<Java Class>> ProductNameValidator
ProductNameValidator()
validate(Product):void
<<Java Class>> PriceValidator
PriceValidator()
validate(Product):void
<<Java Class>> EmailValidator
EmailValidator()
countChar(String,char):boolean
validate(Client):void
<<Java Class>> ClientNameValidator
ClientNameValidator()
validate(Client):void
<<Java Class>> StockValidator
StockValidator()
validate(Product):void

The three model classes are:



The classes from representation are:



3.2. Relationships

The relationships existing among the classes of the project are mainly associations, inheritances and dependencies.

Inheritance can be observed in the dao package, where the classes ClientDAO, ProductDAO and OrderDAO extend the AbstractDAO class, since it has some queries that are common for all data models.

Between BLL and DAO classes the relationship is association, every pair is associated (for example association between ClientBLL and ClientDAO).

Usually every BLL class contains a list of validators, and some validators are used, so we can identify dependencies, except for OrderBLL(only one known validator is used).

The Controller class is associated with every part of the GUI (View, ViewOrder, ViewProduct, ViewOrder classes) and with every class from the logic of the application (the BLL classes).

3.3. Algorithms and Data Structures

Algorithms used in the business logic layer are validating the data model and executing the command. Their complexity is reduced, every validator checks the new object, if no exception happens, the object can be used.

In the OrderBLL in the confirm order method the available stock is checked, if it is enough, the order is processed and the product is updated by subtracting the bought amount from the stock.

A bill is built up in a string taking information from the order, product and client. The name of the text file containing the bill is unique generated by the product id, client id, amount and a counter.

Abstract queries are constructed with reflection techniques, class name, declared fields names are accessed and built into the query this way having generic queries.

Data from result sets are extracted by accessing each value and creating objects through reflection.

In the controller from a list of objects a JTable is created by accessing the field names (they are stored in an array) through reflection and extracting the values (stored in a two dimensional array prepared for the JTable constructor).

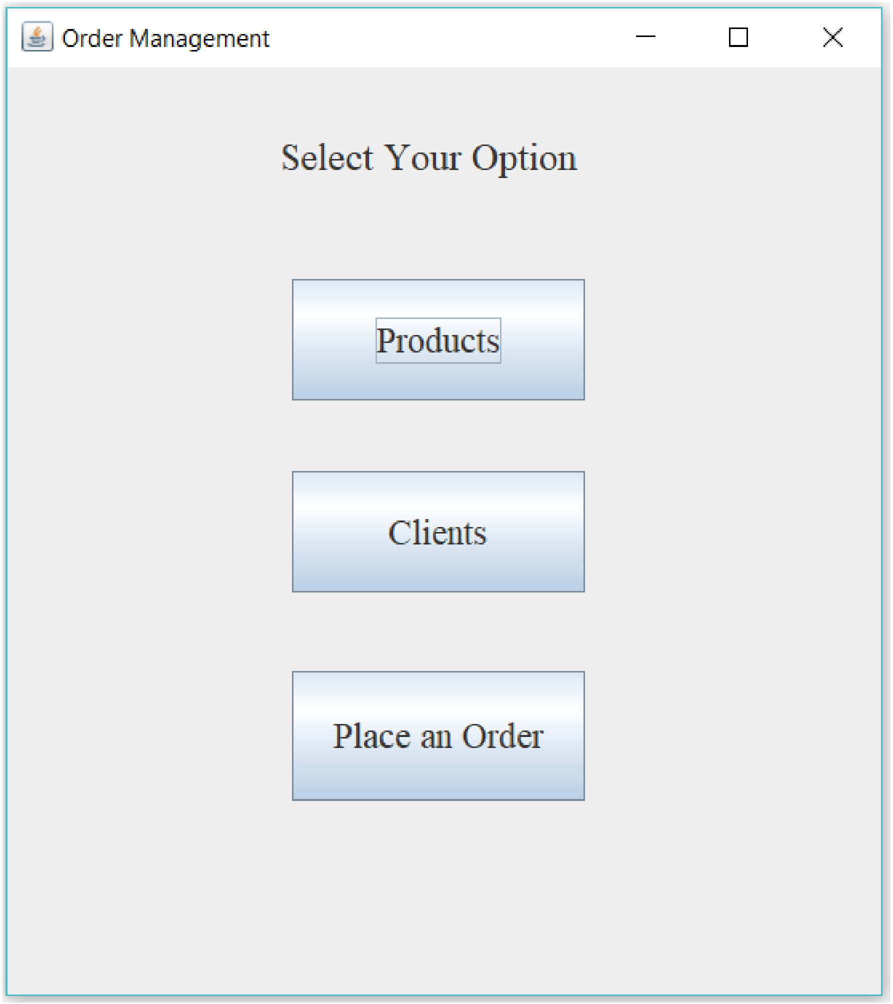
When a button is a pressed, first data is taken and checked, then the operation is processed accessing the method in the BLL classes.

The data structures used are lists, resultsets (for query results) and statements. Exceptions are used to identify every time an error occurs.

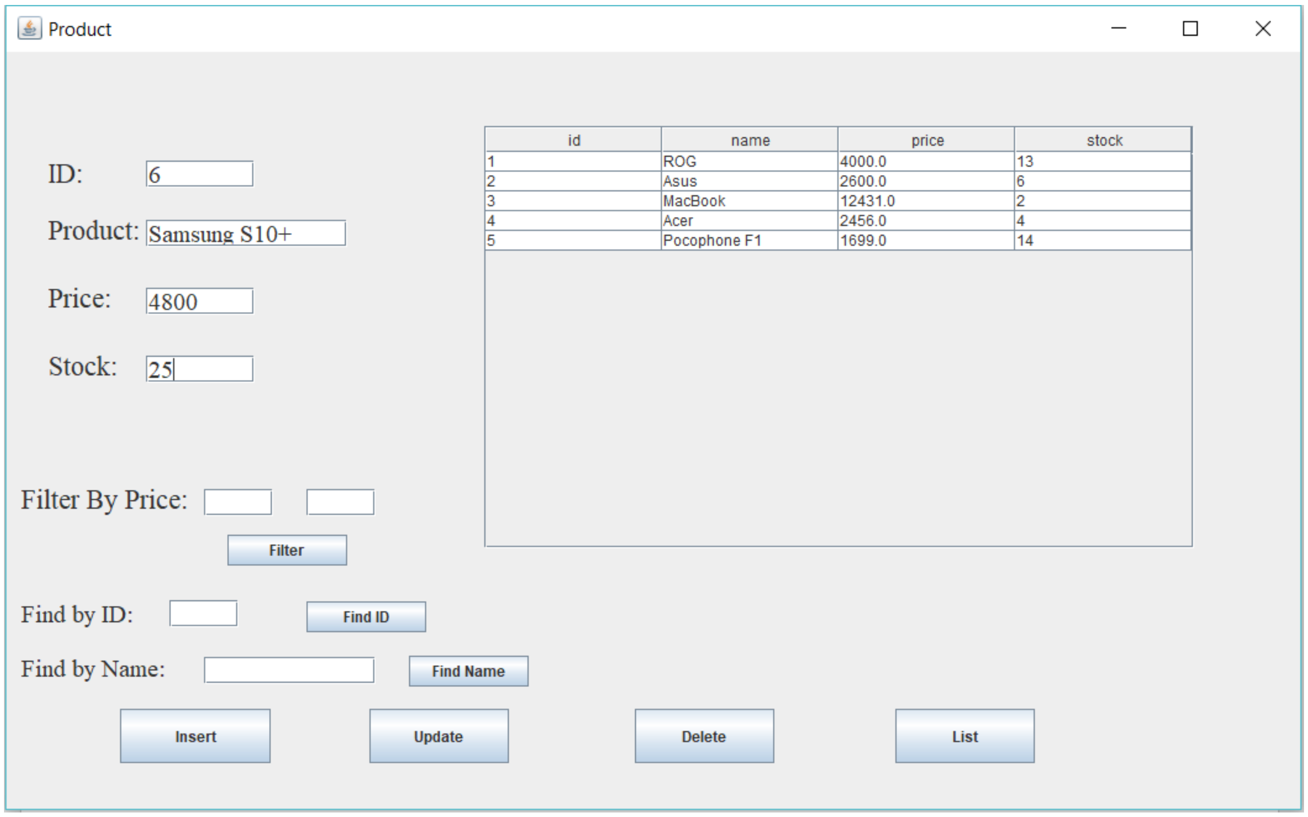
3.4. Graphical User Interface

The graphical user interface provides an easy handling of the system for the user. It has some labels (for information), text fields (for introducing data and output of the result) and buttons. The buttons are representative since they have a label describing shortly the operation it performs.

It is constructed using the swing library and the windowbuilder of Eclipse to have a nicer GUI and to save time. There are four frames (JFrame) in total. When the application starts, the first frame appears, where the user is told that he can choose an option : Products, Clients or Place an Order.



When the user chooses Products, the Products frame appears, where he can maintain the Product table. There are labels describing each textfield and buttons that are representative for their operations.



ID: 6

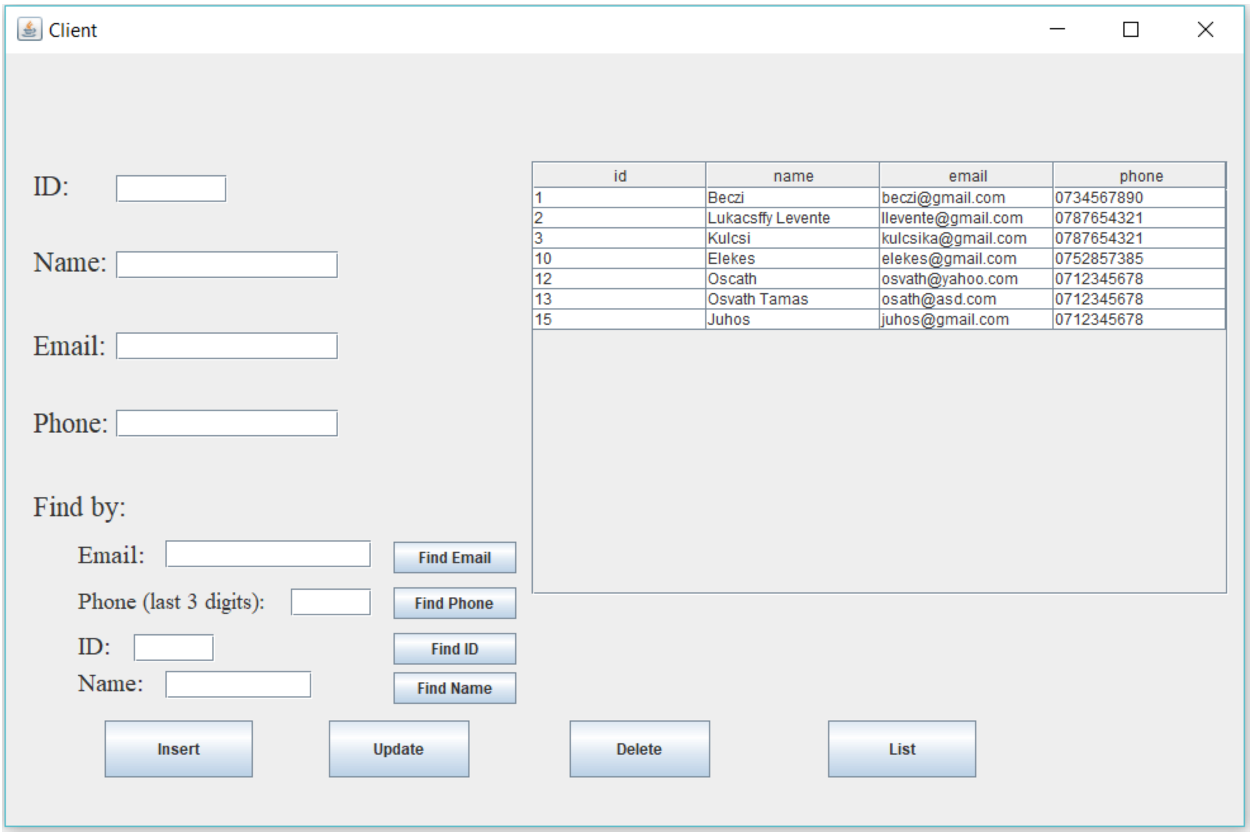
Product: Samsung S10+

Price: 4800

Stock: 25

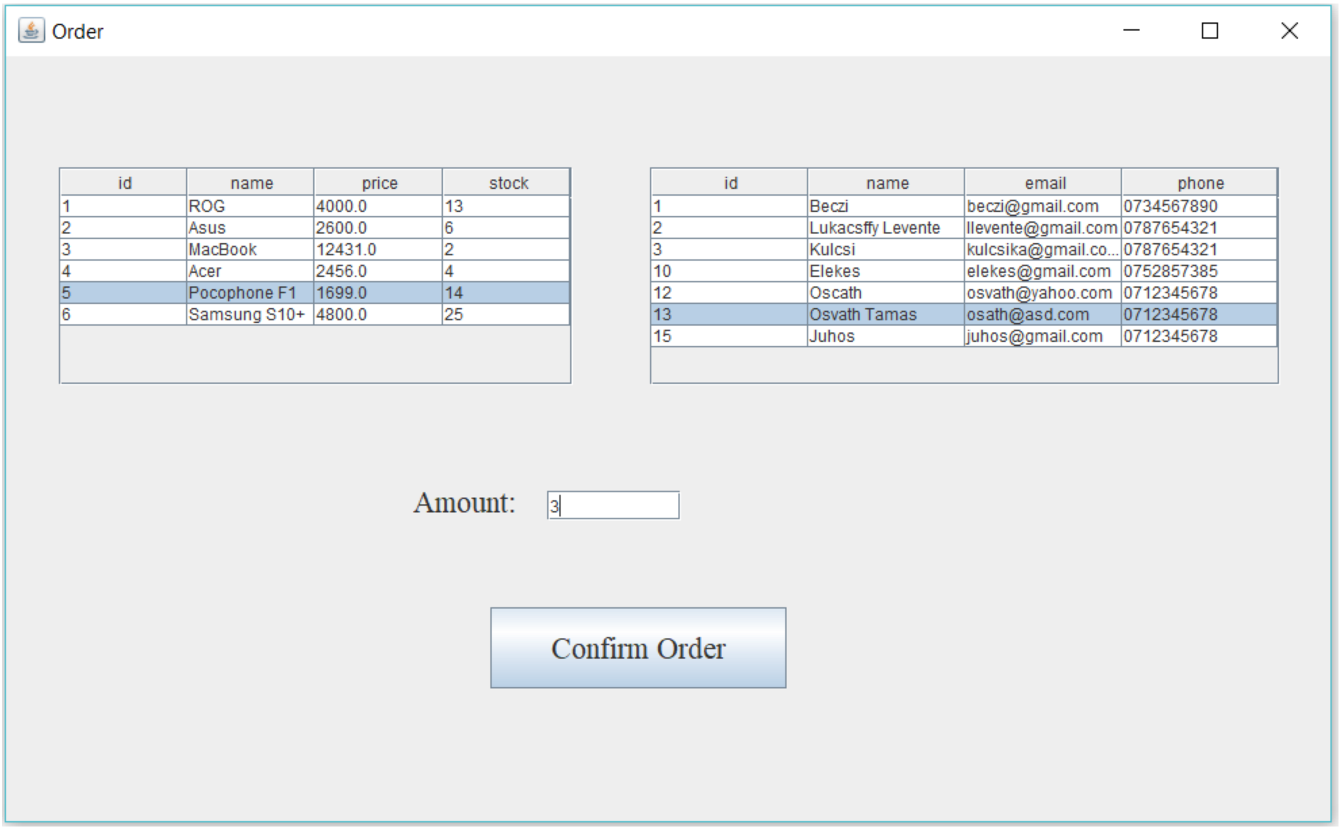
id	name	price	stock
1	ROG	4000.0	13
2	Asus	2600.0	6
3	MacBook	12431.0	2
4	Acer	2456.0	4
5	Pocophone F1	1699.0	14

When the user chooses Clients, the Client window appears, where he can maintain the Client table.



id	name	email	phone
1	Beczi	beczi@gmail.com	0734567890
2	Lukacsffy Levente	llevente@gmail.com	0787654321
3	Kulcsi	kulcsika@gmail.com	0787654321
10	Elekes	elekes@gmail.com	0752857385
12	Oscath	osvath@yahoo.com	0712345678
13	Osvath Tamas	osath@asd.com	0712345678
15	Juhos	juhos@gmail.com	0712345678

When the user presses Place on Order, a frame appears with two tables, clients and products where he can select one from each table and give an amount. After pressing the confirm order, the order is sent.



id	name	price	stock
1	ROG	4000.0	13
2	Asus	2600.0	6
3	MacBook	12431.0	2
4	Acer	2456.0	4
5	Pocophone F1	1699.0	14
6	Samsung S10+	4800.0	25

id	name	email	phone
1	Beczi	beczi@gmail.com	0734567890
2	Lukacsffy Levente	llevente@gmail.com	0787654321
3	Kulcsi	kulcsika@gmail.co...	0787654321
10	Elekes	elekes@gmail.com	0752857385
12	Oscath	osvath@yahoo.com	0712345678
13	Osvath Tamas	osath@asd.com	0712345678
15	Juhos	juhos@gmail.com	0712345678

After almost each operations (excepts are filtering and searching operations) a message dialog box appears and tells if the operation was successful or tells the error that appeared.

4. Implementation

In this chapter some examples from each package and some methods will be presented.

Data model classes are entities with attributes that corresponds to the database's table columns like

```
/**Id of the client.*/
private int id;
/**Name of the client.*/
```

```
	private String name;
	/**Email address of the client.*/
	private String email;
	/**Phone number of the client.*/
	private String phone;
```

For every model class some getters and setters are written to have access to their attributes.

The ConnectionFactory class is a singleton class, it's constructor is private and all attributes are static, and they are part of the connection (like url, user and password of the database).

Methods are implemented for closing the connection, statement and result set.

```
private ConnectionFactory() {
		try {
			Class.forName(DRIVER);
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
	}

	private Connection createConnection() {
		Connection connection = null;
		try {
			connection = DriverManager.getConnection(DBURL, USER, PASS);
		} catch (SQLException e) {
			LOGGER.log(Level.WARNING, "An error occured while trying to 
connect to the database");
			e.printStackTrace();
		}
		return connection;
	}


public static void close(Connection connection) {
		if (connection != null) {
			try {
				connection.close();
			} catch (SQLException e) {
				LOGGER.log(Level.WARNING, "An error occured while trying 
to close the connection");
			}
		}
	}
```

In the AbstractDAO class queries are build up with StringBuilders with reflection techniques, two exemples are Select and Insert queries:

```
private String createSelectQuery(String field) {
		StringBuilder sb = new StringBuilder();
		sb.append("SELECT ");
		sb.append(" * ");
```

```
            sb.append(" FROM ");
            sb.append(type.getSimpleName());
            sb.append(" WHERE " + field + " =?");
            return sb.toString();
      }
```

And insert:

```
private String createInsertQuery() {
            StringBuilder sb = new StringBuilder();
            sb.append("INSERT INTO ");
            sb.append(type.getSimpleName());
            sb.append(" (");
            for(Field field : type.getDeclaredFields()) {
                  sb.append(field.getName() + ",");
            }
            sb.deleteCharAt(sb.length()-1);
            sb.append(") VALUES (");

            for(int i = 0; i < type.getDeclaredFields().length; i++) {
                  sb.append("?,");
            }
            sb.deleteCharAt(sb.length()-1);
            sb.append(")");
            return sb.toString();
      }
```

These queries are used when a statement is prepared to be executed. When we execute a query, the connection is instantiated than the statement is prepared using a string containing the query (returned by one of the previous methods). Then the parameters are set, and the statement is executed, than the result is stored in a result set.

```
public T insert(T t) {
            Connection connection = null;
            PreparedStatement statement = null;
            //ResultSet resultSet = null;
            String query = createInsertQuery();
            try {
                  connection = ConnectionFactory.getConnection();
                  statement = connection.prepareStatement(query);
                  int pos = 1;
                  for(Field field : type.getDeclaredFields()) {
                        field.setAccessible(true);
                        Object value = field.get(t);
                        statement.setString(pos, value.toString());
                        pos++;
                  }
                  statement.executeUpdate();
            } catch (IllegalAccessException e) {
                  e.printStackTrace();
            } catch (SecurityException e) {
                  e.printStackTrace();
            } catch (IllegalArgumentException e) {
                  e.printStackTrace();
            } catch (SQLException e) {
```

```
				e.printStackTrace();
			} finally {
				ConnectionFactory.close(statement);
				ConnectionFactory.close(connection);
			}

			return t;
		}
```

We take as example the ProductDAO class to show why such a class is required. In that class, product specific queries are implemented like filter by price. The strings containing the queries are declared as private final static variables, than the query is executed using the same things as before, connection, statement etc.

```
public List<Product> filterByPrice (float limit1, float limit2){
		Connection connection = null;
		PreparedStatement statement = null;
		ResultSet result = null;

		//List<Product> products = new ArrayList<Product>();
		try {
			connection = ConnectionFactory.getConnection();
			statement = connection.prepareStatement(findPriceBetween);
			statement.setFloat(1, limit1);
			statement.setFloat(2, limit2);
			result = statement.executeQuery();
			return createObjects(result);

		} catch (SecurityException e) {
			e.printStackTrace();
		} catch (IllegalArgumentException e) {
			e.printStackTrace();
		} catch (SQLException e) {
			e.printStackTrace();
		} finally {
			ConnectionFactory.close(result);
			ConnectionFactory.close(statement);
			ConnectionFactory.close(connection);
		}
		return null;
```

In the BLL classes are operations are resolved and other logic is added if needed. An example for inserting a client is :

```
public int insertClient(Client c) throws SQLIntegrityConstraintViolationException {
		for(Validator<Client> val : validators) {
			val.validate(c);
		}

		if(clientDao.hasRecord(c.getId())) {
			throw new SQLIntegrityConstraintViolationException("Client with
id " + c.getId() + " already exists in the table!");
		}
		return clientDao.insert(c).getId();}
```

where the SQLIntegrityConstraintViolationException exception is thrown when a duplicate primary key appears and validators are created in the constructor of the class:

```
public ClientBLL() {
		validators = new ArrayList<Validator<Client>>();
		validators.add(new ClientNameValidator());
		validators.add(new EmailValidator());
		validators.add(new PhoneValidator());

		clientDao = new ClientDAO();
	}

Each validator implements the Validator interface:
public interface Validator<T> {

	public void validate (T t);

}
```

An example of a validator is:

```
public class ClientNameValidator implements Validator<Client> {
	/**Validates the name of a client
	 * Must contain only letters, '-' symbols or spaces
	 */
	public void validate(Client t) {
		boolean isSuccess = true;
		char [] name = t.getName().toCharArray();
		for(char c : name) {
			if(!Character.isLetter(c) && !(c==' ') && !(c=='-')) {
				isSuccess = false;
			}
		}
		if(!isSuccess) {
			throw new IllegalArgumentException("The name contains other
characters than letters!");
		}

	}

}
```

Other validators are simpler than this example, numbers are only checked to be non-negative.

Classes representing the GUI contain some atomic swing elements, and some methods are implemented to return data from text fields. ViewClient, ViewOrder and ViewPublic classes extend JFrame, and they are set to dispose when they are closed, only the View class is set to exit. This way we can open and close the other windows anytime we want.

A method to mention would be the method that extracts a selected client (or product) in the ViewOrder class:

```java
public Client getClientSelected() {
		int row = tableClient.getSelectedRow();
		String idValue = tableClient.getValueAt(row, 0).toString();
		int id = Integer.parseInt(idValue);
		String name = tableClient.getValueAt(row, 1).toString();
		String email = tableClient.getValueAt(row, 2).toString();
		String phone = tableClient.getValueAt(row, 3).toString();
		Client client = new Client(id, name, email, phone);
		return client;
	}
```

In the Controller class the most important is the implementation of the listeners for the buttons and the method that creates a JTable from a list of Objects. In this method, we store the column names in an array of Strings, and the content of the table is saved in a two-dimensional array. Column names are found with the getDeclaredFields() method using reflection techniques and the data is also accessed in this way. Since the attributes are private, we have to set them accessible first and the values can be obtained.

The ActionListener is implemented by using a switch case with a case for each button. An example would be:

```java
case "deleteclient" :
				row = viewClient.getTable().getSelectedRow();
				id = Integer.parseInt(viewClient.getTable().getValueAt(row, 
0).toString());
				try {
					clientBll.deleteClient(id);
					viewClient.printMessage("Client deleted successfully!");
	
	viewClient.setTable(createTable(clientBll.selectClient()));
	
	viewOrder.setTableClient(createTable(clientBll.selectClient()));
	
	viewOrder.setTableProduct(createTable(productBll.selectProducts()));
				} catch (IllegalArgumentException e) {
					viewClient.printMessage(e.getMessage());
				}
				break;
```

Every time a change is made, the tables are updated.

The class Start contains the main method where a controller is created, in order to start the application.

5. Usage and Testing

The user starts the application, he can choose from the three options. For example if the Product window was opened, he can insert, update, delete and list objects by introducing data and pressing the corresponding buttons. In addition, he can apply some filters for price or name, and can search for a Product by ID. For processing Clients, user have to open the Client window where he has similar options to do.

To make an order, Place an Order button has to be pressed, in that window the user should select the client, the product and introduce the desired amount and after that confirm the order.

The application was tested by introducing different data, performing operations and observing the response of the system, in this way some bugs were detected and corrected, to have a system with minimal errors.

6. Conclusions

Future developments:

- More complex database
- More filters, more complex queries
- Check the uniqueness of some fields like phone number
- Ability to buy more products at the same time, not only one
- Creating an interface for logging in or registration

During the development of this assignment, some research was needed for understanding the concepts reflection techniques, database connection and some parts of the layered structure. In conclusion I have learnt how to create a connection between a java application and a relational database, also I understood the power and the usage of the reflection techniques. Moreover, I learned to use a window builder for achieving a better functioning and looking graphical user interface. I understood how important are java exceptions for a correct functioning of an application.

7. Bibliography

- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW3_Tema3/Tema3_HW3_Indications.pdf
- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW3_Tema3/HW3_Mysql_Installation_Guidelines.pdf
- https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-layered-architecture.git
- https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-reflection-example.git
- https://stackoverflow.com/questions/1966836/resultset-to-list
- https://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/
- https://dzone.com/articles/layers-standard-enterprise
- http://tutorials.jenkov.com/java-reflection/index.html