MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

FUNDAMENTAL PROGRAMMING TECHNIQUES

ASSIGNMENT 2
SOLUTION DESCRIPTION DOCUMENT

ORDER MANAGEMENT

Student: Mihai Cristina-Mădălina
Teaching assistant: Chifu Viorica

Faculty of Automation and Computer Science
Computer Science Department (En)
2nd year of study, gr. 30424

Content:

1. Objective

- **The main objective** of this laboratory work is to **design and implement an order management application** for **processing customer orders** for a warehouse. Relational databases are used to store the products, the clients and the orders.
- The project will implemented as a **Maven Project** using **Intellij IDEA application**.

Secondary objective:

- to structure the application in packages: model, business logic, presentation, data access;
- to process commands from a text file given as argument;
- to create a jar file for executing the application;
- to implement a PDF file generator to generate reports;
- to create JavaDoc files;
- to create SQL dump file containing the SQL statements for creating the database, the tables and populating the tables.

2. Analysis of the laboratory work, scenarios, use-cases

Analysis:

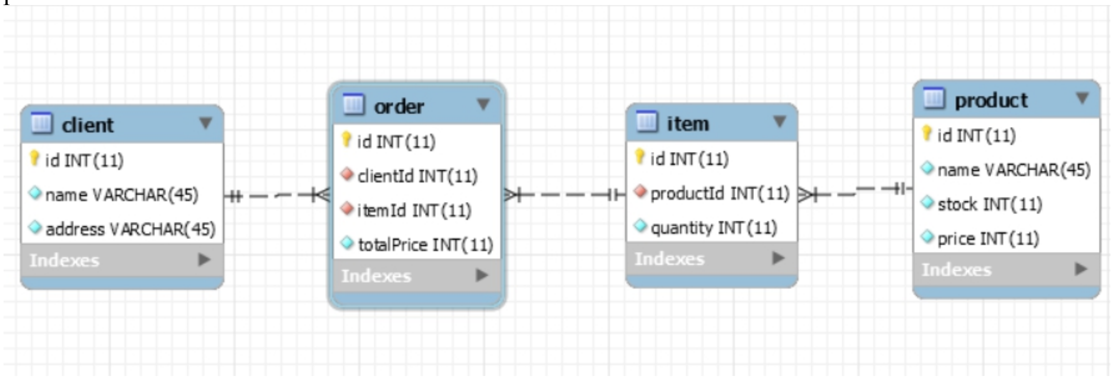
The application should be able to read commands from the text file input (i.e. ***commands.txt***) and execute them according to the following table. The result of the executed commands will be visible in the database tables.

Command name	Command Syntax	Description
Add client to the database	Insert client: Ion Popescu, Bucuresti	Insert in the database a new client with name Ion Popescu and address Bucuresti
Delete Client from the database	Delete client: Ion Popescu	Delete from database the client with name Ion Popescu
Add product to the database	Insert product: apple, 20, 1	Add product apple with quantity 20 and price 1
Delete product from the database	Delete product: apple	Delete product apple from database
Create order for client	Order: Ion Popescu, apple, 5	Create order for Ion Popescu, with apple quantity 5. Also update the apple stock to 15. Generate a bill in pdf format with the order and total price of 5
Generate reports	Report client Report order Report product	Generate pdf reports with all clients/orders/products displayed in a tabular form. The reports should contain the information corresponding to the entity for which reports are asked (client, order or product) returned from the database by a SELECT * query, displayed in a table in a PDF file.

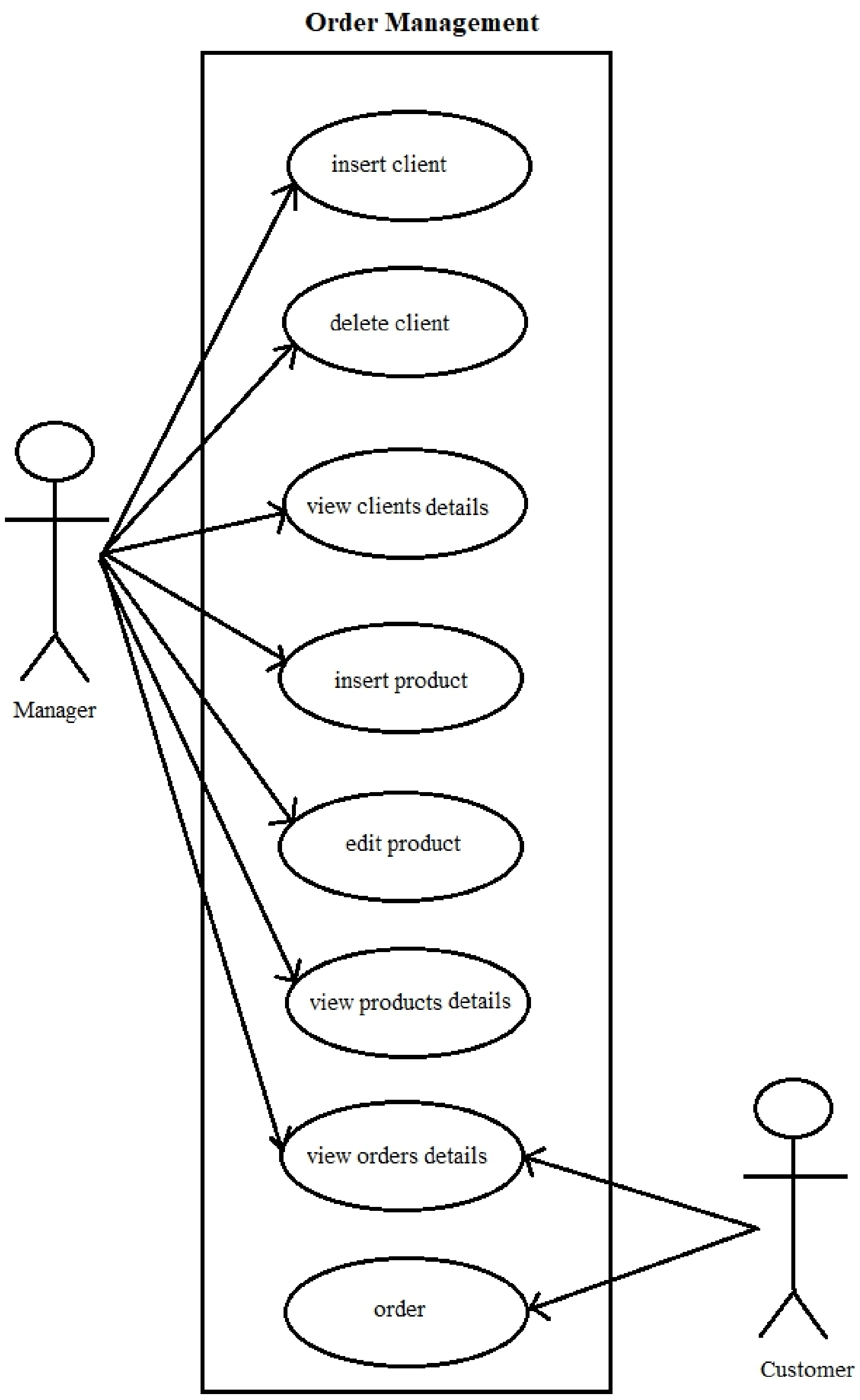
For this application, there has been chosen 4 table to work with and they store the following data:

- ***Client***: id, full name and address
- ***Product***: id, name, current stock and price
- ***Item***: id, id of the product and quantity
- ***Order***: id, client's id, id of the item and total price.

The relations between the tables is given below and other details will be discussed later in the presentation:



Use-cases:


Order Management
insert client
delete client
view clients details
insert product
edit product
view products details
view orders details
order
Manager
Customer

- Title: insert client
  Resume: The user can insert a new client in the database by writing in the *commands.txt* file the command on a new line "Insert client:" and then the full name of the client and their address. If the spelling of the command is correct, the customer will be successfully introduced in the database. Otherwise, an exception will be thrown.
  Actor: Manager

- Title: delete client
  Resume: The user can remove a client from the database by inserting in the *commands.txt* file the command "Delete client:" followed by the full name of the client. If the syntax is correct, the client will be successfully deleted from the database, along with their previous orders. Otherwise, an exception will be thrown.
  Actor: Manager

- Title: view clients details
  Resume: The user can view all the current clients stored in the database by inserting in the *commands.txt* file the command "Report client". If the syntax is correctly written, then a PDF report in tabular form will be generated. The table will contain the following fields: the id, the full name and address of each client. The name of the report will be "ClientReportX.pdf", where X is the number of generated report. If the syntax is incorrect, the report will not be generated.
  Actor: Manager

- Title: insert product
  Resume: The user can insert a new product in the database by writing in the *commands.txt* file the command "Insert product:" followed by the name of the new product, the available quantity and the price for a single product of that type. If the syntax is correctly written, then the product will be successfully inserted into the database. Otherwise, an exception will be thrown.
  Actor: Manager

- Title: edit product
  Resume: The user can edit a product either by inserting a new available quantity to an existing product or by deleting a product. In case of *updating* the quantity of a product, the user have to write in the *commands.txt* file the command "Insert product:" followed by the name of the product in the database, the new available quantity and the same price. The final quantity will be computed as the sum of the previous and freshly inserted values. If the syntax is correct, the quantity will be updated. Otherwise, an exception will be thrown. In case of *deleting* a product from the database, , the user has to write in the *commands.txt* file the command "Delete product:" followed by the name of the product. If the syntax is correct and the product exists in the database, then the product will be successfully deleted. Otherwise, an exception will be thrown.
  Actor: Manager

- Title: view products details
  Resume: The user can view all the current products stored in the database by inserting in the *commands.txt* file the command "Report product". If the syntax is correctly written, then a PDF report in tabular form will be generated. The table will contain the following fields: the id, the name, the quantity and the price of each product. The name of the report will be "ProductReportX.pdf", where X is the number of the generated report. If the syntax is incorrect, the report will not be generated.
  Actor: Manager

- Title: view orders details
  Resume: The user can view all the processed orders or place an order at this step. In case of *viewing all the orders* stored in the database, the user has to insert in the *commands.txt* file the command "Report order". If the syntax is correctly written, then a PDF report in tabular form will be generated. The table will contain the following fields: the id of the order, the name of the client who placed the order, the product order, its quantity and the total price of each product. The name of the report will be "OrderReportX.pdf", where X is the number of the generated report. If the syntax of the command is incorrect, the report will not be generated. In case of *viewing the current status of order* of

a client, the user has to insert the command “Order:” followed by the name of the client, the required product and its quantity. If the syntax is correct, then a PDF report in tabular form will be generated, containing the following fields: the name of the client, their product and the quantity and price of the each ordered product. The name of the report will be “OrderReportX.pdf”, where X is the full name of the client. If the order cannot be processed, then an PDF with a corresponding message will be created.
Actor: Manager/Customer

- Title: order
  Resume: The user can place an order by introducing in the *commands.txt* file the command “Order:” followed by the name of the client, the required product and the quantity. If the syntax is correct and there are enough available products in the stock, then a PDF report in tabular form will be generated, containing the following fields: the name of the client, their product and the quantity and price of the each ordered product. The name of the report will be “OrderReportX.pdf”, where X is the full name of the client. Otherwise, a PDF report with an under-stock message will be created.
  Actor: Customer

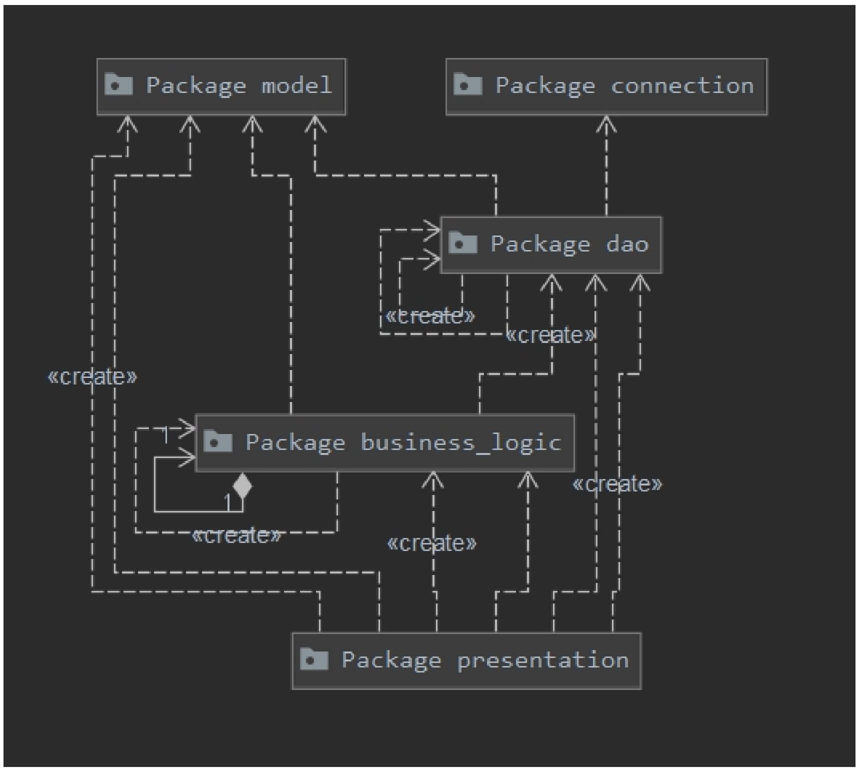
Scenarios:

- Preconditions: The user has to insert the desired commands in the *commands.txt* file. Then the user has to introduce the following command line in the terminal: java -jar PT2020_30424_Cristina_Mihai_Assignment_3.jar commands.txt
- Normal Scenario: The user has perfectly introduced and has perfectly run the command. After this he/she can open (1) the MySQL Workbench to check the content of the tables and (2) the PDF files generated along the execution of the application for a better understanding of the status of each command.
- Alternative Scenario: If the user does not introduce the command line as predefined, an error will occur.

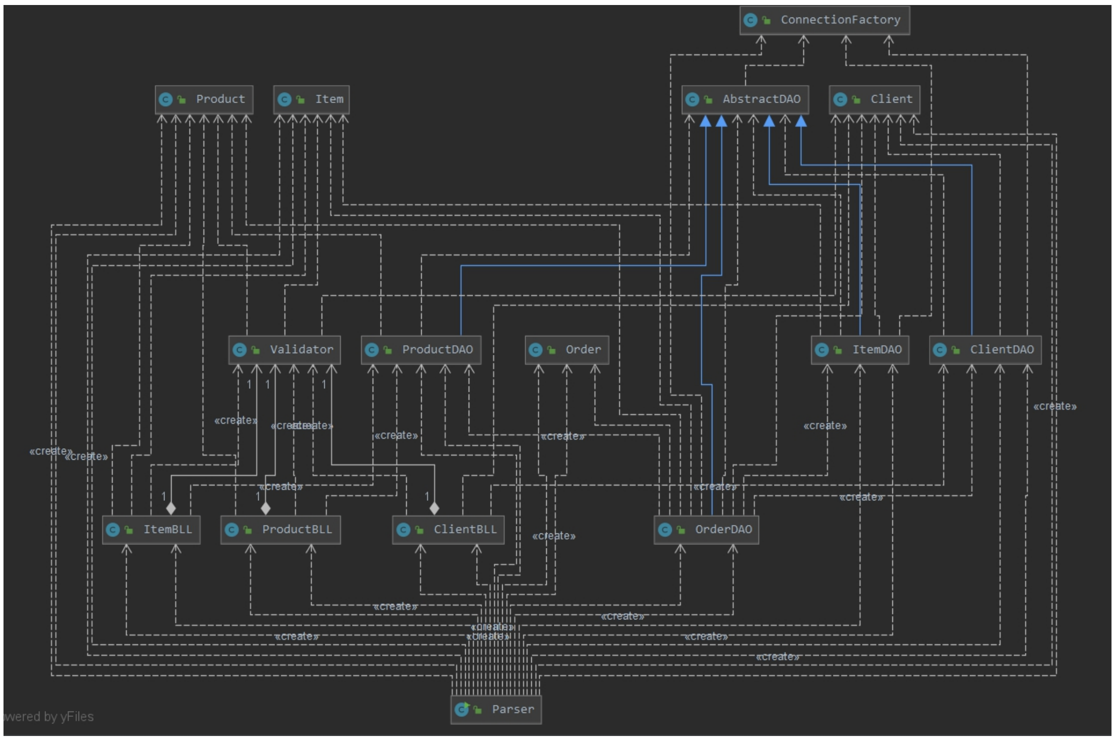
3. Design

3.1 Design approach

The order management simulator is composed of multiple packages with the following meaning:
1. The *model* package contains the classes which models the tables in the database.
2. The *connection* package contains the creation of the channel through which the application and the database will interact.
3. The *dao* package performs the operation on the database tables.
4. The *business_logic* package contains the steps through which the data input gets before any modification in the database is made.
5. The *presentation* package has the responsibility to read the data from the file input, split in into understandable commands and send them to be checked and then to be executed.



3.1 Class diagram



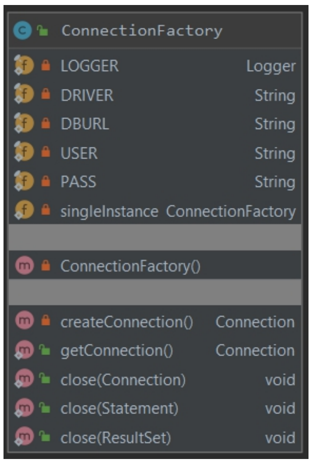
3.1 Data structures

- List<T>: An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list[1]. I chose this data structure to store the elements for a given table T. I used methods for adding elements in the list, checking if it has any elements and retrieving them.

4. Implementation

Package connection:
ConnectionFactory class:

Instance variables:
- LOGGER: Logger
- DRIVER: String
- DBURL: String
- USER: String
- PASS: String

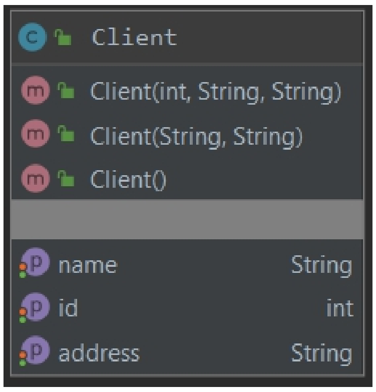


ConnectionFactory	
LOGGER	Logger
DRIVER	String
DBURL	String
USER	String
PASS	String
singleInstance	ConnectionFactory
ConnectionFactory()	
createConnection()	Connection
getConnection()	Connection
close(Connection)	void
close(Statement)	void
close(ResultSet)	void

All variables are private and help the application log to the database. There has been implemented methods for connecting to the database, obtaining the connection whenever needed and closing the connection, statements and result sets created. All rights for this class are reserved to the UTC-N DSRL[2]. The only changes that have been made are to the values of DBURL, USER and PASS variables because they had to be specific to the current database we are working on.

Package model:

Client class:



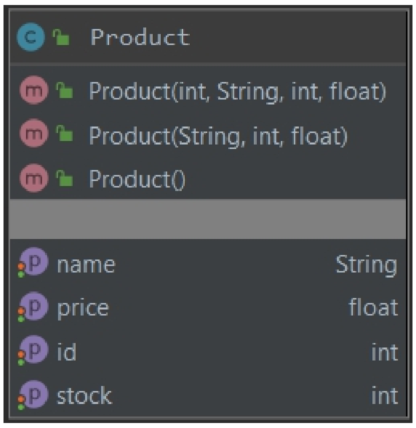
Client
Client(int, String, String)
Client(String, String)
Client()

name	String
id	int
address	String

Instance variables:
- *id*: int
- *name*: String
- *address*: String

There has been used only private variables for this class and there has been implemented just constructors, getters and setters. The name and type of variables has been chosen intentionally identical to the fields of the client table in the database. Because of that, the *id* represents the id of each client in the database, the *name* represents their full name and the *address* represents their city.

Client class:



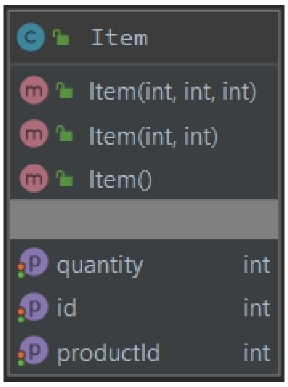
Product
Product(int, String, int, float)
Product(String, int, float)
Product()

name	String
price	float
id	int
stock	int

Instance variables:
- *id*: int
- *name*: String
- *stock*: int
- *price*: float

All the used variables are private and their names and and types are identical to the fields of the product table in the database. The *id* represents the identification of the product, the *name* represents the name of the product, the *stock* represents the current available product of that type and the *price* represents the price of one product with that name. For this class, there has been implemented constructors, getters and setters.

Item class:



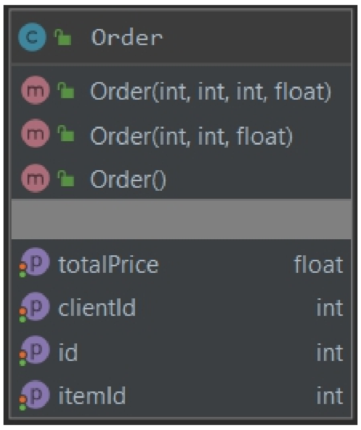
Item
Item(int, int, int)
Item(int, int)
Item()

quantity	int
id	int
productId	int

Instance variables:
- *id*: int
- *productId*: int
- *quantity*: int

All the variables above are private and their names and types are identical to the fields of the product table in the database. The *id* represents the identification of the item, the *productId* represents the product which will be identified and the *quantity* represents the number of products requested by the client. For this class, there has been implemented constructors, getters and setters.

Order class:



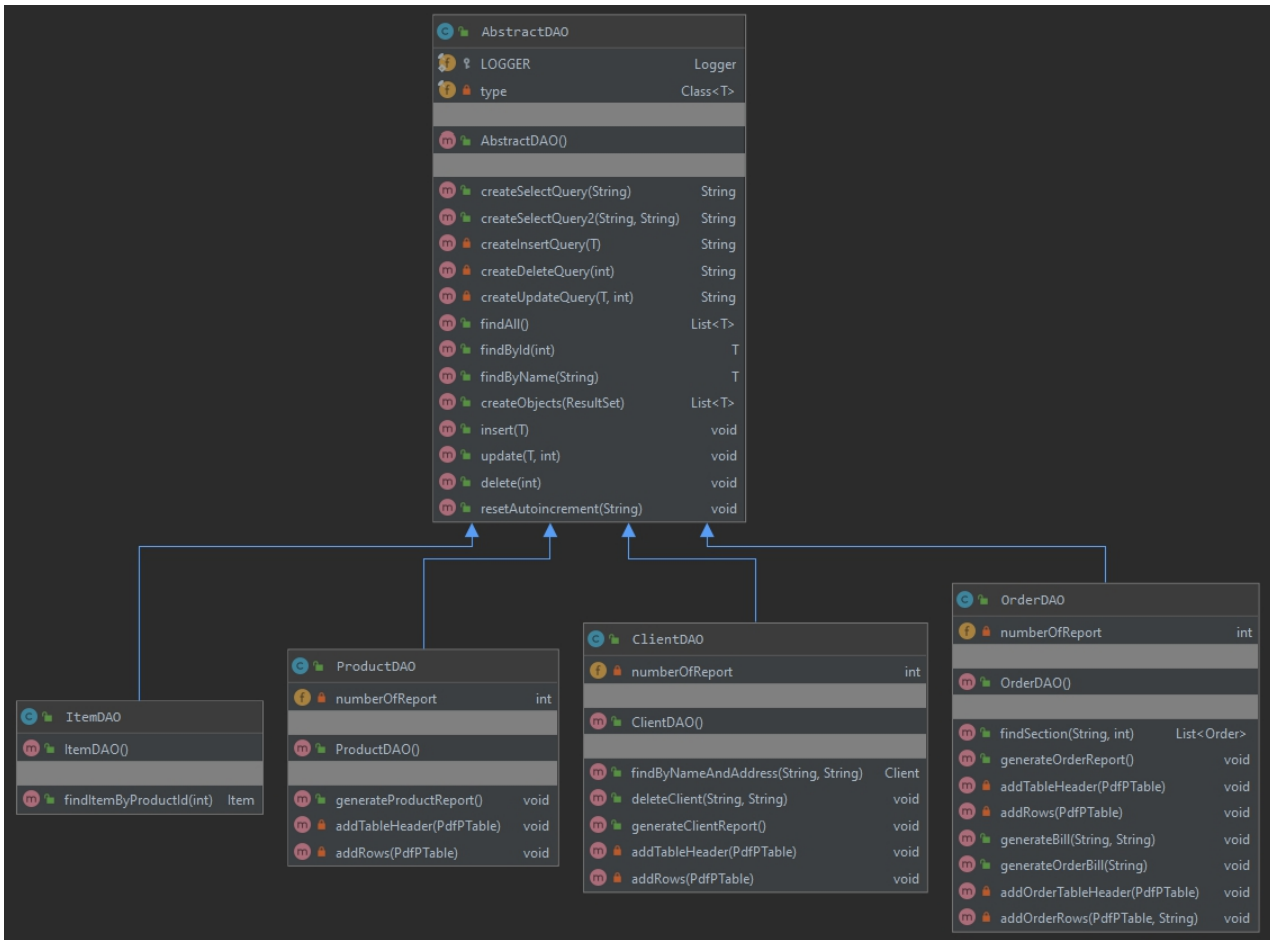
Order
Order(int, int, int, float)
Order(int, int, float)
Order()

totalPrice	float
clientId	int
id	int
itemId	int

Instance variables:
- *id:* int
- *clientId:* int
- *itemId:* int
- *price*: float

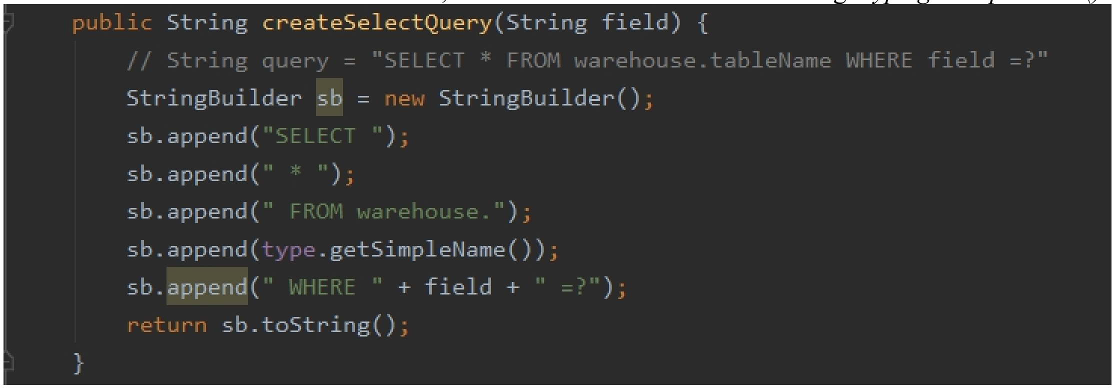
All the variables above are private and their names and types are identical to the fields of the product table in the database. The *id* represents the identification of the item, the *clientId* and *itemId* represents the client and item which will be identified and the *price* represents the monetary value of the product requested by the client. For this class, there has been implemented constructors, getters and setters.

Package model:


AbstractDAO
LOGGER Logger
type Class<T>
AbstractDAO()
createSelectQuery(String) String
createSelectQuery2(String, String) String
createInsertQuery(T) String
createDeleteQuery(int) String
createUpdateQuery(T, int) String
findAll() List<T>
findById(int) T
findByName(String) T
createObjects(ResultSet) List<T>
insert(T) void
update(T, int) void
delete(int) void
resetAutoincrement(String) void
ItemDAO
ItemDAO()
findItemByProductId(int) Item
ProductDAO
numberOfReport int
ProductDAO()
generateProductReport() void
addTableHeader(PdfPTable) void
addRows(PdfPTable) void
ClientDAO
numberOfReport int
ClientDAO()
findByNameAndAddress(String, String) Client
deleteClient(String, String) void
generateClientReport() void
addTableHeader(PdfPTable) void
addRows(PdfPTable) void
OrderDAO
numberOfReport int
OrderDAO()
findSection(String, int) List<Order>
generateOrderReport() void
addTableHeader(PdfPTable) void
addRows(PdfPTable) void
generateBill(String, String) void
generateOrderBill(String) void
addOrderTableHeader(PdfPTable) void
addOrderRows(PdfPTable, String) void

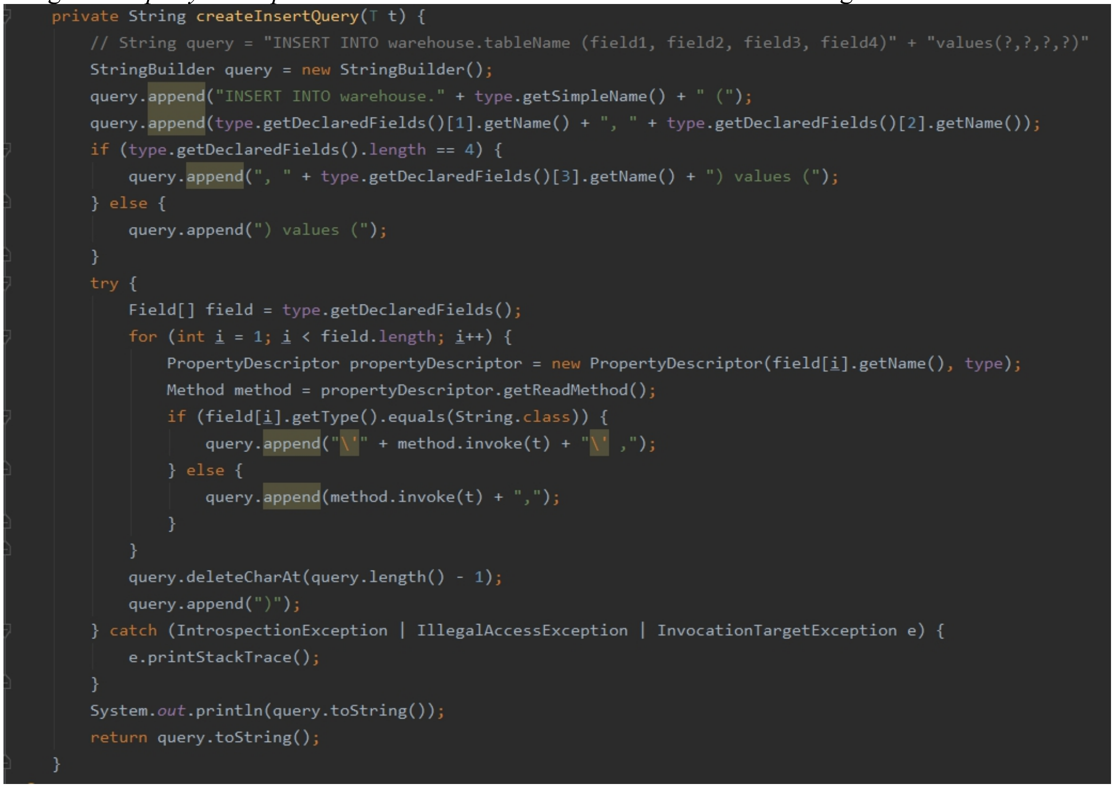
AbstractDAO class:

It is a generic class that defines the common operations for accessing the database: *find, insert, update, delete*, which will be discussed in further details later in the presentation. The class uses as parameter *<T>* which is basically any Java Model Class that is mapped to the database, and has the same name as the table and the same instance variables and data types as the table fields.
The constructor obtains the class of generic type T.
The *createSelectQuery(field: String): String* method builds the following query: "SELECT * FROM warehouse.*tableName* WHERE field =?", where the table name is taken through *type.getSimpleName()*.



```
public String createSelectQuery(String field) {
    // String query = "SELECT * FROM warehouse.tableName WHERE field =?"
    StringBuilder sb = new StringBuilder();
    sb.append("SELECT ");
    sb.append(" * ");
    sb.append(" FROM warehouse.");
    sb.append(type.getSimpleName());
    sb.append(" WHERE " + field + " =?");
    return sb.toString();
}
```

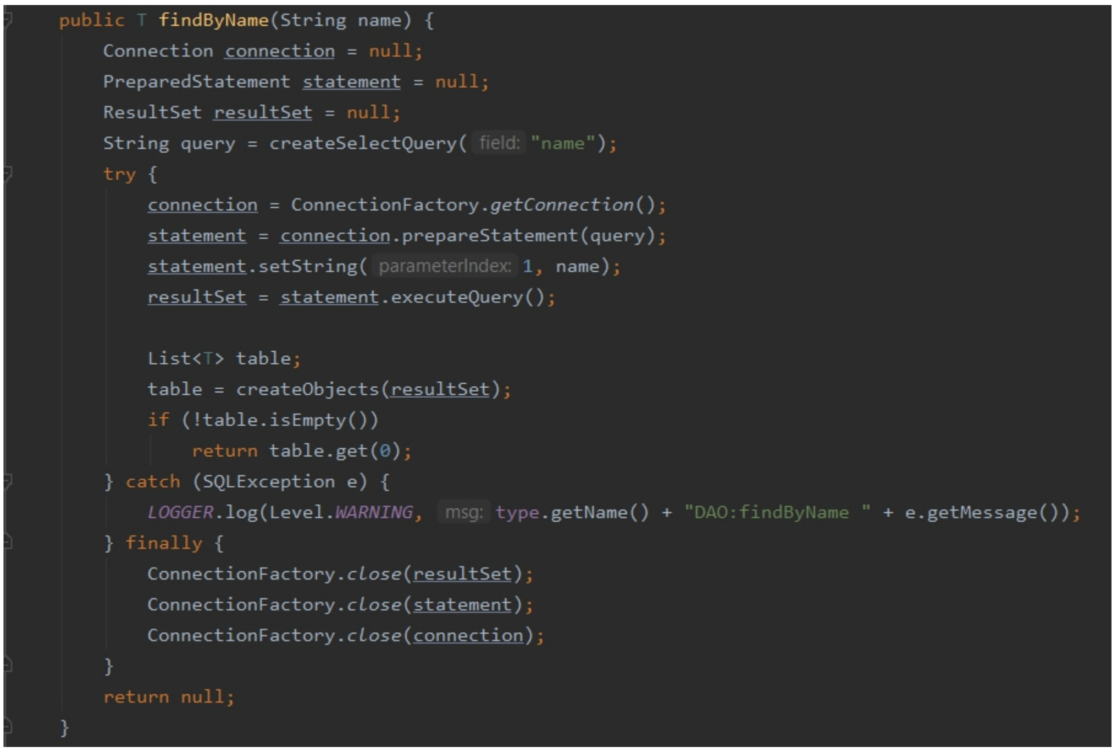
The *createSelectQuery2(field1: String, field2: String):String* method builds a similar query, but with fields for identification.
The *createInsertQuery(t: T): String* method builds a generic insert query given the model object parameter *t* and returns ”INSERT INTO warehouse.tableName (*field1, field2, field3, field4*) values (?, ?, ?, ?)”. The fields will be replaced b the corresponding column names int he table by calling the method *getDeclaredFields()* and the question marks will be replaced by the corresponding values by using the *PropertyDescriptor* and *Method* to reach and read the content of the given *t*.



```java
    private String createInsertQuery(T t) {
        // String query = "INSERT INTO warehouse.tableName (field1, field2, field3, field4)" + "values(?,?,?,?)"
        StringBuilder query = new StringBuilder();
        query.append("INSERT INTO warehouse." + type.getSimpleName() + " (");
        query.append(type.getDeclaredFields()[1].getName() + ", " + type.getDeclaredFields()[2].getName());
        if (type.getDeclaredFields().length == 4) {
            query.append(", " + type.getDeclaredFields()[3].getName() + ") values (");
        } else {
            query.append(") values (");
        }
        try {
            Field[] field = type.getDeclaredFields();
            for (int i = 1; i < field.length; i++) {
                PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field[i].getName(), type);
                Method method = propertyDescriptor.getReadMethod();
                if (field[i].getType().equals(String.class)) {
                    query.append("\'" + method.invoke(t) + "\' ,");
                } else {
                    query.append(method.invoke(t) + ",");
                }
            }
            query.deleteCharAt(query.length() - 1);
            query.append(")");
        } catch (IntrospectionException | IllegalAccessException | InvocationTargetException e) {
            e.printStackTrace();
        }
        System.out.println(query.toString());
        return query.toString();
    }
```

The *createDeleteQuery(id: int): String* and *createUpdateQuery(t: T, id: int): String* are implemented in similar way and return the strings: “DELETE FROM warehouse.tableName WHERE id = ?” and “UPDATE warehouse.tableName SET field2 = ?, field3 = ?, field4 = ? WHERE id = ?”. As it can be noticed, the first field is never inserted because the id field of every table is incremented automatically, so there is no need to worry about finding the last id in order to insert a new element.

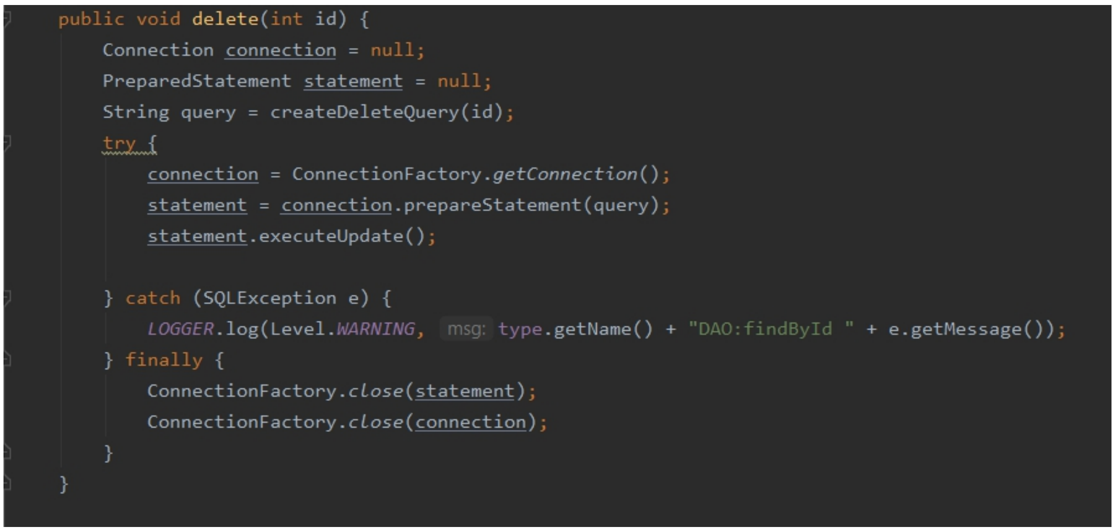
Next three methods have similar functionality, namely they are used to find and return either all the entries in a table or those with a specific id/name given as a parameter. In the picture below is shown the implementation of the last one where *name* represents either the full name of a client or the name of a product. The idea is that this method can be useful just for the *client* table and *product* table, but the other two are functional for all the tables in the database.
The approach used is easy to understand:
- the method builds the query needed to search in the table;
- obtains the connection to the database;
- sets the statement with the parameter given;
- executes the query;
- returns the entry if any;
- closes the statement and connection.



```java
public T findByName(String name) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    String query = createSelectQuery( field: "name");
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.setString( parameterIndex: 1, name);
        resultSet = statement.executeQuery();

        List<T> table;
        table = createObjects(resultSet);
        if (!table.isEmpty())
            return table.get(0);
    } catch (SQLException e) {
        LOGGER.log(Level.WARNING,  msg: type.getName() + "DAO:findByName " + e.getMessage());
    } finally {
        ConnectionFactory.close(resultSet);
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
    return null;
}
```

The next functions, namely *insert(t: T), update(t: T, id: int), delete(id: int)* are similar to the one above. An exemplification is given below. It basically follows the same steps:
- the method builds the query needed to search in the table;
- obtains the connection to the database;
- sets the statement with the parameter(s) given;
- executes the statement(query);
- closes the statement and connection.



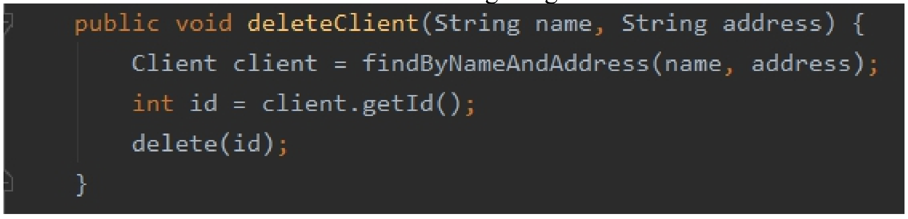
```java
public void delete(int id) {
    Connection connection = null;
    PreparedStatement statement = null;
    String query = createDeleteQuery(id);
    try {
        connection = ConnectionFactory.getConnection();
        statement = connection.prepareStatement(query);
        statement.executeUpdate();

    } catch (SQLException e) {
        LOGGER.log(Level.WARNING,  msg: type.getName() + "DAO:findById " + e.getMessage());
    } finally {
        ConnectionFactory.close(statement);
        ConnectionFactory.close(connection);
    }
}
```

The last method *resetAutoincrement(tableName: String)* can be used by any table and has the responsibility to reset the autoincrement of each table, so it keeps a smooth id numbering after each deletion of an entry in the table. The functionality is similar to the one above, except for the fact that the built query is “ALTER TABLE warehouse.*tableName* AUTO_INCREMENT = 1”.

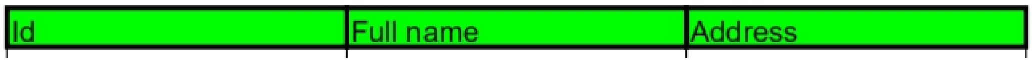
ClientDAO class:

Regarding the database, it has specific methods for finding a client in the table (i.e. *findByNameAndAddress(name: String, address: String): Client* which has the same implementation as the methods in AbtractDAO for finding a specific entry, but with two conditions) and for deleting a client (shown below). Because of the fact that, for deleting a client, the given input is the name and the address I decided to do the deletion the following way: find the client by name and address, then get the id of the searched client and delete it using the generic *delete* method.



```
public void deleteClient(String name, String address) {
    Client client = findByNameAndAddress(name, address);
    int id = client.getId();
    delete(id);
}
```

Next, three more methods are used to generate the PDF reports which will include a tabular form with all the current clients in the database with the following header:

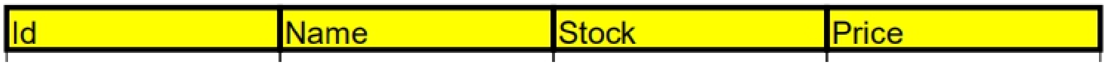


Id	Full name	Address

For these methods, there have been imported *itextpdf* library to help create a document, give an arbitrary name to that report, create a customisable table in the document, build and populate the rows with the data needed. For populating the table, there has been used the *findAll* method to get all the entries in the *client* table. To avoid overriding of a report, I used a private instance variable called *numberOfReport* which is initialized with 0 in the constructor and will be incremented each time a new report is generated. This number can help the manager keep track of the order in which the reports where generated.

ProductDAO class:

This class has access to all the methods in the AbstractDAO class. Moreover, for this class there has been used the same methods for generating a PDF file as for the ClientDAO class, but this time the header looks like this:



Id	Name	Stock	Price

ItemDAO class:

This class has access to all the methods in the AbstractDAO class as well. The only additional method it has is the *findItemByProductId(productId: int): Item* which basically uses the same implementation as the *findById* in AbstractDAO, but with the difference that the field we are looking for is called *productId* which will help get translate the id from order class.

OrderDAO class:

This class has access to all the methods in the AbstractDAO class, but has one more useful method, namely *findSection(fieldName: String, id: int): List<Order>* which is built to find just a specific section of the entries in the table. I implemented this method to find the orders of a client. Since a client can order multiple products and all the orders are stored in the order table, I have chosen to find the orders of a specific client this way. In this class there has been implemented 3 ways of generating data about the orders:

1. A report on all the orders made until now
2. A report on the orders of a specific client
3. A report with an error of under-stock message.

We will discuss them briefly. (1) is done similar way as the one for the client and product. A PDF file in tabular form with all the orders is generated. It has the following header:

Id	Client	Item	Total price

The table is easy to read and understand. (2) will be generated automatically each time a client orders any product that is available in the stock. The header is the following:

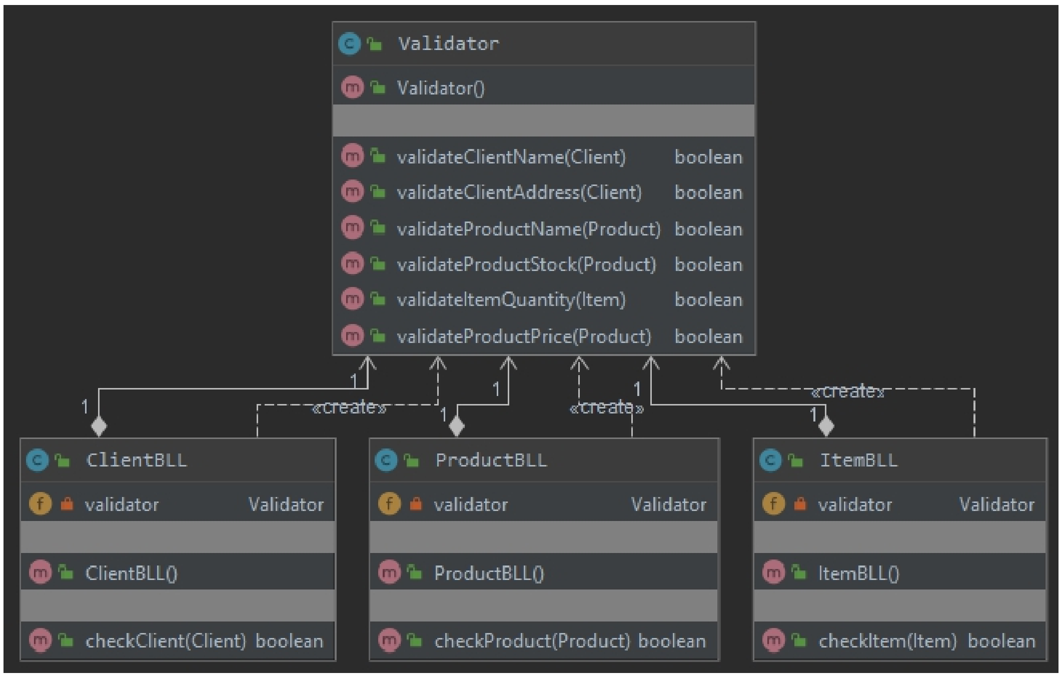
Client name	Ordered item	Quantity	Total price

(3) In case a client orders a product and that product is insufficient in the stock, the order will not be considered, so it will not be inserted in the database and that client will receive a personalized bill like in the example
below.

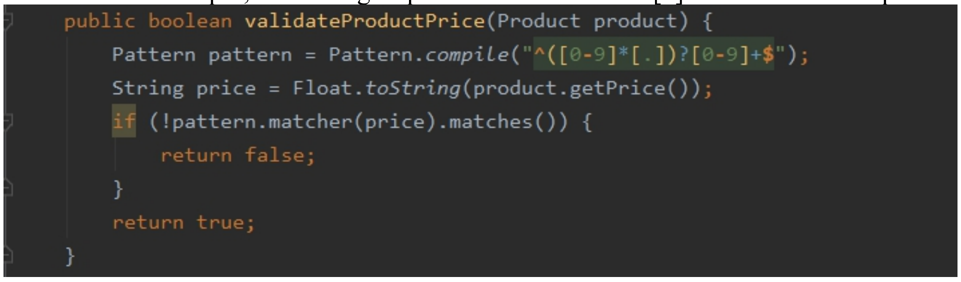
Error message: insufficient stock of apple to process order of Sandu Vasile

For each report there has been a suggestive name and has been used a different color to differentiate them.

<u>Package business_logic:</u>


Validator
Validator()
validateClientName(Client) boolean
validateClientAddress(Client) boolean
validateProductName(Product) boolean
validateProductStock(Product) boolean
validateItemQuantity(Item) boolean
validateProductPrice(Product) boolean
«create»
«create»
«create»
1
1
1
1
1
1
ClientBLL
validator Validator
ClientBLL()
checkClient(Client) boolean
ProductBLL
validator Validator
ProductBLL()
checkProduct(Product) boolean
ItemBLL
validator Validator
ItemBLL()
checkItem(Item) boolean

<u>Validator class:</u>
There has been used methods to validate the spelling of each input in the command line. For validating the format of the input, I used RegEx patterns and matchers. [3] Let's see an example:

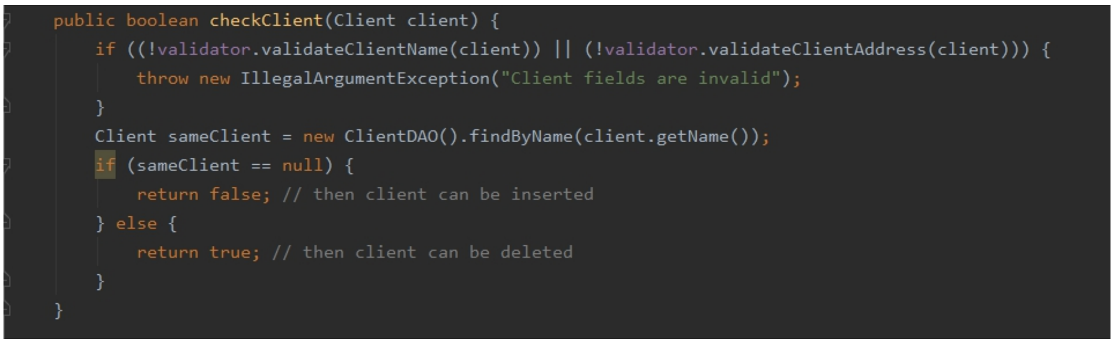


```java
public boolean validateProductPrice(Product product) {
    Pattern pattern = Pattern.compile("^([0-9]*[.])?[0-9]+$");
    String price = Float.toString(product.getPrice());
    if (!pattern.matcher(price).matches()) {
        return false;
    }
    return true;
}
```

For the price, there has been used a pattern to recognize just the input or the float.

ClientBLL class:

For this class there has been invoked a validator and the method looks like this:

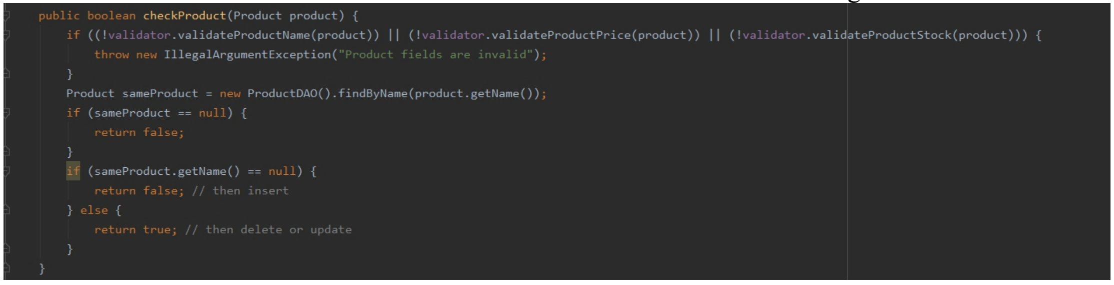


```java
public boolean checkClient(Client client) {
    if ((!validator.validateClientName(client)) || (!validator.validateClientAddress(client))) {
        throw new IllegalArgumentException("Client fields are invalid");
    }
    Client sameClient = new ClientDAO().findByName(client.getName());
    if (sameClient == null) {
        return false; // then client can be inserted
    } else {
        return true; // then client can be deleted
    }
}
```

It tests where the name and address are correct and then searches for the given client in the database. If the client exists, he can be deleted, otherwise, he can be inserted.

ProductBLL class:

For this class there has been invoked a validator and the method is the following:

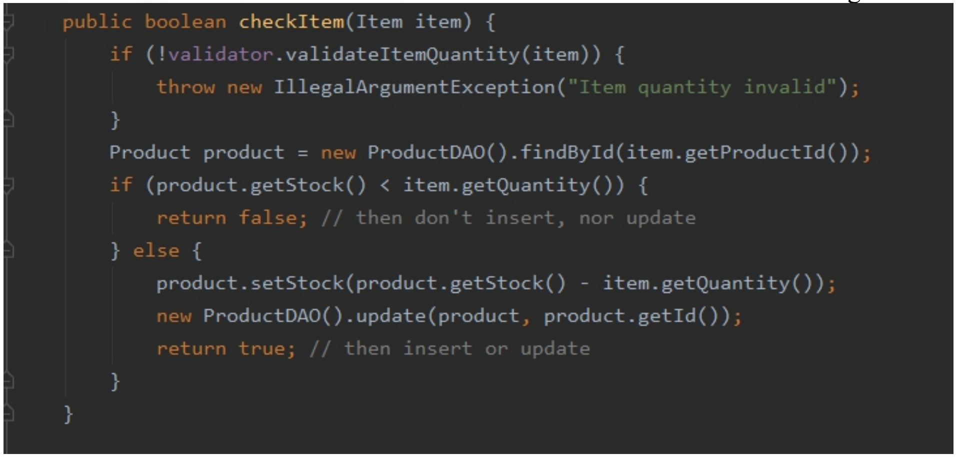


```java
public boolean checkProduct(Product product) {
    if ((!validator.validateProductName(product)) || (!validator.validateProductPrice(product)) || (!validator.validateProductStock(product))) {
        throw new IllegalArgumentException("Product fields are invalid");
    }
    Product sameProduct = new ProductDAO().findByName(product.getName());
    if (sameProduct == null) {
        return false;
    }
    if (sameProduct.getName() == null) {
        return false; // then insert
    } else {
        return true; // then delete or update
    }
}
```

It tests the given name, stock and price and return false in case of a new product, so it can be inserted and false otherwise, so the product can be either deleted or its quantity can be updated.

ItemBLL class:

For this class there has been invoked a validator and the method is the following:

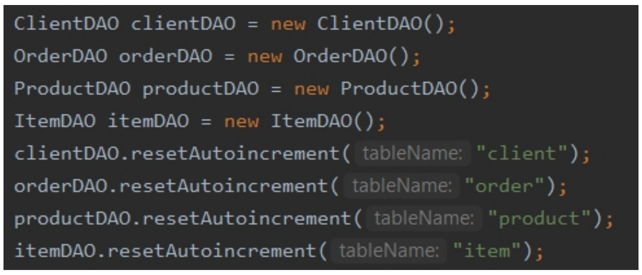


```java
public boolean checkItem(Item item) {
    if (!validator.validateItemQuantity(item)) {
        throw new IllegalArgumentException("Item quantity invalid");
    }
    Product product = new ProductDAO().findById(item.getProductId());
    if (product.getStock() < item.getQuantity()) {
        return false; // then don't insert, nor update
    } else {
        product.setStock(product.getStock() - item.getQuantity());
        new ProductDAO().update(product, product.getId());
        return true; // then insert or update
    }
}
```

It tests the quantity and returns false in case there are not enough available products in the stock to fulfill the order and true otherwise, so the order can be processed. The method updates the stock of the product as well.

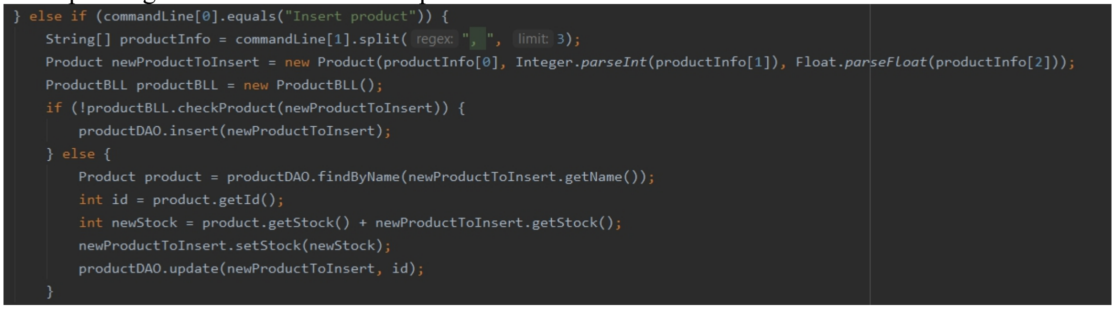
<u>Package parser:</u>

It includes just one class, i.e. *Parser*. Here reading the command from the text file happens and the start of the program as well. First of all, the direct access to all the tables in the database is initialized and the autoincrement is reset in order to have a smooth numbering each time we run the program.



```
ClientDAO clientDAO = new ClientDAO();
OrderDAO orderDAO = new OrderDAO();
ProductDAO productDAO = new ProductDAO();
ItemDAO itemDAO = new ItemDAO();
clientDAO.resetAutoincrement( tableName: "client");
orderDAO.resetAutoincrement( tableName: "order");
productDAO.resetAutoincrement( tableName: "product");
itemDAO.resetAutoincrement( tableName: "item");
```

Next, each line is split into the command name and the data which needs to be inserted into the corresponding table. Let’s see an example.

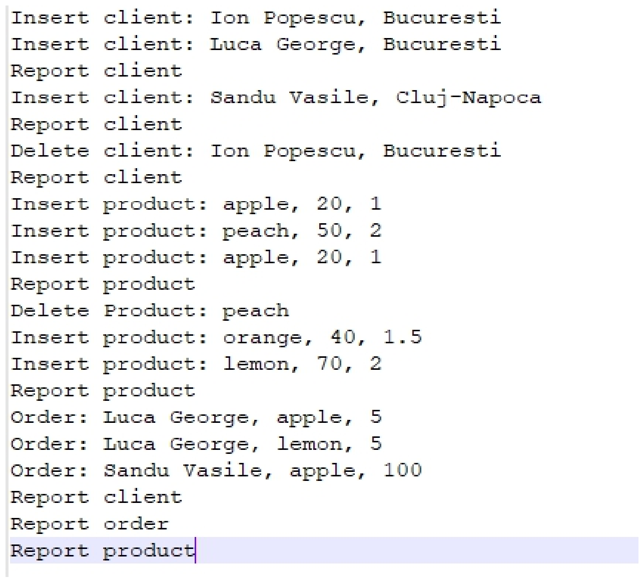


```
} else if (commandLine[0].equals("Insert product")) {
    String[] productInfo = commandLine[1].split( regex: ", ",  limit: 3);
    Product newProductToInsert = new Product(productInfo[0], Integer.parseInt(productInfo[1]), Float.parseFloat(productInfo[2]));
    ProductBLL productBLL = new ProductBLL();
    if (!productBLL.checkProduct(newProductToInsert)) {
        productDAO.insert(newProductToInsert);
    } else {
        Product product = productDAO.findByName(newProductToInsert.getName());
        int id = product.getId();
        int newStock = product.getStock() + newProductToInsert.getStock();
        newProductToInsert.setStock(newStock);
        productDAO.update(newProductToInsert, id);
    }
```

Here the command is “insert product”. A new product is created with the given input. The content of the product is checked through the ProductBLL. If the product do not exist already in the database, it is inserted. Otherwise, it is searched by name in the product table and its stock is updated by summing the old stock with the new one given. And so on and so for all the commands are done. The input is translated, checked and the necessary changes in the database are don, and the requested report are generated.

5. Results

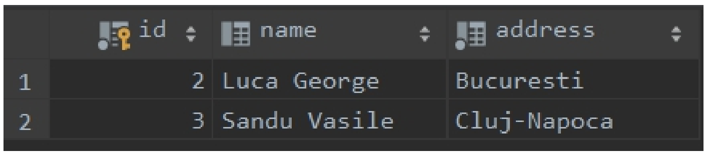
After executing the commands below:



```
Insert client: Ion Popescu, Bucuresti
Insert client: Luca George, Bucuresti
Report client
Insert client: Sandu Vasile, Cluj-Napoca
Report client
Delete client: Ion Popescu, Bucuresti
Report client
Insert product: apple, 20, 1
Insert product: peach, 50, 2
Insert product: apple, 20, 1
Report product
Delete Product: peach
Insert product: orange, 40, 1.5
Insert product: lemon, 70, 2
Report product
Order: Luca George, apple, 5
Order: Luca George, lemon, 5
Order: Sandu Vasile, apple, 100
Report client
Report order
Report product
```

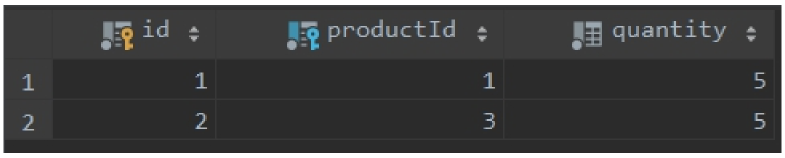
Each table in the data base is populated in the following way:

Client:



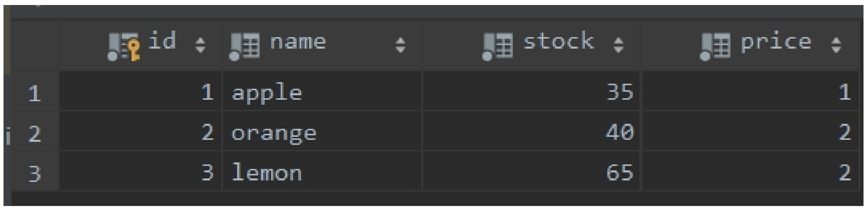
	id	name	address
1	2	Luca George	Bucuresti
2	3	Sandu Vasile	Cluj-Napoca

Item:



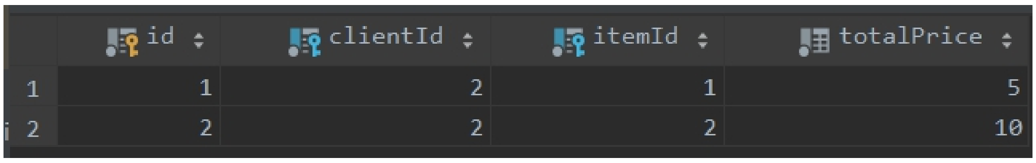
	id	productId	quantity
1	1	1	5
2	2	3	5

Product:



	id	name	stock	price
1	1	apple	35	1
2	2	orange	40	2
3	3	lemon	65	2

Order:



	id	clientId	itemId	totalPrice
1	1	2	1	5
2	2	2	2	10

A better understanding of these tables can be found in the PDF files:
- ClientReport3.pdf
- OrderReport2.pdf
- Productreport2.pdf

6. Conclusions

Lessons learned:
- better understanding of the reflection techniques
- how to write and generate JavaDoc files
- how to write, insert table, customize tables and generate file in PDF format
- how to generate SQL dump file

Possible feature improvements:
- the price of the product can be updated as well
- a new table with the order details can be implemented, containing the exact address of the client, the date of ordering, a phone number
- a new command which can enable a client to cancel one of his orders

7. Bibliography

[1] List:
https://docs.oracle.com/javase/8/docs/api/java/util/List.html
[2] Bitbucket:
https://bitbucket.org/utcn_dsrl/pt-reflection-example/src/master/
[3] RegEx:
https://www.regextester.com