

*Faculty of Automation and Computer Science*

**Order Management**

Project made by  
Ștefan Ciuprina

Contents

[1. Objective 3](#_Toc37738026)

[2. Problem analysis, scenarios, use cases 3](#_Toc37738027)

[3. Project Design 4](#_Toc37738028)

[3.1 Approach 4](#_Toc37738029)

[3.2 UML Diagram 5](#_Toc37738030)

[3.3 Packages 6](#_Toc37738031)

[3.4 Classes’ Design 6](#_Toc37738032)

[3.4 Data Structures 7](#_Toc37738033)

[4. Implementation 7](#_Toc37738034)

[5. Results 9](#_Toc37738035)

[6. Conclusions 10](#_Toc37738036)

[7. Bibliography 10](#_Toc37738037)

# Objective

The objective of this project is to create an application capable of managing customer orders for a warehouse. The following list contains the secondary objectives which will make possible the creation of this project:

* Parsing from a text file given as command line argument the commands of inserting/deleting a client/product in/from the database, ordering an amount of products and reporting the data;
* Exporting a bill each time an order is being made, an understocked notice each time someone tries to order more items of a product than they actually are and reports, all as pdf files;
* Establishing a connection with the local database to store the data and managing it using SQL queries through the program; 4 tables will be created in our database: clients, products, orders, order\_details;
* Developing the program using *Layered Architecture*;
* Documenting the entire project using *JavaDoc;*

# Problem analysis, scenarios, use cases

The functioning of this program will be as it follows: the user creates a text file containing one or more commands, writing each new command on a new line. After that, the user runs in the terminal the jar file with his/hers text file as argument. The actions of his/hers commands will update the database and/or generate one or more pdf files. The possible commands (and their syntax) one can enter are:

* Insert client: *name*, *address*
* Delete client: *name*
* Insert product: *name*, *quantity*, price
* Delete product: *name*
* Order: *client\_name*, *product*, *quantity*
* Report client
* Report product
* Report order

The possible pdf files that might be generated as a result of entering the commands are:

* report\_clients
* report\_products
* report\_orders
* bill\_order\_no
* understocked\_*product\_name*

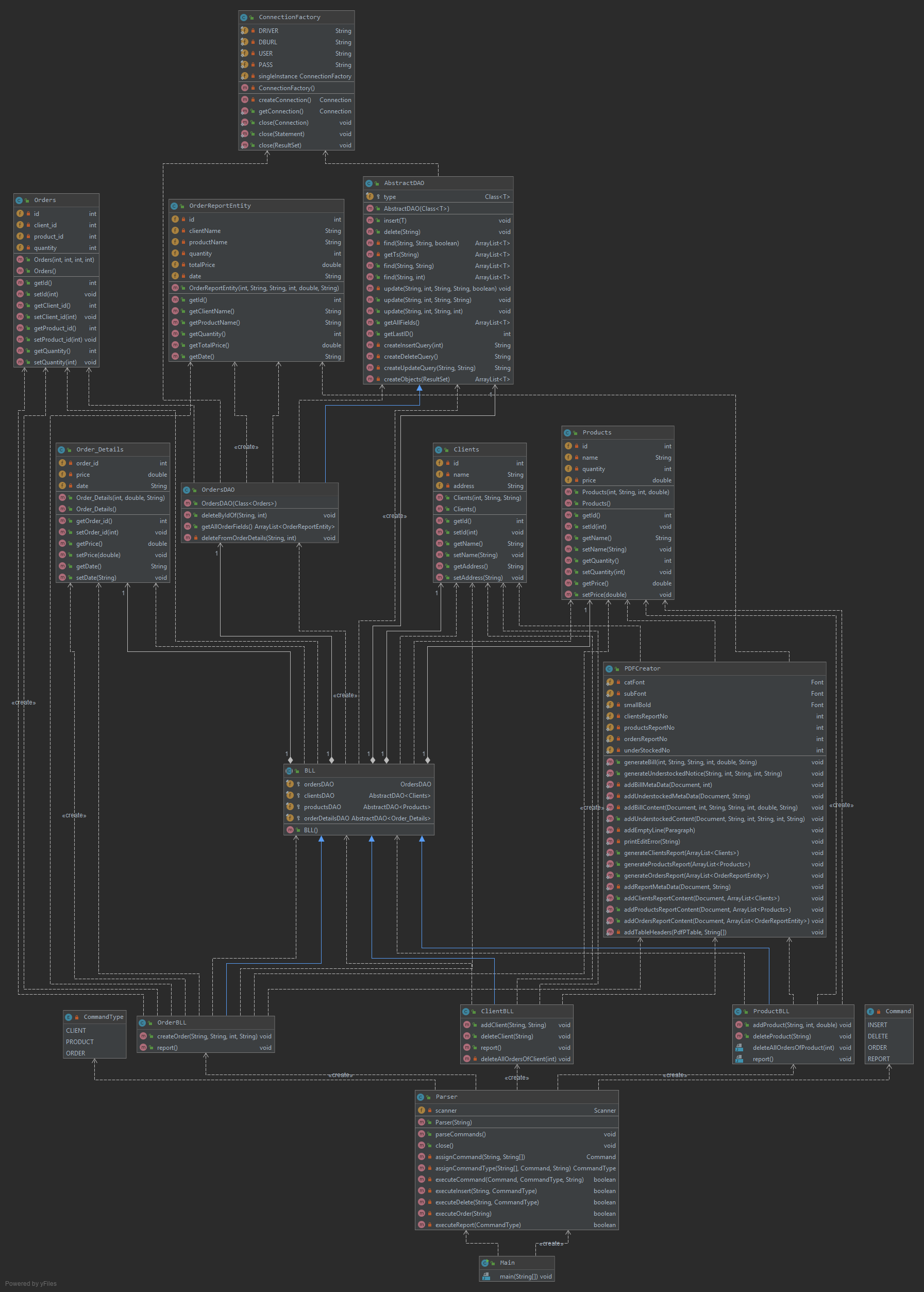
If one or more commands are typed incorrectly by the user or fail to execute, one or more error messages will be printed in the terminal.

# Project Design

## Approach

As mentioned in **chapter 1**, our project follows the *Layer Architecture* technique. Therefore, there are six main packages in our project: *businessLayer* , *dao*, *dataAccessLayer*, *model*, *presentation*, *start*. The *presentation* class also has an inner class called *reportEntities*. The input file will be taken as a command line argument in the main function located in the *start* package, sent to the *presentation* package for parsing the commands. From there, the *businessLayer* will identify the action that must be taken, attempt to do it (it may not be successful, printing an error in this case) and print the possible . If the command can be executed, the *dao* package will take care of the SQL statements and update the database, connecting through it using the *dataAccessLayer*. More information regarding the packages is presented in **section 3**.

## UML Diagram



## Packages

As mentioned before, our project is divided in the 6 packages:

* *start*: contains the *Main* class, used for starting the application. It sends the input text file received as command line argument to the *presentation* package, where a *Parser* class will read each line and identify the type of command that is needed to be used;
* *model*: contains class entities for each field of the table: *Clients, Products, Orders, Order\_Details*, having the exact same fields as entries in the database tables;
* *presentation*: contains the *Parser* class, the *PDFCreator* class, which is responsible for creating pdf files for bills, understocked notices and reports;
* *reportEntities*: contains a special entity class called *OrderReportEntity*, a class used when we want to report the orders. It was created since the orders table contains the following entries: *id, client\_id, product\_id, quantity*, and when we want to report the orders, we want to show the full information. That’s why, using the *client\_id* and *product\_id* entries we will get the client and product names respectively, and using the order\_details table, we will also aquire the total price for the order and its date and time. Therefore, our report will present: *id, client name, product name, quantity, total price, date and time*;
* *businessLayer*: provides business logic classes, giving easy access to the *Parser* to execute each command, taking the responsibility of handling the database manager (the *dao* package); Also, if needed, the classes in this package will call the *PDFCreator* to output bills, understocked notices and/or reports;
* *dao*: handles the database queries needed to update the database;
* *dataAccessLayer*: establishes the connection between the application and the database.

## 3.4 Classes’ Design

The Model package has 3 classes: *Monomial, Polynomial* and *Model.* The *Model* class uses polynomials created using the *Polynomial* class, which are made of multiple monomials created using the *Monomial* class. The View and the Controller packages each contain only a class having the same name, each one of them taking care of its role presented in the previous sections. The Controller class also contains an inner class, responsible with the *action listener* of a button in the view (more on that in **section** 4). The *PolynomialCalculator* class is in neither of those 3 packages, being present only in the *polynomialcalculator* package . It calls the main classes in order to make the program runnable.

There are a total of 15 classes in our project. Eight of them have already been described in the previous section(*Main, Parser, PDFCreator, Clients, Products, Orders, Order\_Details –* the last four are the model classes –, *OrderReportEntity*).

The business logic classes, present in the *businessLayer* package, are giving easy access for execution of each command, as said before. There are 4 classes in these package: the abstract class *BLL* (declared abstract since we shouldn’t create an object of it), which, when instantiated, creates a ”DAO” class for each entity – clients, products, orders, order\_details – (more on DAO classes in the next paragraph). The other 3 classes are *ClientBLL, ProductBLL, OrderBLL*, classes which inherit from the *BLL* abstract class, the first two handling the operations of insertion and deletion of clients and products, and the third taking care of creating orders. Also, all of them are capable of calling the *PDFCreator* class in order to make a report.

The dao classes, present in the *dao* package, are the ones responsible with the direct access to the database, through the *ConnectionFactory* class, present in the *dataAccessLayer* package. According to the data received from the business logic classes, the dao classes create and execute queries for inserting, deleting, updating or finding data in the database. As mentioned earlier, the *ConnectionFactory* class establishes the connection between the program and the local database. It provides a set of static methods, not needing to instantiate the class when its use is needed (its instantiation is even prohibited due to the private constructor).

## Data Structures

The data structure mostly used in this project is the *ArrayList*. It is used by the DAO classes, returning a list of elements when the *find* methods are being called. It is also used in the business logic classes when creating reports, since we have to store first all the entries from the specific table in an entity type class.

# Implementation

In this section I will present the most important parts of the project that made its realization possible.

Probably the most important feature added to this project is the use of *reflection techniques*. This was done when creating the classes for the *dao* package. Instead of making a class for each entity (client, product, orders, order\_details), a generic class, called *AbstractDAO<T>* , has been created, which is able of performing the query operations on any table, depending on the type T, where T must be an entity class of a table (T is, this way, also the name of the table). An example of the use of reflection is given below:

public void delete(String name) {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 String query = createDeleteQuery();  
 try {  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query);  
 statement.setString(1, name);  
 statement.executeUpdate();  
 } catch(SQLException e) {  
 e.printStackTrace();  
 } finally {  
 if(connection != null) ConnectionFactory.*close*(connection);  
 if(statement != null) ConnectionFactory.*close*(statement);  
 }  
}

private String createDeleteQuery() {  
 return "DELETE FROM " + type.getSimpleName() + " WHERE name = ?";  
}

As it can be seen, the name of the table to be written in the table is given by the invocation of the method *getSimpleName()* for the field *type* (which holds the type T mentioned above). So, either the class that is sent to the generic class, the query will be created accordingly.

However, *AbstractDAO* is not the only class present in the *dao* package. Another class had to be created, namely *OrdersDAO*, its creation being needed due to the need of extra functionality of the orders table (to also keep the functionality of the *AbstractDAO* class, *OrdersDAO* extends from *AbstractDAO<Orders>*. As mentioned earlier, when creating a report for the orders, the contents of the orders table is not enough. Those are: *id, client\_id, product\_id, quantity,* while the ones we want for the report are *id, client name, product name, quantity, total price, date*. The client name and the product name are got from their specific tables (clients and products respectively) by identifying them using the their specific ids. The total price and date, however, are got from the order\_details table, which has the following fields: *order\_id, price, date*, the order id entry being correspondent to an id from the orders table.

To get all this information altogether, three queries had to be formed. They have been put inside a string array and each one executed one after another, collecting the necessary data. This has been done in the *getAllOrderFields()* method from the *OrdersDAO* class. The points we have mentioned can be seen below:

public ArrayList<OrderReportEntity> getAllOrderFields() {

[…]  
for(Orders order : allOrders) {  
 String[] query = {"SELECT `name` FROM clients WHERE id = " + order.getClient\_id(),  
 "SELECT `name` FROM products WHERE id = " + order.getProduct\_id(),  
 "SELECT price, `date` FROM order\_details WHERE order\_id = " + order.getId()};

[…]  
for(int i = 0; i < 3; i++) {  
 statement = connection.prepareStatement(query[i]);  
 resultSet[i] = statement.executeQuery();  
 resultSet[i].next();  
}

Further implementation details are presented in the project using *JavaDoc*. The description of each class and method could be seen in the *JavaDoc* files as well as in the project itself.

# Results

After the implementation of the project has finished, a jar file was created to make possible the specification of the input text file as a command line argument. Afterwards, a *command.txt* file was created with the following input:

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

The output pdfs were as expected, the program generating four client reports, three product reports, one order report, two bills and one understocked notice pdf.

# Conclusions

All of the objectives presented in **section 1** have been achieved, the final form of the project being the wanted order management app, capable of handling actions on a database and outputting pdfs according to the request of the user. By working on this project, I learned the Layered Architecture technique of structuring the project, I familiarized myself more with working with SQL databases and their connection to java apps, acquired knowledge on reflection techniques, learned how to create PDF files through a java program and how to document a project using JavaDoc. Further improvements could be done to this project in the future, such as support for multiple commands, more information regarding the products and the clients and many more.

# Bibliography

* For learning to work with layered architecture, database connection and reflection techniques:  
  http://coned.utcluj.ro/~salomie/PT\_Lic/4\_Lab/Assignment\_3/Assignment\_3\_Indications.pdf
* SQL queries:  
  <https://www.w3schools.com/sql/>
* Creation of PDF Files through Java:  
  <https://www.vogella.com/tutorials/JavaPDF/article.html>
* JavaDoc:  
  [https://www.jetbrains.com/help/idea/working-with-code-documentation.html#](https://www.jetbrains.com/help/idea/working-with-code-documentation.html)