

Order Management

1. Purpose of the laboratory work	2
2. Analysis, modelling and use-case scenarios	2
3. Development	4
3.1 Package Diagram	4
3.2 Class Diagram	5
3.3 Algorithms, Data Structures, External Packages	6
4. Implementation	6
4.1 Start Package	6
4.1.1 Main Class	6
4.2 presentation Package	6
4.2.1 FileManipulator Class	6
4.3 model Package	6
4.4 dataAccessLayer Package	6
4.4.1 AbstractDAO Class	6
4.4.2 ClientDAO Class	7
4.4.3 OrderDAO Class	7
4.4.4 OrderItemDAO Class	7
4.4.5 ProductDAO Class	7
4.4.6 ConnectionFactory Class	7
4.5 businessLayer Package	8
4.5.1 MainBLL Class	8
4.5.2 ClientBLL and ClientValidator Classes	8
4.5.3 ProductBLL and ProductValidator Classes	8
4.5.4 OrderBLL Class	8
5. Results	9
6. Conclusions	9
7. Bibliography	10

1. Purpose of the laboratory work

For this requirement we are required to build an application that handles operations on a MySQL database through a set of commands given as an input file. The database should store clients and their information, products and their respective quantity and price and finally, the orders placed by the clients.

The development process consisted of the following steps:

- Building the empty database
- Creating the model classes that resemble the tables of our database
- Creating universal queries through reflection and specific queries for our instructions
- Parsing the input file
- Handling the logic that manipulates the input data through queries
- Managing the output and printing it to PDF files as requested

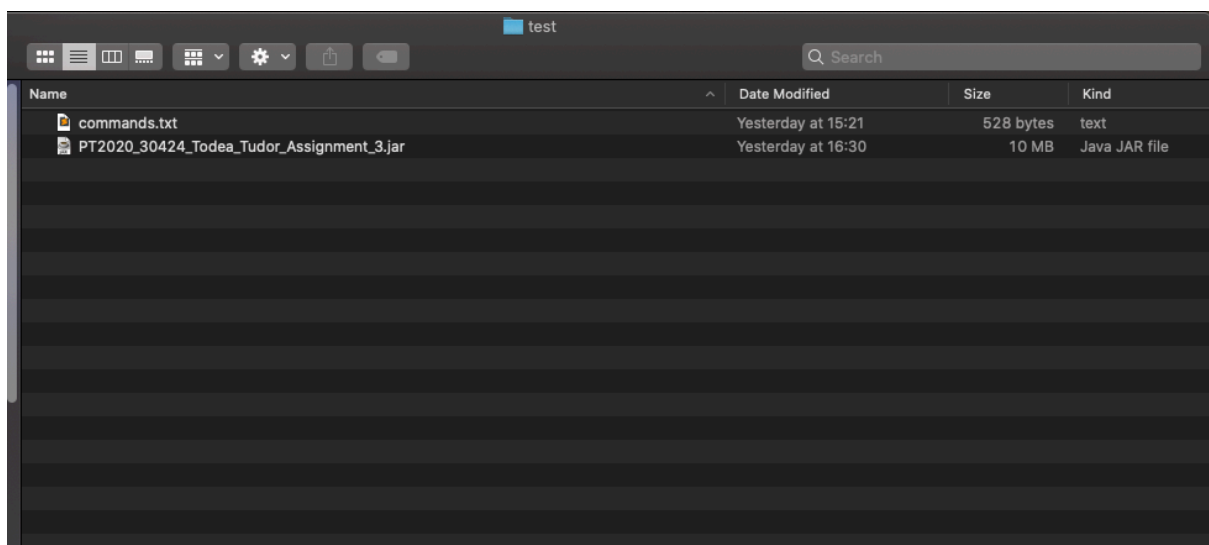
All of these steps will be described in more detail in the 3rd and 4th chapter of this documentation.

2. Analysis, modelling and use-case scenarios

Our application is required to be run from a .jar file inside the terminal. The user needs a database schema with the following tables:

1. Client with the fields: PK: name, city and deleted
2. Product with the fields: PK: name, quantity, price and deleted
3. Order with the fields: PK: id, name (foreign key from Client), cost and deleted
4. OrderItem with the fields: order_id (foreign key from Order), name (foreign key from Product) and quantity

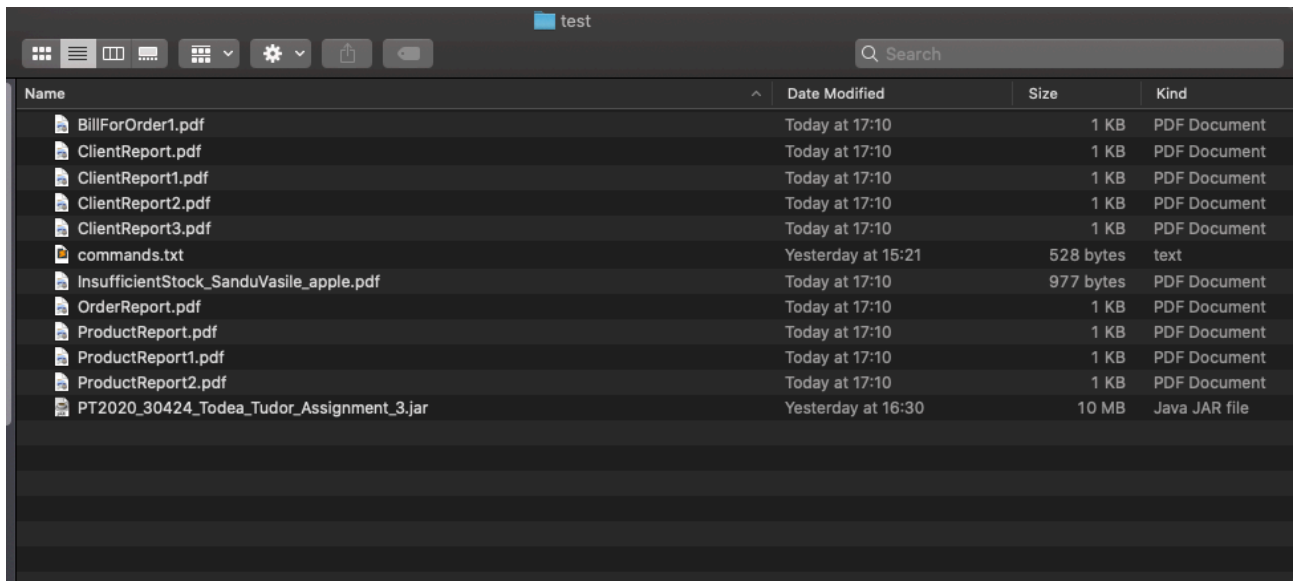
Alongside the database and the .jar project we also need a text file with valid commands that our program needs to execute in order to manage our database. This text file is provided as a command line argument. We have an example below:



We have the folder above containing the .jar file and the commands.txt file. We open a terminal instance and change the directory to our current one and execute the command:

```
tudort@192: ~/Desktop/test
tudort@192 ~$ cd Desktop/test
tudort@192 ~/Desktop/test$ java -jar PT2020_30424_Todea_Tudor_Assignment_3.jar commands.txt
tudort@192 ~/Desktop/test$
```

After running the .jar we will have our database populated with the necessary information as provided through the commands and with the following PDF files inside the folder of our project:



Name	Date Modified	Size	Kind
BillForOrder1.pdf	Today at 17:10	1 KB	PDF Document
ClientReport.pdf	Today at 17:10	1 KB	PDF Document
ClientReport1.pdf	Today at 17:10	1 KB	PDF Document
ClientReport2.pdf	Today at 17:10	1 KB	PDF Document
ClientReport3.pdf	Today at 17:10	1 KB	PDF Document
commands.txt	Yesterday at 15:21	528 bytes	text
InsufficientStock_SanduVasile_apple.pdf	Today at 17:10	977 bytes	PDF Document
OrderReport.pdf	Today at 17:10	1 KB	PDF Document
ProductReport.pdf	Today at 17:10	1 KB	PDF Document
ProductReport1.pdf	Today at 17:10	1 KB	PDF Document
ProductReport2.pdf	Today at 17:10	1 KB	PDF Document
PT2020_30424_Todea_Tudor_Assignment_3.jar	Yesterday at 16:30	10 MB	Java JAR file

If we want to repeat this and apply these same commands again, we need to clear the database again. This can be done by applying the following query inside MySQL Workbench:

```
1  SET SQL_SAFE_UPDATES = 0;
2  DELETE FROM order_manager.orderItem;
3  DELETE FROM order_manager.product;
4  DELETE FROM order_manager.order;
5  DELETE FROM order_manager.client;
6  ALTER TABLE order_manager.order AUTO_INCREMENT = 1
```

Use case scenarios:

1. Insertion:

-The syntax for the insertion of a client is: "Insert client: FirstName LastName, City". Inserts the client into the Client table. Clients must have unique names (mentioned in the requirements).

-For the insertion of a product we have: "Insert product: productName, quantity, price". It inserts the product into the Product table. Products are unique. If we insert an already existing product, we update the quantity and the price of the existing one.

2. Deletion:

-For clients: "Delete client: FirstName LastName, City". We set the deleted field for that client to 1. That entry will no longer show in our queries. For practical purposes it no longer exists in our database. We also trigger the delete flag for the orders made by that client in the Order table.

-For products: "Delete product: productName". Does the exact same thing as the client deletion except it does not void the already existing orders containing the product. We consider that the deletion of a client means he is no longer a client and its presence is no longer needed in the database and his order no longer possesses any importance. For the product it merely means that the product is no longer sold, but that does not void previous order that contain the deleted product.

3. Order

-For orders, the syntax is: "Order: FullClientName, productName, quantity". If we have two consecutive lines that request and order from the same client we will process it as a singular order. If the client orders more items than we have in stock, we display a message in a PDF file.

4. Report

-For reports the syntax is: "Report client/order/product" depending on what we want to report. It generates a PDF with the requested report.

3. Development

For this project we had more freedom when it comes to the development. We will now present the major decisions when it comes to our take on the assignment:

1. When it comes to deletion, we decided to use a deleted flag that is set to 1 when we decide to delete an entry. We can delete entries from the Product, Client and Order tables.

2. We chose to have the name of the products as Primary Key in the Product table since we have unique products. When we try to insert an already existing product we just update the stock.

3. We picked the same approach for the Client since we were told the name of the clients are unique. This is not always the case in reality, but since we are guaranteed client name uniqueness, dropping a column is more efficient when it comes to the data. More columns means more possibilities to get tangled in the data, even so if the extra columns are redundant, which is the case here since a product ID and client ID would have bared no use. The name of the client or the product are already unique identifiers for an entry.

4. We designed an additional table called OrderItem. This table would have been redundant if we considered each order command in our file as an individual command. But this is not the case in our approach. If we are faced with multiple consecutive input lines of type order from the same client, we will process it as a single order containing multiple products. This means that there is a one-to-many relationship between the Order and OrderItem tables through the Foreign Key order_id in OrderItem.

For example:

Order: Luca George, apple, 5

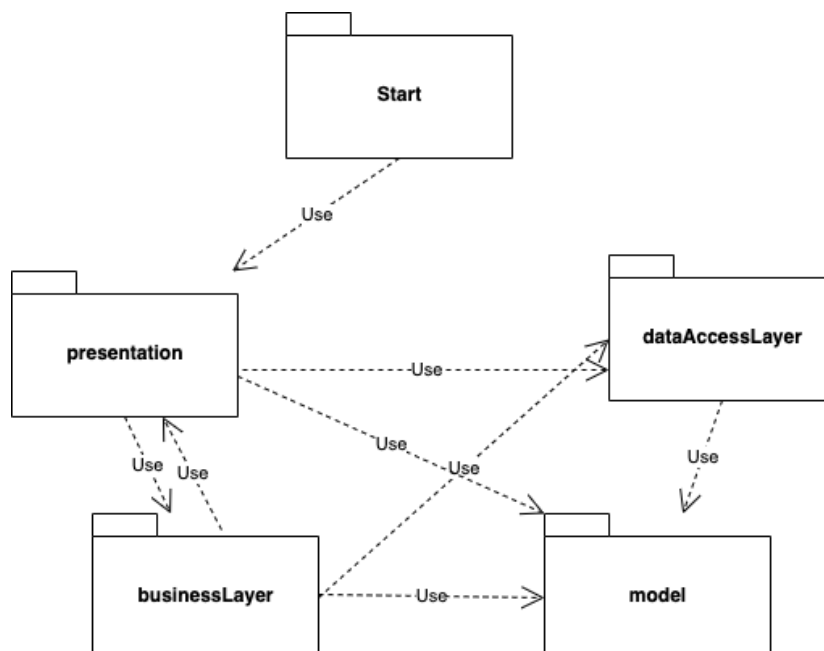
Order: Luca George, lemon, 5

Order: Luca George, orange, 5

These three lines will be considered as a single order. That means the Order table will have one entry for "Luca George" that has as the cost the total price of 5 lemons, oranges and apples. While the OrderItem table will have three entries with the same order_id.

5. Upon the deletion of a client we delete the corresponding orders. However, upon the deletion of a product we do not delete the corresponding order items. The reasoning behind this is mentioned in chapter 2.

3.1 Package Diagram



The class diagram is quite large, and sadly the Pages Document editor can't display only a page in Landscape mode. I will attach the diagram PNG to this project for better readability if needed. The classes and their contents will be described in detail in the next chapter.

3.3 Algorithms. Data Structures. External Packages

As far as data structures go, we only used lists in this project and no special algorithms. We used the JDBC API to connect to the database and the iText and PdfBox libraries in order to handle the PDF files.

4. Implementation

4.1 Start Package

4.1.1 Main Class

The class that contains the main method of our project. It calls a FileManipulator object to handle the file given as a command line argument.

4.2 presentation Package

4.2.1 FileManipulator Class

The constructor of this class receives the file and calls the method **handleFile** to split the file into a list of strings that will be then passed to the MainBLL class to handle. Aside from that, inside our class we also have methods used to create PDF documents necessary for our project.

The private variables *clientReportCount*, *productReportCount* and *orderReportCount* are used to count the number of reports for clients, products and orders have occurred in order to generate PDF reports with different names. The **addClientTableHeader**, **addClientRows** and **reportClients** are the methods that handle the generation of a PDF file containing a table with all the clients in our database and their information. The methods **addProductTableHeader**, **addProductRows** and **reportProducts** do the exact same thing for products.

When it comes to the order we have the **addOrderTableHeader**, **addOrderRows** and **reportOrder**. In the **reportOrder** we have to report all the items bought inside an order, so we use a data access method to get all the OrderItem objects of a certain order. Then we display the total cost of that order on the next row of the table. After that we move onto the next order and so on.

We also have the methods **reportBill** and **reportInsufficientStock**. The first one generates a bill for an order after it has been placed. We only generate the bill after we are sure that the whole order has been processed. This is the case when there are multiple consecutive "Order" commands in the input file. The second method generates a customised PDF informing us that a client wanted to order more items than the amount we have in stock

4.3 model Package

The classes in the model package resemble their respective table in name and in the order, name and type of their attributes. Alongside that there are only accessor methods and an empty constructor since in the data access package we use reflection and we need an empty constructor when creating an empty instance.

4.4 dataAccessLayer Package

4.4.1 AbstractDAO Class

This class contains general database access methods that are built through reflection. These methods can work for multiple tables or for all the tables in our project. These methods deal with the insertion of an entry, the deletion of entries, searching by name for certain entries (such as clients or products), and also finding all the entries in a table which is useful when creating the reports. Since we use reflection, the table used for each case depends on the class that implements AbstractDAO. We will call this class T.

The main methods in this class are:

createSelectQuery(String *field*): creates a generic SELECT query which selects all the attributes of an object from table T with the constraint *field*.

findByName(String *name*): finds an entry in table T with the desired name. Returns the found object if it is present in the table, by calling the method **createObjects** and returning the first object present in the list created by that method, otherwise it returns null.

createObjects(ResultSet *resultSet*): while there are results from a query inside the resultSet we create a new instance of the object T, get its fields and add the contents returned by the resultSet to the fields of the newly created object. We then add this object to a list and repeat the process for the next results of the resultSet. At the end of the method we return the list of objects.

findAll(): finds all the entries in the table T and stores them in a list of objects of the class T with the help of the **createObjects** method.

createInsertQuery(): this method builds an INSERT query for the table T. Adds a '?' in the VALUES section for every field as long as that field is not an auto incremented id, in which case we append a '0'. An example of insertion for the Order table would be: "INSERT INTO order_manager.order VALUES (0,?,?,?)".

insert(*t*): inserts an object *t* into its respective table. We go through every field of the class T and as long as one of them is not an auto incremented id, we check the type of that field then we insert the value of that field in our query. We then execute the query and we should see our entry added to the MySQL Workbench.

delete(*t*): in our application, the deletion is done through an UPDATE command of the deleted field, since we do not actually physically delete an entry from our table, we just no longer take it into consideration. As a consequence, all the SELECT queries also check if the deleted flag is 0. The **delete** operation is quite simple, we just take the name of the entry and SET the deleted flag to 1 for that entry of table T.

4.4.2 ClientDAO Class

This class extends AbstractDAO<Client> making it possible to create objects of type ClientDAO and use the abstract methods build through reflection for objects of type Client to manipulate the Client table. This class has no methods since we did not need to build any Client specific queries.

4.4.3 OrderDAO Class

Just as the ClientDAO, this Class extends AbstractDAO<Order>. As a consequence, we can use abstract queries for Order objects. The same will be available for the other DAO classes. We also have two methods that implement Order specific queries:

selectLastID(): since it is a little harder to implement a way to get the auto generated keys from the **insert** method of AbstractDAO (that is because we only have an auto increment id for the Order table, we already explained why we do not need to use this approach for the other tables) we decided to build a query that gets the MAX(id) from the Order table so that we can get the id of the last inserted entry. This method will be important for the handling of multiple consecutive orders. We will describe this in the OrderBLL class.

updateCost(*id*, *cost*): when we add an item to our order we need to update the total cost of the order. We do this by incrementing the total cost of the order with the price of the newly added item. This total is the *cost* argument which will be set in the Order table at the entry with the corresponding *id*.

4.4.4 OrderItemDAO Class

Aside from the abstract queries, we also have a specific query in the OrderItemDAO class:

findOrder(*id*): finds all the items that belong to the order with the identifier *id* and stores them in a list of OrderItem objects. This is used to print out all the items that belong to an order.

4.4.5 ProductDAO Class

updateProduct(*t*): whenever we have the command to insert an already existing product, we instead update its information. We add the new quantity to our already existing stock and we update the price with the new one. This is what we do through this method, we create an update query to update the price and the stock of a product.

4.4.6 ConnectionFactory Class

This class handles the connection to our database through a singleton instance. It contains the methods to create the connection, as well as methods responsible of closing Statements, ResultSets and finally, the Connection itself.

4.5 businessLayer Package

4.5.1 MainBLL Class

The constructor of this class takes the list of strings parsed from the input file and handles them through the **handleLines** method which we will describe below:

handleLines(line): takes every line and handles it accordingly. We use a matcher to match the string of the provided line with a valid command. When we find out the command of that line we call the necessary method from the other business logic classes. It can be seen that after every call of a command method we also generate a bill. That is done because we consider consecutive order commands from the same clients as belonging to a single order. Whenever we encounter a new command we call the **generateBill** method from the OrderBLL class to generate a bill. If the previous command was an order the bill will be generated, otherwise the method will simply return, generating nothing.

4.5.2 ClientBLL and ClientValidator Classes

The ClientValidator class checks that the client that we are about to insert into the table is a valid one. It does so by splitting the client information into three parts: first name, last name, city. It then checks if these are names through the **isName** method. That means that if the first letter is an uppercase one and the entry contains only letters and the '-' character, the string is a valid name.

Inside the ClientBLL class we have the following methods:

insertClient(line): we check if the input has a valid client and city name. If it does, we create a Client object with the input information and we **insert** it into the Client table using the ClientDAO class.

deleteClient(line): we check that the client we want to delete has a valid name, and if it does, we search for that name in the Client table. If we can find that name, we firstly set the delete flag to 1 for all the orders made by that client using the **delete** method of an OrderDAO object. After that we use the **delete** method to delete the client from the table using a ClientDAO object.

report(): in order to report the clients we use the **findAll()** method to get a list of all the available clients (that have the deleted flag 0) and send that list of clients to the FileManipulator to process it and print it to a PDF file.

4.5.3 ProductBLL and ProductValidator Classes

The ProductValidator checks the validity of the input when we want to add a product to our database. It checks if the name of the product consists of only letters, the quantity is an integer and the price is a double. If all of these conditions are met, we have a valid product that we can process further.

Inside the ProductBLL class these methods are present:

insertProduct(line): we check the validity of the input. If it is valid, we then check if we already have that product inside our Product table. If we do, we call the **updateProduct** method from the ProductDAO class. If we do not, we add the product to the database through the **insertProduct** method of the ProductDAO.

deleteProduct(line): we again check the validity of the input. We then proceed to use the **findByName** method to get the object inside the table and call the **delete** method to set its deleted flag to 1. If the product is not present in the table we will display an error message.

report(): we use the **findAll** method to get a list of all the available products in our Product table. We then use the FileManipulator class to generate a PDF with the provided list of products.

4.5.4 OrderBLL Class

Due to our implementation, the Order class is more complex than the other business logic classes. We have already mentioned the way we deal with the orders and the consecutive orders coming from the same client. It is important that we mention the *previousOrder* field that keeps track of the last consecutive order placed. We use this Order object to check if the order we handle comes from a "new" client and should be handled as a new order, or it comes from the "same" client, meaning that we have the same order and we only need to add items to that order and update its cost.

The methods of this class are:

handleOrderFromNewClient(*auxOrder*, *auxOrderItem*): if the *previousOrder* attribute indicates that the current order comes from a different client, we handle this as a new order. The *auxOrder* argument represents the order that is to be processed and the *auxOrderItem* represents the item that this order contains. We use the **setCost** method to set the price of the item * the quantity of the item as the cost of our order. We **insert** the order in the table, and get the id of the insertion. We use this to set the id of the updated *previousOrder* field and to set the *order_id* of the *auxOrderItem* that we are going to **insert** to the OrderItem table. The *auxOrder* becomes the new *previousOrder* field.

handleOrderFromTheSameClient(*auxOrderItem*): if the *previousOrder* and the new order have the same client name we just set the id of the *previousOrder* as the *order_id* of the *auxOrderItem* and we then insert it into the OrderItem table. We then update the cost of the order using the **updateCost** method.

generateBill(*order*): If the *previousOrder* is not null, we generate a bill for the order with the id of the *previousOrder*. After the generation of a bill with the help of the FileManipulator, we set the *previousOrder* to null. This method is important because after every single command of our file (except for the "Order:" commands, which are handled in **handleOrder**) we are not sure if an order has been finished. We'll take the following example:

Order: Luca George, apple, 5
Order: Luca George, lemon, 5
Report client

We process the first two commands as a singular order. When we reach the third command, we need to check if an order has been finished. So after we read the line "Report client" we know that this is no longer an order and we need to check if we have stopped an order. This is done by calling the **generateBill** method. If we had a *previousOrder* we will generate its bill, if not, the method does nothing since the *previousOrder* would be null.

handleOrder(*line*): this method checks if the quantity that we want to order is an integer, and also if the client and the product are also present in our database. If the input is valid we then check if the client wants to order a larger quantity than what we have in stock. If he does, we generate a bill for the *previousOrder* if it exists since we are sure that the new command is not a consecutive order from the same client. If a *previousOrder* exists a bill will be generated using **generateBill**. Then we create a PDF with a warning that the order can not be fulfilled since the client ordered more than we can handle. If the ordered quantity is valid we check if we have a continuous order such as:

Order: Luca George, apple, 5
Order: Luca George, lemon, 5

We consider that we are processing the second line. This means that this command belongs to the same order as the previous one and we process it as such by calling **handleOrderFromTheSameClient**. If we have something like this:

Order: Luca George, apple, 5
Order: Luca George, lemon, 5
Order: Ion Ionescu, orange, 7
Insert product: mango, 10, 2.5

If we are processing the third line we can see that that this new order does not belong to the *previousOrder* which was made by "Luca George". We then generate a bill using **generateBill** for the order made by "Luca George" containing 5 apples and 5 lemons. We then process the order by "Ion Ionescu" which becomes the new *previousOrder*. At this point we do not know if "Ion Ionescu" plans on adding more products to his order. Only when we reach the 4th command we know that his order is complete and we generate a bill using **generateBill** which is called from the method **handleLines** in the MainBLL class.

report(): finds all the orders that have the deleted flag 0 using the **findAll** method. These orders are then printed to a PDF by the methods inside the FileManipulator class.

5. Results

The only way to see if the application does what it is required to do is to check the reports generated by it and compare it with the expected result. The reports can be found in the project's (jar's) folder.

6. Conclusions

Throughout the development of this application we have learned how to establish the connection and manipulate a database through PreparedStatements and to get the results from a ResultSet. Alongside that, we have learned how to generate basic PDF documents using tables, paragraphs and chunks through the iText and PdfBox libraries. When it comes to further development, if we consider that the clients can have the same name we may need to add a unique identifier for the Client table.

7. Bibliography

Lab presentation - http://www.coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_3/Assignment_3_Indications.pdf

Reflection example: https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-reflection-example.git

Creating PDF files in Java o <https://www.baeldung.com/java-pdf-creation>

Connect to MySql from a Java application o <http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>

MySQL dump from the MacOS terminal: <https://stackoverflow.com/questions/38675885/how-to-make-mysql-dump-file-from-terminal-in-mac>