*Technical University of Cluj-Napoca*

*Faculty of Automation and Computer Science*

*2nd Semester 2019-2020*

**Programming Techniques**

**Homework 3**

Student: Osváth Tamás

Group: 30421

Supervising teacher: Pop Cristina

Contents

[1. Objectives 3](#_Toc37701131)

[2. Problem Analysis, Modelling, Scenarios, Use Cases 3](#_Toc37701132)

[2.1. Problem Analysis 3](#_Toc37701133)

[2.2. Modelling 4](#_Toc37701134)

[2.3. Scenarios, Use Cases 4](#_Toc37701135)

[3. Design 7](#_Toc37701136)

[3.1. Class Design, Packages and UML Diagram 7](#_Toc37701137)

[3.2. Relationships 9](#_Toc37701138)

[3.3. Algorithms and Data Structures 9](#_Toc37701139)

[4.Implementation 10](#_Toc37701140)

[5.Usage and Testing 15](#_Toc37701141)

[6.Conclusions 15](#_Toc37701142)

[7.Bibliography 16](#_Toc37701143)

# Objectives

The main objective is to design and implement an Order Management application for processing customer orders from a warehouse.

Relational Databases are used to store the products, the clients and the orders. The application is structured in packages using a layered architecture and uses the following classes:

* Model classes - represent the data models of the application
* Business Logic classes - implement the application logic
* Presentation classes - classes that implement the user input/output
* Data access classes - classes that contain the access to the database

Another objective is to use reflection techniques to create a generic class that contains the methods for accessing the Database: create object, edit object, delete object and find object. The queries for accessing the DB for a specific object that corresponds to a table will be generated dynamically through reflection.

As secondary objectives we can enumerate:

* Create queries in order to work with the data in the Database
* Design the tables from the Database
* Implement the parser
* Create a connection between the SQL Database and the java warehouse application

# Problem Analysis, Modelling, Scenarios, Use Cases

## Problem Analysis

An application for order management can be useful in almost every situation where a user can buy a certain product from a shop. Since nowadays the e-commerce is being used by almost everybody, a system like this can be widely used. This helps the seller, as well as the customer for achieving their task.

## Modelling

For such a system it is necessary to have a basic element: a table in which we can keep information about certain things. The application’s database will contain 3 tables: Client, Orders and Product. The application will be able to communicate with the database through a connection.

In order to maintain the data from the tables, the basic CRUD (Create, Read, Update, Delete) and some additional operations need to be implemented based on SQL queries.

An important aspect of the implementation is the creation of a generic class, which creates the needed queries for the CRUD operations by the reflection technique.

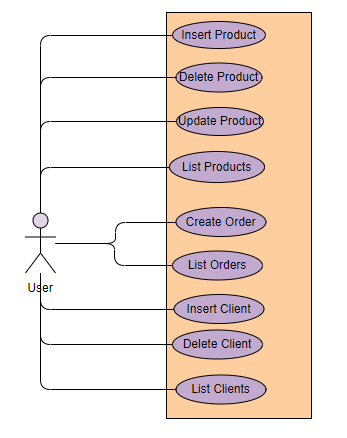
**IMPORTANT NOTE:** In my implementation, an order is considered to consist of one product, so for every order from the input file a bill is created. Inclusion of more products in an order can be considered as a future development of the application.

## Scenarios, Use Cases

Scenarios are real life situations, when some commands are given, and based on them, the database must be manipulated. The commands are given in a text file.

The user can insert, delete clients and products, can update a product’s price, can create orders and can generate reports on the clients, products and orders. The report consists in a PDF file, where the content of the corresponding table is shown. Also, when an order is placed, another PDF file will be generated: a bill with some fields if the order is successful, and an error bill if not.

The use case diagram is the following:



The User in this application can be an Administrator or a Client. Some use cases will be exemplified below.

Use case title: Adding a Client (Same for the Product)

Actor: Admin

Main success scenario:

1. Admin introduces data in the input file
2. Admin runs the application
3. Client is added to the Database

Main failure scenario:

1. Admin introduces (wrongly formatted) data in the input file
2. Admin runs the application
3. The Client is not added since the corresponding command is not valid

Use case title: Updating (increasing) a product’s stock

Actor: Admin

Main success scenario:

1. Admin introduces data in the input file (stock of the product is smaller than the order quantity)
2. Admin runs the application
3. The product’s stock is updated

Note: If the product doesn’t exist, it is created

Use case title: Placing an order

Actor: User

Main success scenario:

1. User introduces data in the input file (The client and the product exist, stock > order quantity)
2. User runs the application
3. The order is created, Bill in generated

Main failure scenario:

1. User introduces data in the input file (stock of the product is smaller than the order quantity)
2. User runs the application
3. The order is not created, ErrorBill is generated

Use case title: Deleting a Product

Actor: Admin

Main success scenario:

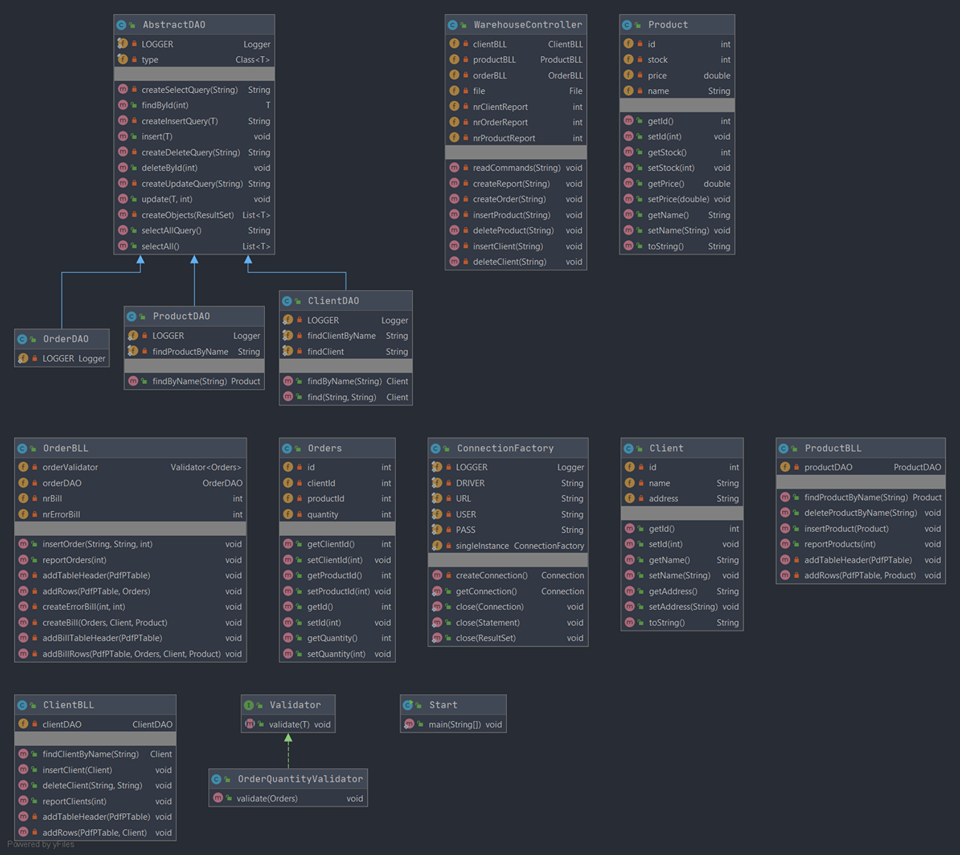
1. Admin introduces data in the input file (The Product must be in the database)
2. Admin runs the application
3. The product is deleted, so any order placed will be rejected, and an error bill will be generated

Main failure scenario:

1. Admin introduces data in the input file (The Product is not in the database)
2. Admin runs the application
3. The product is not deleted

# Design

## 3.1. Class Design, Packages and UML Diagram

Below can be found the Class Diagrams for all the classes in the project:

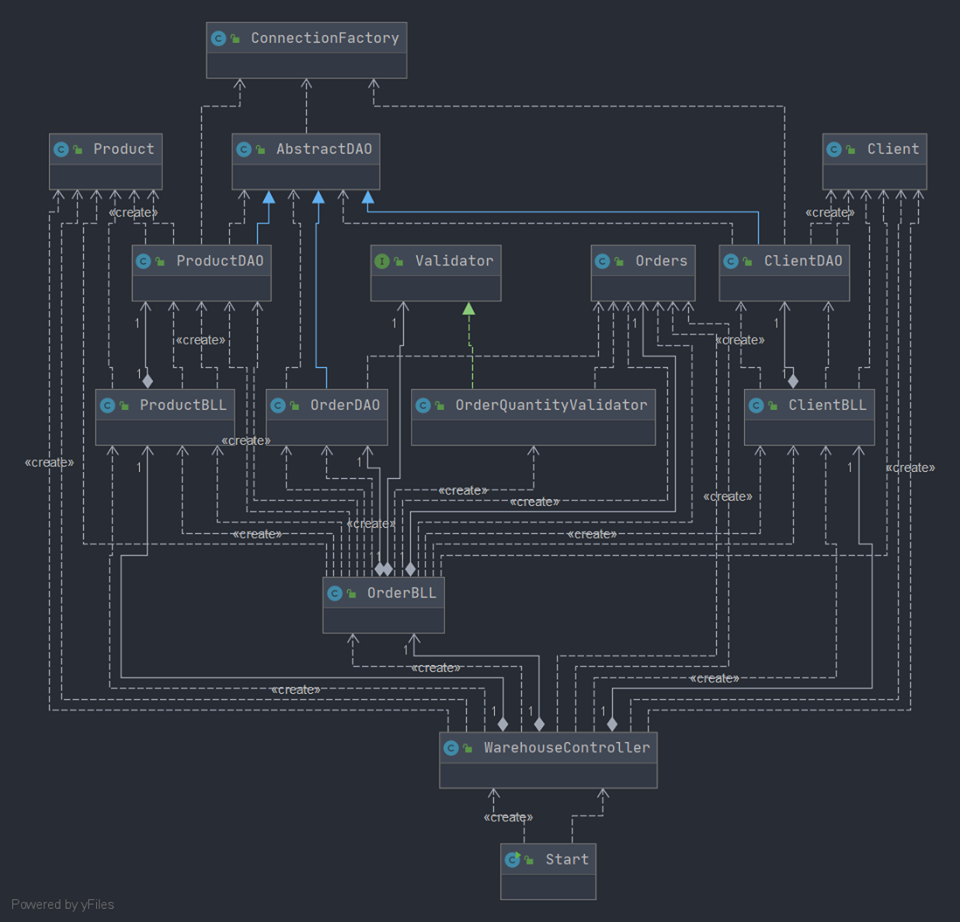
The project is built on a layered architecture. The packages are the following:

The bll package, as its name states, contains the Business Logic Layer of the application: ClientBLL, OrderBLL, ProductBLL. This package also contains another package, validators, with classes which validate some data: Validator interface, and its OrderQuantityValidator implementation.

The connection package contains a singleton class ConnectionFactory, responsible for creating a connection between the application and the database.

The model package contains the basic entities from the database: Client, Orders and Product. These are used for mapping the tables in the database, thus the attributes represent the table column names.

The dao package, as its name states, contains Data Access Objects. The AbstractDAO class creates the access to the database, implementing the CRUD (Create, Read, Update, Insert) operations. Also, there are the ClientDAO, ProductDAO and OrderDAO classes, which extend the AbstractDAO, adding some more functionalities.

The presentation package contains one package, controller, which has the WarehouseController class.

The start package contains one class, Start, which is basically the ‘Main’ of the application, here the controller is instantiated and begins the execution.

In the right side the UML diagram for the application can be seen.

## 3.2. Relationships

The relationships between the classes are: associations, dependencies and inheritances. Some of the most important ones are the following:

Inheritance can be found in the dao package, where the AbstractDAO class is extended by the ClientDAO, OrderDAO and ProductDAO classes.

The Validator interface is implemented by the OrderQuantityValidator class. Every BLL contains a list of validators

Between bll and dao classes there is an association relationship. The other relationships can be easily seen in the UML Diagram which is above.

## 3.3. Algorithms and Data Structures

There are no complex algorithms associated to some classes, each algorithm will be discussed in the implementation chapter.

The data structures used are mostly basic types like Integer, String. The price of a product is a Double.

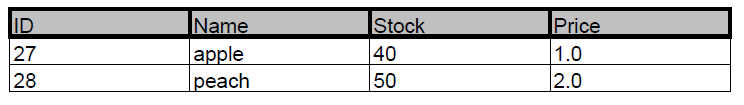
In order to be able to work with SQL databases, data structures like Connection, PreparedStatement and ResultSet are used.

For the creation of the PDF files, data structures like Document, PdfWriter, PdfPTable, PdfPCell are used.

**3.4. Design of the PDF**

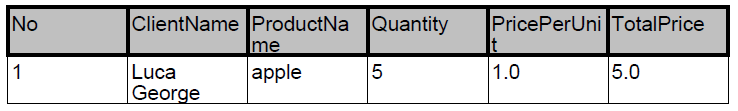
The PDF reports and bills are designed in a minimalistic manner, thus it is easier for the user to understand their meaning.

The report PDFs are made in a table manner, the header containing every column’s name. An example of a PDF is shown below:



To be mentioned that the ID’s are automatically generated by the database, so the developer doesn’t need to keep track of that.

The bill PDFs are also made in a table manner, but they contain more information in order to be meaningful and close to a real life bill. The bill contains the Client’s name, the product’s name, the quantity, the price per unit (product), and the total price. Also, as future development, the VAT could be calculated and added to a separate column, to be even closer to a real bill.



# 4.Implementation

In this chapter details of the implementation will be discussed. The purpose of this chapter is not to discuss every little detail of the implementation, only the most important ones.

* **MODEL**

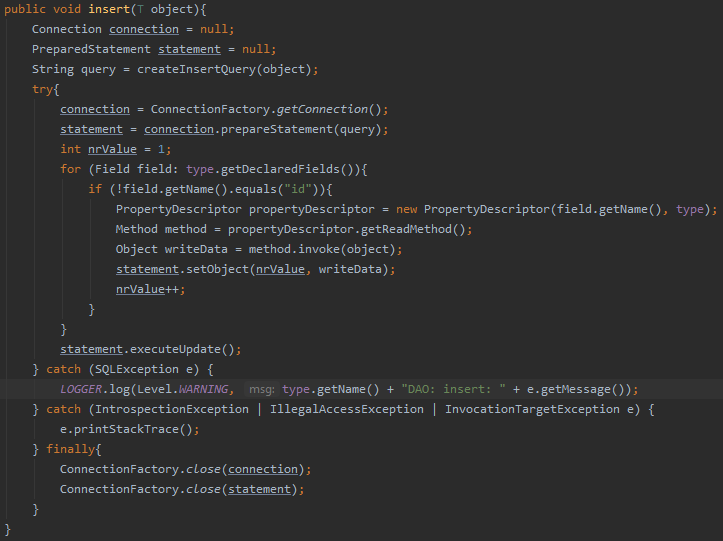
The Model contains the following classes, corresponding to the database’s tables: **Client, Orders, Product**. The Classes contain instance variables corresponding to the table’s columns, both as name and data type. There are more constructors implemented, with different parameter list, needed in some operations. Also, for each instance variable there are getters and setters implemented, to be able to retrieve and modify them.

* **DATA ACCESS**

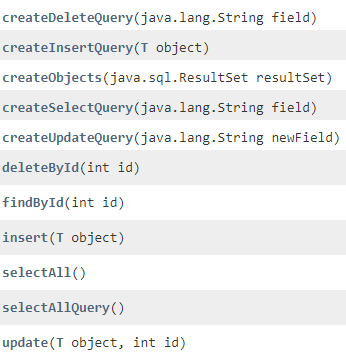
The **ConnectionFactory** class is responsible for the connection of the project and database. Because only one connection is needed at a time, the Singletion pattern is used. The class contains the getConnection() static method, which returns the connection. There are also methods for closing the Connection, the Statement and the ResultSet.

The classes which work with querys are AbstractDAO, ClientDAO, OrderDAO and ClientDAO.

The **AbstractDAO** is a generic class which implements methods for the CRUD (Create, Read, Update, Delete) operations. These methods can be used by any class which extends this one. The class uses also the reflection technique.

Let’s take as an example the insert operation. Its query is obtained with the help of the reflection technique, which is also used in the insert method. The insert(T) method does the following: the connection to the database is made, the statement is prepared with the corresponding insertion query (createInsertQuery method). Since the id will be automatically generated, there is no need to give one, so only the other fields will be taken into consideration. With help of the reflection, the fields of the class are extracted, and also the corresponding values, and finally added to the database.

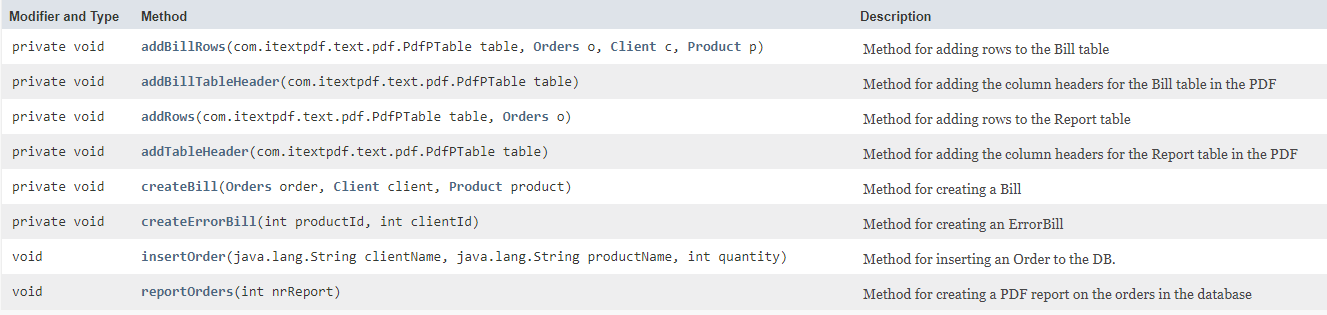
The methods of this class are the following, a short description of each can be found in the javadoc:



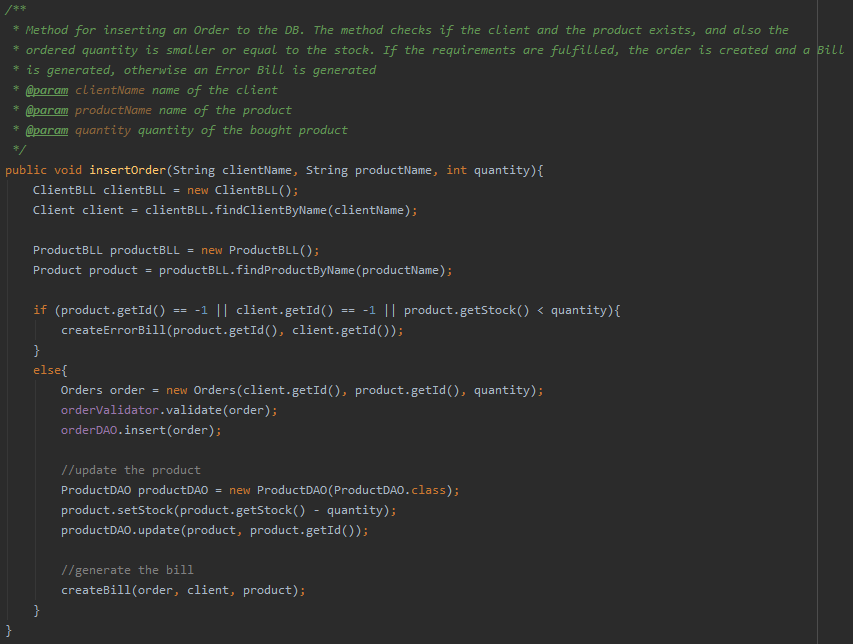
The **ClientDAO, ProductDAO and OrderDAO** classes inherit all the methods from the generic parent class, but they also have some specific functionalities. For example, the ClientDAO have 2 methods: findByName(name) for finding a client based on his name, and find(name, address) for finding a client based on his name and address, in case 2 clients have the same name.

* **BUSINESS LOGIC**

In the bll package there is a corresponding class for each class from the dao package (except the AbstractDAO, obviously). The role of this classes are to ensure that the user uses appropiately the data access methods. Each class contains a list of validators, although only one validator is implemented, OrderQuantityValidator, which checks if the ordered quantity is a valid number. As a future development of the project, other validators can be added, for example: validator for email, phone number, zip code, county, town and so on. These classes basically simply use the methods implemented in the dao classes, but adding some extra functionality, like generating PDF’s.

Let’s take as an example the **OrderBLL** class. As presented in the javaDoc, it has methods for inserting an order (insertOrder), reporting the orders (reportOrders), creating a bill (createBill) and creating an error bill (createErrorBill). Also, in order to create the tables in the PDF’s, there are some methods for adding table headers and rows.

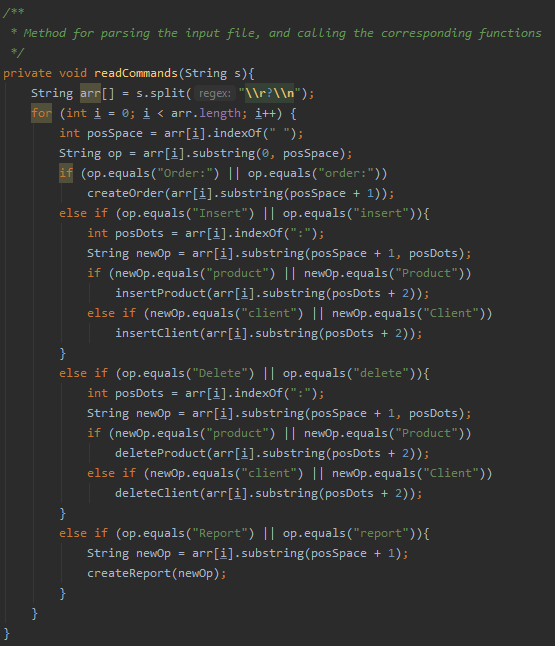
The insertOrder method has the following implementation:



The **ClientBLL** and **ProductBLL** are in the same manner implemented, with their corresponding methods. For more information, the javadoc can be consulted.

* **Presentation**

Since there is no Graphical User Interface, the presentation package contain only the controller package. The **WarehouseConroller** class basically creates a bond between the functionality (model) and the view (in our case, the view is represented by the generated PDFs). It is straightforward that this class contains instance variables for the ClientBLL, OrderBLL and ProductBLL, as well as some counters for the generation of the PDFs. There are methods for creating an order, a report, inserting clients, products and for deleting a client or a product. An important method is for parsing the input file, since this contains the data used in the application. The implementation is below, its functioning being pretty straightforward.



To be mentioned that the formats of the instructions (commands) are the same as in the input file given in the assignment’s description.

* **START**

The start package contains only one class, Start, responsible only for running the application. In its main method, it creates an instance of the WarehouseController class. Also, for the input file, it gets the first argument given when running the application.

# 5.Usage and Testing

Testing was done on multiple sets of inputs, starting with the one given in the assignment’s description. There were also considered inputs where the stock was too small, ordered product/client doesn’t exist, ordered amount is not a valid number, to be sure that everything is working as expected.

# 6.Conclusions

During the development of this assignment, some research was needed for understanding the concepts of layered architecture, Singleton pattern, connecting a MySQL database to an application, creation of a generic class and Reflection technique

Future developments:

* Extending the database with both records and tables, to grow the functionality of the application
* Graphical user interface to be easier to work with the application for both admin and users
* More information on the clients, addition of email address, phone number, address with zip code, county, etc.
* More information for a product (expiration date, manufacturing company, country of origin and so on)
* Accounting functionalities for the warehouse, calculation of the incomes, VAT and so on
* As an ending note, on this documentation I didn’t focus so much on explaining every single method from the classes since there is the javaDoc as a support, instead I tried to discuss the most important aspects from the implementation and the learned programming methods.

# 7.Bibliography

* <https://mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>
* <https://www.baeldung.com/javadoc>
* <http://tutorials.jenkov.com/java-reflection/index.html>
* <https://dzone.com/articles/layers-standard-enterprise>
* <https://www.baeldung.com/java-pdf-creation>
* <https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>