Technical University of Cluj-Napoca

Fundamental Programming Techniques

**ORDER MANAGEMENT**

ASSIGNMENT 3

Tilea Anda Corina

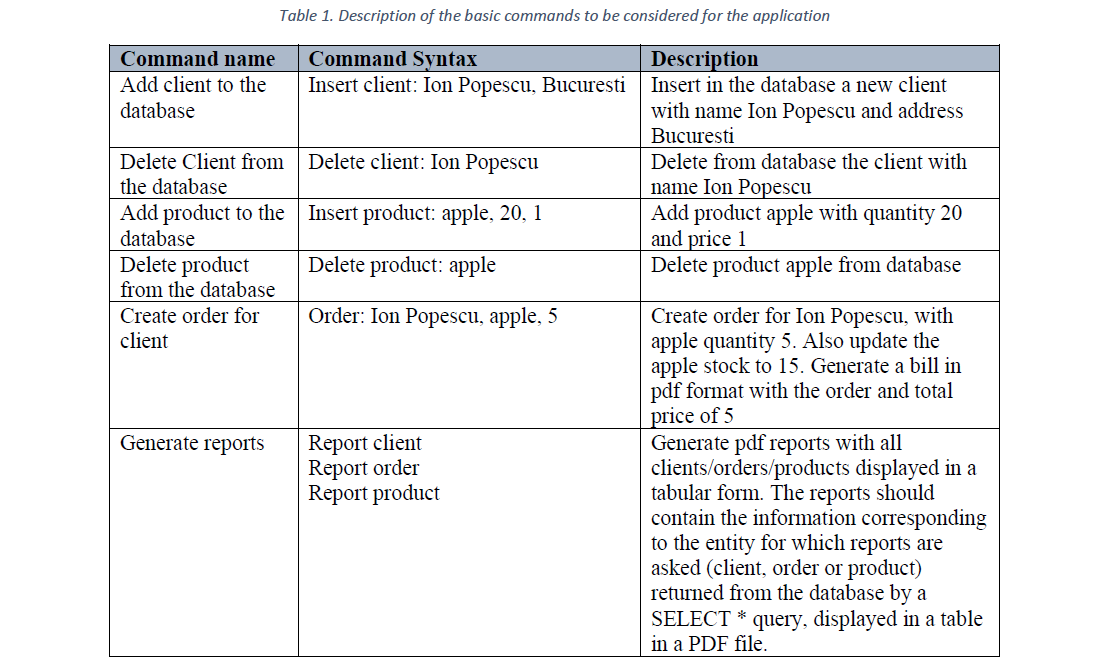
Group 30421

**Contents:**

|  |  |
| --- | --- |
| 1.Problem’s Objective | pg3 |
| 2.Problem’s Analysis | pg3-4 |
| 2.1 Use Case | pg5 |
| 2.2 Scenarios | pg5-6 |
| 3. Design | pg6-7 |
| 3.1 UML Diagram | pg7 |
| 4. Classes and Packages | pg8 |
| 4.1 Implementation | pg8-16 |
| 5. Testing | pg17-18 |
| 6. Further Development | pg19 |
| 7. Conclusions | pg19 |
| 8. Bibliography | pg19 |

1. **Problem’s Objective:**

Consider an application **Order Management** for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders. The application should allow processing commands from a text file given as argument, perform the requested operations, save the data in the database, and generate reports in pdf format. Implement a parser to read commands in the Presentation layer (instead of the standard graphical user interface), and a pdf file generator to generate reports.



1. **Problem’s Analysis:**

MySQL is the most popular database management system. A database is a structured collection of data. It may be anything from a picture gallery to, in our case, processing customer orders for a warehouse. In order to add, access and process data stored in a computer database, a database management is needed. So, the official definition for a database is: a structured set of data, with other words, a collection of connected tables, empty or containing data, on which certain operations can be done.

In this application, we will create three tables (Client, Product, Order), which are made of their headers and the data they contain after the execution of the requested commands.

Talking about the operation the application is going to use, we have:

* **Insert Client/Product/Order/Delivery** – this application will access the database and will create a new row in the specific table it belongs, containing the information given as parameters from the command.
* **Delete Client/Product** – this application will access the database and will delete a row from the specific table. The deletion will be done using a name String the command specifies.
* **Update Product** – this application will access the database and will update a specific row from the Product My SQL Table. The changed value represents the Product’s quantity, which will decrease immediately after an Order is accepted and processed.
* **Find Client** – this application will access the database and will search for a specific row from the Client My SQL Table. The Client is searched by its name. If it is found in the Table, then that Client will be able to place an Order. If the searched name does not belong in the Client Table, then the Order will not be accepted, nor processed. A PDF notice will be generated in which the user is asked to register first.

**As a short Approach:**

* Firstly, we suppose that the javadbconnection database is already created. The creation of the SQL tables can be checked accessing the dumpEmpty file. The tables are initially empty and will be filled after the input commands are processed. The added values can be seen in the dumpAfterCommands file.



* The next thing which must be done is the connection between the MySQL database and the Eclipse Java Project. It has to be established. So, the current workspace will contain a .jar file, called Connector which will assure this connection, along with the classes implemented in the “connection” package.



* After the connection is established, then the next task will be to create a Layered Architecture. The project will contain five packages, named suggestively. These layers will maintain an order in the project and gives each class a very specific role. These can be seen in detail, in the Design topic of the Documentation.
* Each ‘essential’ table from the SQL database need to correspond to a class from the Java project, which means they need to have the same attributes, with the same type. So, the three “table classes” will be represented by the Model Classes: Client, Product and Order.
* The application will develop a Class in the Presentation package, called Controller, in which the input text will be loaded. Then, the information will be computed and processed using the logic in the business Layer. The same package will also be responsible for the PDF generators requested by the problem.

2.1 User Case:

In this application, the User will not participate directly. Since the graphical user interface is not required, the User won’t be able to physically interact with the Simulation. The alternative is the following: The User will type in an input text file, all the commands needed (Insert Client, Delete Client, Insert Product, Delete Product, Order and Reports) by the format presented in the Problem’s Objective. The application will take care of the rest and in the case where the commands are correctly inserted, the PDF documents will be generated and the databases will be uploaded.

2.2 Scenarios:

1. Use Case: <Simulation>

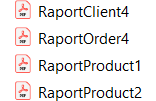
Primary Actor: User

Main success scenario:

* The application is launched successfully by running the command for the .jar file created;
* The argument typed in the Command Prompt represents the correct name of the input text file with the extension “.txt”. This file must contain the commands to be performed. Its name will be shown as an output in the Command Prompt before it executes the commands.



* All the commands are structured after the model described in the Problem’s Objective.
* The application will start immediately, and the results will be visible by opening the PDF reports generated in the Project folder. Also, some suggestive messages will be shown in the command prompt, only to help the User see the evolution of the program.
* The Reports are numbered in ascending order, for each procedure, helping the User track the steps performed.
* The SQL database will be updated with the specific records.
* The same registered client can place as many Orders as he/she wants, but the bills will be generated individually for an easiest way to check the amount to pay for each product.
* In the case when the Order is computed then the Delivery details will appear in the SQL table.

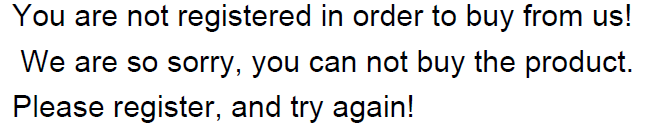


1. Alternative Sequences:

* If in the input file, an already registered Client wants to register, an error will occur and it will be shown in the Command Prompt. Since the name of the Client represents the primary key for the Client table, the error displayed will be “DUPLICATE PRIMARY KEY”. Since that Client is already registered (existing in the table), he/she can already make an Order;



* If in the input file, an unknown Client wants to make an Order, a message will be displayed in a specific PDF Report. The message will be telling the Client to register first, then order a product. In this case, the Order won’t be processed and the bill isn’t generated.



* If in the input file, an already existing Product is added, an error will occur and it will be shown in the Command Prompt. Since the name of the Product represents the primary key for the Product table, the error displayed will be “DUPLICATE PRIMARY KEY”. Since that Product already exists, it can be ordered as long as the stock allows it;



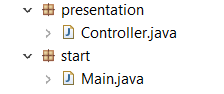
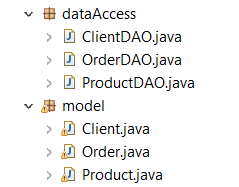
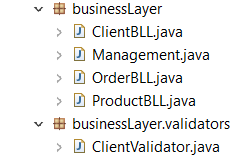
* If in the input file, the Product ordered isn’t added and it does not exist in the database either, a ‘null’ Java error will occur, it will be shown in the Command Prompt and the Order won’t be taken into consideration;

1. **Design:**

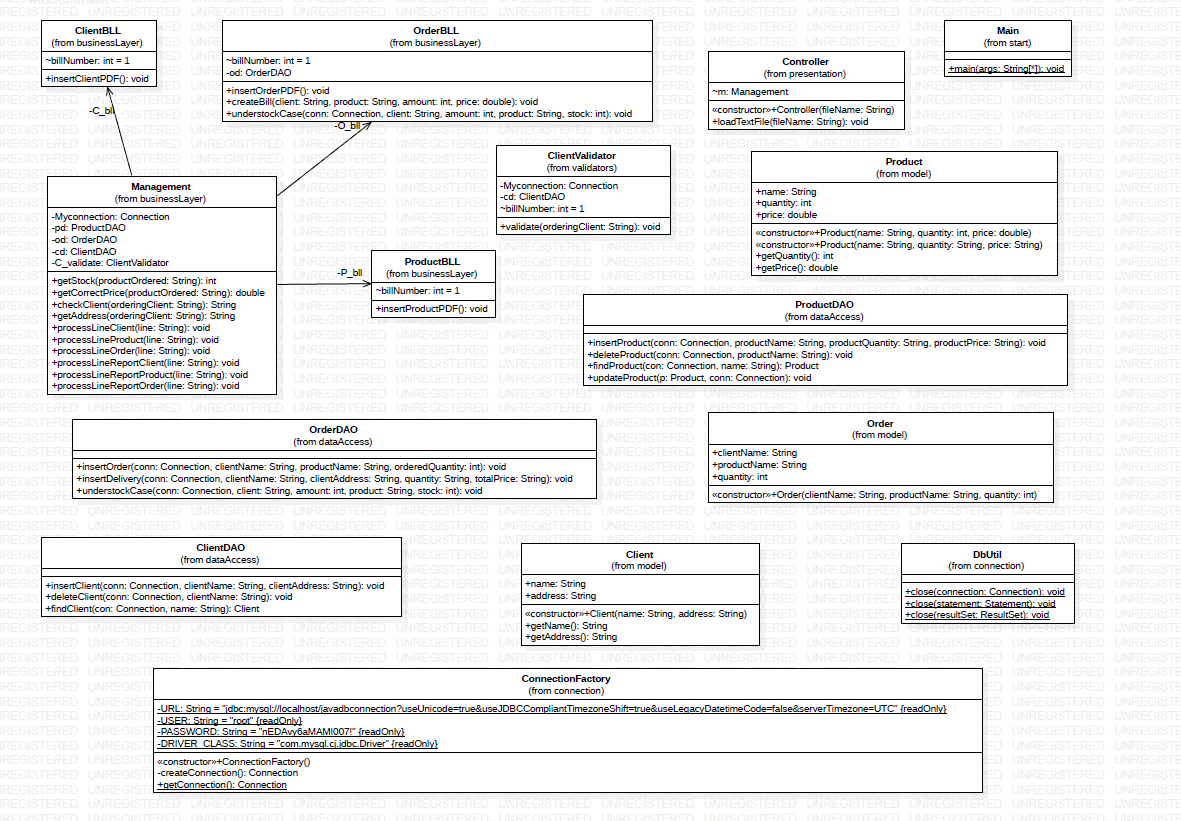
This Project will present two major parts. Firstly, the logic of the Application will be mentioned. The second part will be represented by the ‘presentation’ Layer, which represents loading the input data from the created file. In order for the Application to run correctly and perform all the commands given, both parts must be implemented.

The Application is structured in packages, using a layered architecture. Each layer has a special purpose. The ‘presentation’ Layer: contains the Controller Class which implements the input of the Application. The ‘business’ Layer: contains the Management Class, ClientBLL Class, ProductBLL Class and OrderBLL Class, they will implement the application’s logic. The ‘model’ Layer: contains the Client, Product and Order classes which are mapped to the database tables. The ‘dataAccess’ Layer: contains the ClientDAO, ProductDAO and OrderDAO classes, which contain the queries. They implement the access to the database in order to modify the records in the existing SQL tables. The ‘connection’ Layer: contains the ConnectionFactory and DbUtil Classes which create the connection to the database, close the executed statements, close the Result Set after performing the queries and also closing the connection.

The last package is represented by the ‘start’ one: which contains the Main class, the class which will read the file-name argument and run the entire Application.



* 1. UML Diagram:



1. **Classes and Packages:**

This application is created using a Layered Architecture. Therefore, the project has six packages:

* businessLayer (along with the Client Validator sub-folder);
* connection;
* model;
* dataAccess;
* presentation;
* start;

A Layered Architecture increases the maintainability of the project. In this way, the user can easily see the separation between the loading of the inputs and the business logic of the Application, between the connection area and the data access logic. Also, using this technique, it is allowed to swap and reuse components at the user’s will.

4.1 Implementation:

This chapter will provide a closer look at the role and the implementation of each class, with its attributes, constructors and methods. Some of the details can be also found in the JAVADOC files generated in the ‘doc’ folder. The application will contain fifteen classes, each one belonging to a specific package.

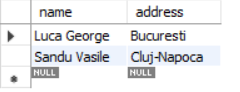
* **Client Class (Model Package):** it resembles the Client table from the SQL database. The attributes of the Client Class represent the header of the corresponding table;

This class will contain two Fields:

* name: a String data type variable, which will represent the name of the Client;
* address: a String data type variable, which will represent the address of the Client;

This class will contain one Constructor:

* Client (String name, String address): it creates a new object, using the parameters given. Each new client created represents a new row. The Client Table presents as the primary key: the name of the Client.



This class will contain two Methods:

* getName(): in order to access the name of the Client, this getter was created. It returns a String data type, representing the current name.

This method will be needed when the Application will try to find a Client for checking if he/she is registered.

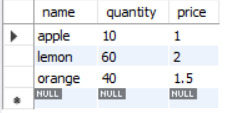
* getAddress(): this method was created in order to access the address of the Client. It returns a String data type, representing the current client’s address. This method will be needed for the ‘Delivery’ information, regarding any processed Order.
* **Product Class (Model Package):** it resembles the Product table from the SQL database. The attributes of the Product Class represent the header of the corresponding table;

This class will contain three Fields:

* name: a String data type variable, which will represent the name of the Product;
* quantity: an Integer data type variable, which will represent the stock of the Product (the number of available existing products);
* price: a Double data type variable, which will represent the price of the Product;

This class will contain two Constructors:

* Product (String name, int quantity, double price): it creates a new object, using the parameters given. Each new product created represents a new. The Product Table presents as the primary key: the name of the Product. This constructor will be used for the Orders, in order to check the Price and the remaining stock;
* Product (String name, String quantity, String price): This constructor will be used later in order to perform the Find function of the Product.



This class will contain two Methods:

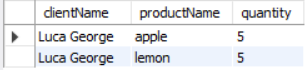
* getQuantity(): It returns an Integer data type. The method will be used in the Management, in order to find the correct total stock for a certain Product which appears in the Order;
* getPrice(): It returns a Double data type. The method will be used in the Management class, in order to find the correct Price for the Product being ordered. Also the total amount which needs to be paid depends on this method.
* **Order Class(Model Package):** it resembles the Order table from the SQL database. The attributes of the Order Class represent the header of the corresponding table;

This class will contain three Fields:

* clientName: a String data type variable, which will represent the name of the Client making the Order;
* productName: a String data type variable, which will represent the name of the Product ordered;
* quantity: an Integer data type variable, which will represent the quantity wanted;

This class will contain one Constructor:

* Order (String clientName, String productName, int quantity): it creates a new object, using the parameters given. Each new Order created represents a new row. The Order table does not have a primary key, because the policy of the Warehouse is that, each Client can place as many orders as they want, with the condition that: the product is still in stock, and each bill will be payed individually.



* **ClientDAO(dataAccess Package**): it performs all the operations needed for a Client. Using the Connection with the SQL database, some queries are created in order to insert/delete and find a certain Client;

This class will contain three Methods:

* insertClient (Connection conn, String clientName, String clientAddress): the insertion method has as parameters: the connection the Application works with and two Strings which contain the information wanted to insert. We will insert these values in the Client table. In order to do that, a SQL String statement is created. It covers the insertion of the wanted values, updating the database each time an insertion occurs.
* deleteClient (Connection conn, String clientName): the deletion method has as parameters the Connection we are working with and the name after which the application deletes the record. In order to do that, a String statement is created. This SQL query is executed in order to delete the desired row, updating the database each time a deletion occurs.
* findClient (Connection con, String name): the search method takes as parameter the Connection and the name of the Client we want to find. In order to find the wanted Client a SQL query is executed. This method returns the found client with all its data (the correct name and the corresponding address). The usage of this method can be seen when the Client is validated. The application (helped by this function) will always check if the Client which made an Order is registered or not. Depending on the answer, the corresponding PDF file will be generated.

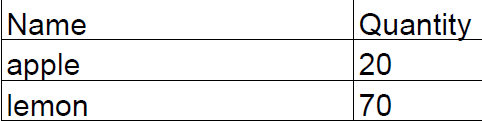


* **ProductDAO(dataAccess Package**): it performs all the operations needed for a Product. Using the Connection with the SQL database, some queries are created in order to insert/delete/update and find a certain Product;

This class will contain four Methods:

* insertProduct (Connection conn, String productName, String productQuantity, String productPrice): We will insert these values in the Product table. In order to do that, a SQL String statement is created. It covers the insertion of the wanted values, updating the database each time an insertion occurs.
* deleteProduct (Connection conn, String productName): the deletion method has as parameters the Connection we are working with and the name after which the application deletes the record. In order to do that, a String statement is created. This SQL query is executed in order to delete the desired row, updating the database each time a deletion occurs.
* findProduct (Connection con, String name): the search method takes as parameters the Connection and the name of the Product we want to find. In order to find the wanted Product a SQL query is executed. This method returns the found product with all its data (the correct name and the corresponding quantity and price). The usage of this method can be seen when an Order is done. The application (helped by this function) will always get the correct price for each Product and also the remaining stock, needed in order to successfully complete the Order.
* updateProduct (Product p, Connection conn): The Update function takes as parameters: a Product Object and the Connection we are working with. Getting the fields from the Product Table, the Application will be able to execute another SQL statement in order to update the database. The method is used for permanently updating the Product stock, after each Order.

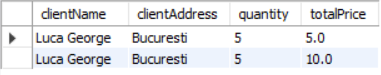
Before Orders: After Orders:



* **OrderDAO(dataAccess Package**): it performs all the operations needed for an Order. Using the Connection with the SQL database, a query is created in order to insert a certain Order;

This class will contain three Methods:

* insertOrder (Connection conn, String clientName, String productName, String orderedQuantity): the insertion method has as parameters: the connection the Application works with and three Strings which contain the information wanted to insert. We will insert these values in the Order table. In order to do that, a SQL String statement is created. The database is updated each time an Order is accepted and performed. To mention is that, the Order table does not have a primary key. The SQL table is created this way because each Client should be able to place as many orders as he/she wants, with the same/or different products, as long as the stock allows it.
* insertDelivery(Connection conn, String clientName, String clientAddress, String quantity, String price): this is one of the two ‘Extra Methods’ added in order to re-create a Database Structure. It takes the needed data and add it in the table ‘delivery’ in order for a delivery to be executed. The method is implemented here because it requires operations done on the SQL Database, but it will be really used in the Business Logic Layer.



* understockCase(Connection conn, String client, int amount, String product, int stock): this is the second ‘Extra Method’ added in order to re-create a Database Structure. It takes all the needed data and added in the table ‘understock\_products’ specially created for understocked Products. A much more organized Warehouse presents a special table from which we can easily be notified which Products to restock.



* **ConnectionFactory (connection Package):** this class makes it possible to connect the Java Eclipse Workspace to the MySQL database created. Here, all the information regarding the connection to the database server will be declared, such as: the name of the database, the username and the password for the host. The URL String used is a particular one due to the difference of Time Zones, an error which sometimes occurs in the case of multiple connected devices.

This class will contain one Constructor:

* ConnectionFactory(): the constructor facilitates the Connection by using the downloaded Connector;

This class will contain two Methods:

* createConnection(): this method is used in order to actually create the Connection with the database server. Based on this Connection, all the other operations of this Project will be performed.
* getConnection(): The attributes were declared private so, in order to access the Connection, this getter was created. It returns the just created Connection used from now on in order to execute the commands on the database.
* **DbUtil (connection Package):**  this class will contain all the other functions, which are used in order to close the Connection obtained. Also, the executed Statement and the Result Set obtained after the execution of the query will be closed, due to this class.

This class will contain three Methods:

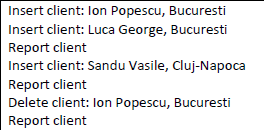
* close(Connection connection): This method checks if the Connection is established. If the Application presents an active Connection, then this function will be able to close it.
* close(Statement statement): This method will help the Application close the executed Statement;
* close(ResultSet resultSet): This method will help the Application close the obtained Result Set, only after the execution of the required queries.
* **Controller (presentation Package):** this Class was created in order to implement the User input. The data will be extracted from a text file using a Parser. All the data will be computed by the businessLayer of the Application. The PDF generators (the output resulted) were chosen to be generated in the logic Layer, each one belonging to the specific Class: ClientBLL, ProductBLL, OrderBLL. They represent particular cases of the Line Processing.

This class will contain one Constructor:

* Controller (String fileName): the Constructor of this class will call for the loadTextFile method. The suggestive name tells that, in this way all the data contained in the text file will be loaded.

This class will contain one Method:

* loadTextFile (String fileName): The commands presented in the text file will be loaded and computed line by line in the business logic of the Application. The commands should be structured as in the Problem’s Objective in order to be parsed and processed correctly.



* **ClientBLL (businessLayer Package)**: this Class will contain the Response on all the operations done on Clients. By using the PDF generator implemented here, all the Client Reports will be produced in the Project Folder. Opening them, the User is able to see the development: some Clients are inserted, some of them are deleted;

This class will contain one Field:

* billNumber: it represents the order number for the generated output files. For each task in particular, an ascending ordering takes place.

This class will contain one Method:

* insertClientPDF(): the method generates the Output PDF file required for Clients;
* **ProductBLL (businessLayer Package):** this Class will contain the Response on all the operations done on Products. By using the PDF generator implemented here, all the Product Reports will be produced. By opening them, the evolution of Products can be seen: some Products are inserted, some of them are deleted, and the stock for the ordered Products will decrease.

This class will contain one Field:

* billNumber: it represents the order number for the generated output files. For each task in particular, an ascending ordering takes place.

This class will contain one Method:

* insertProductPDF(): the method generates the Output PDF file required for Products;
* **OrderBLL (businessLayer Package)**: this Class will contain the Response of the insertion of an Order. By using the PDF generator implemented here, the Order Report will be produced. Also, this Class will be responsible for the Order’s results. If the placed order is accepted, this Class makes sure that a PDF bill is created. If the ordered Product is understock, then this Class makes sure that a PDF warning is created.

This class will contain two Fields:

* billNumber: it represents the order number for the generated output files. For each task in particular, an ascending ordering takes place.
* od: it is calling the operations done on Orders. This variable is needed in order to add all the understock requests in one place. When an Order cannot be processed because the stock is smaller than the ordered amount, the Product will be added in a separate table for a better notice and restocking.

This class will contain three Methods:

* insertOrderPDF(): the method generates the Output PDF file required for Orders;
* createBill(String client, String product, int amount, double price): the method generates the Output PDF file required in the case when the Order is accepted;
* understockCase(Connection conn, String client, int amount, String product, int stock): it uses the SQL insertion for adding the Product in cause in the understock table, in order to be noticed easier by the employees. The method also generates the Output PDF file required in the case when the Order is rejected due to a stock situation.
* **Management (businessLayer Package):** this Class represents the most complex one, and it shows the entire Logic of the Application. Due to the implemented methods, the Application will know exactly what to do, for each line of Commands, as long as the User types the correct format for that specific command.

This class will contain seven Fields:

* Myconnection: it represents the SQL Connection we are working with, in order to insert/delete/find/update the corresponding input records in the specific tables from our database.
* pd: ProductDAO: it represents a call to the Product Class which implements all the needed operations on Products (insertion, deletion, update and find) and sync them in the database. Having a Layered Architecture helps the User in arranging the implementation as organized as possible.
* od: OrderDAO: it represents a call to the Order Class which implements the needed operation for the Order (insertion) and sync it in the corresponding table from the database.
* cd: ClientDAO: it represents a call to the Client Class which implements all the needed operations on Clients (insertion, deletion and find) and also sync them into the “client” table from our SQL database.
* C\_bll: it represents a call to the ClientBLL Class, responsible for the Client’s part of the PDF generator.
* P\_bll: it represents a call to the ProductBLL Class, responsible for the Product’s part of the PDF generator.
* O\_bll: it represents a call to the OrderBLL Class, responsible for the Order’s part of the PDF generator and also the insertion into the Understock SQL table.

This class will contain ten Methods:

* getStock (String productOrdered): This method is used in order to find a specific Product and return its remaining quantity. The Client will order a certain Product. The Application must take that Product name, search it in the database, find its details and return the available, existing stock, in order to decide if the Order can be accepted or not.
* getCorrectPrice (String productOrdered): This method is used in order to find a specific Product and return its correct price. The Client will order a certain Product. The Application must take that Product name, search it in the database, find it details and return the attached Price, in order to calculate the Total Amount which has to be paid by the customer.
* checkClient (String orderingClient) : This method is used in order to find a specific Client. A client name will always appear on the Order command.

The Application must take that Client name, search it in the database, find its details and return one of the two cases: if the Client Name exists in the database, the Order processing will continue, in the correct way, generating the specific PDF file. If the Client Name does not exist in the database, then a PDF warning will be generated. If the Client is not registered (added in the table) so he/she cannot perform any Orders.

* getAddress(String orderingClient): This method is used in order to find a specific Client and return it’s attached Address. This getter helps the insertion of the data into the Delivery SQL table. For each order, the Warehouse provides a Delivery option.
* processLineClient/Product/Order (String line): These methods treat each one of the requested operations regarding Clients/Products/Orders. They check which command is presented on each line by having a specific structure (the format given in the Problem’s Objective).
* processLineReportClient/Product/Order(String line): These methods treat each one of the requested output files. They check which command is presented and generate the corresponding PDF Reports in the Project Folder.
* **ClientValidator(businessLayer.validators Package)**: this Class is specially created for the case in which the validation of a Client fails.

This class will contain three Fields:

* Myconnection: this variable represents the connection with the SQL database. It is used in order to apply the Client operations implemented in the data Access Layer.
* cd: the variable represents a call to the ClientDAO Class. From this Class we will use the ‘find Client’ method, in order to check the registration.
* billNumber: it represents the order number for the generated output file. For each task in particular, (in our case: failed registration), an ascending ordering takes place.

This class will contain one Method:

* validate(String orderingClient): this method helps the Logic of the Application dealing with this particular case. In the case in which the Client is not validated, a PDF Report is created to notify the Client he/she needs to register first.
* **Main(start Package):** this Class represents the main class of the Application which will run the entire Project.

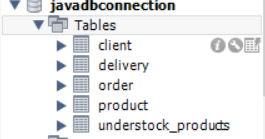
This class will contain one Method:

* main(String[] args): this method will present as an argument the name of the input file, containing the Commands. An output command message will help the User check if the correct file is being loaded. If the file is successfully found, then the Application will start extracting data from it and then compute it as required.

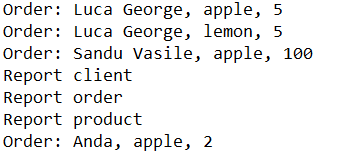


1. **Testing**

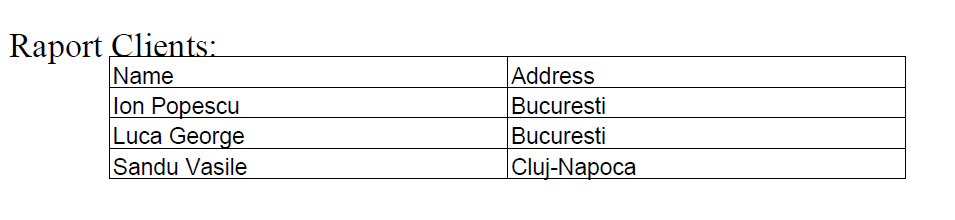
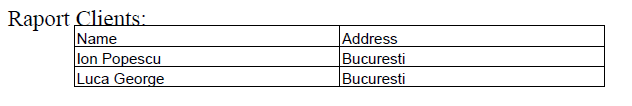
Firstly, the already created SQL database, called in our case javadbconnection, is initially empty for all tables.

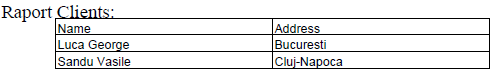


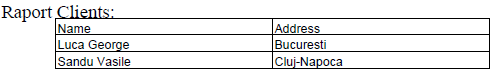
The next step is for the User to write the Commands input text file. The data tested was the one provided in the Request of the Assignment. At the last line, another order was placed, in order to test the case when the Client is not registered and how the Application deals with the situation.

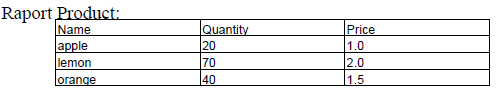


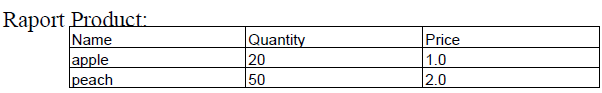
The following step will be to open the Command Prompt and run the .jar file created for this Project. The Application will generate PDF reports for each of the cases encountered with the bill Number in ascending order (for each one of the cases: ReportClient, ReportProduct, bill/understock/ReportOrder, clientNotRegistered).

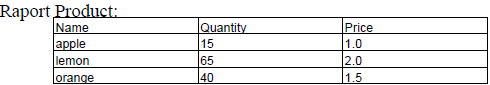
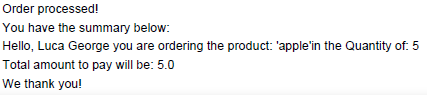




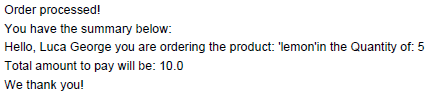




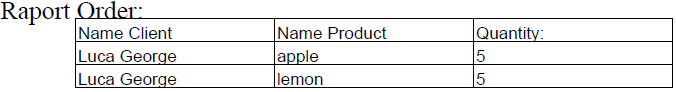


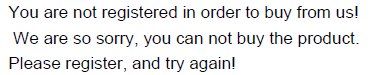
 



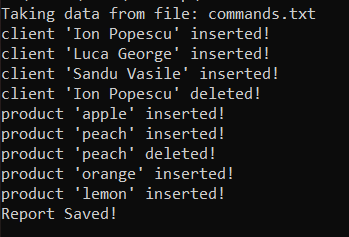
 understock mesaj







The application is now generated. If the application will be run again, without erasing the data from the SQL tables, the Clients, Products with the same name cannot be added again, and the stock will be considered the one from the end of the first running. Regarding the new ‘Extra SQL Tables’ the Delivery Strings can repeat themselves as many times as a Client places an Order or a Product remains understocked. The Application will be left empty, ready to run for the first time. These PDF examples can be found in the PDF Folder. The javadoc files generated, containing extra-information about the Classes and their methods are placed in the doc folder.



1. **Further Development**

The Project idea is a very vast and complex one. The requirements can go into further development by creating an entire Restaurant, with a vast menu. The food may be organized by its origins and the special times of the day in which they must be served. On the same principle, data can be computed for an Internet Banking Application. For this idea, some instructions may be generated and delivered to the clients on more than one devices (both computer and phone – mail and sms). Clients can Register as many times as they want and another SQL will keep track. An Interface may be used in order for the Client to have “direct access” to his/her account and see all the details.

1. **Conclusions**

This Application was a very good way of finding more about Order Management and how it works. I can say that I liked working with a SQL database the most, until now. It is fascinating how the Connection between the Java Workspace and the SQL database can be done.

Also, this was the first real interaction with a Layer Architecture, and I can say that it is a method much more attractive for a User. The specific packages serve a different purpose and it becomes easier to understand the concept and the Logic of an Application.

One of the best things about this Project, it was that it could be interpreted in many other ways, leaving the User the freedom to choose how he/she wants the customer orders for the Warehouse to be processed;

1. **Bibliography**

<https://ibytecode.com/blog/jdbc-mysql-create-database-example/>

<https://www.baeldung.com/javadoc>

<http://www.coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_3/Assignment_3_Indications.pdf>

<http://statweb.stanford.edu/~naras/webdocs/java/tools/javadoc.html#doclet>

<https://www.dummies.com/programming/java/how-to-use-javadoc-to-document-your-classes/>

<https://dev.mysql.com/downloads/connector/j/5.1.html>

<https://stackoverflow.com/questions/15118163/itextpdf-library-cant-find-the-class-com-itextpdf-text-pdf-pdfawriter>

<https://alvinalexander.com/java/java-mysql-update-query-example/>