1. Description of the assignment

We need to consider an application OrderManagement for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application should be structured in packages using a layered architecture.

The application should be connected to a database that has the following tables: order, product and client. It 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.

For reaching that goal, I followed the steps that were presented in Ms. Pop presentation, alongside with trying to implement as many quests as possible form the objective list. Keeping in mind that my application could still be improved in some areas (I’m going to touch on that subject later), it has the following:

* It has a database with the desired tables (client, product and order) alongside with the relationships between them: order table has two foreign keys(id\_client and id\_product) that are used to link it to the other two tables using the IDs of the clients, respectively of the products.
* It presents a file parser which is similar to the one I implemented in the assignment number 2, only this time it is a little bit more complex, because the data that we need to read is a little bit more ‘dynamic’ than last time.
* I have documented each class and each method using a Javadoc which will be attached alongside the project
* For each order command in the input file, I created a PDF output that will be the bill of the order
* The product stock will be decremented every time an order is placed; If the quantity ordered is greater than the current stock, then a PDF output file should be generated showing the message “Out of stock” (unfortunately because the operations on my order table do not work properly, as I will touch on this later, it never displays the ‘out of stock’ pdf)
* I created a .jar file
* I created the functions that allow us to view the reports for the tables at a specific time (they will be all attached alongside the project)
* I used the Layered Architecture, having 5 different packages (4+Start one for the main function): dataAccessLayer, businessLayer, Model and Presentation
* I used the reflection technique as I created a generic class that contains the methods for accessing the database: create object, delete object and find object. The queries for accessing the database for a specific object that corresponds to a table will be generated dynamically through reflection.

1. Problem analysis, problem modeling, scenarios, utilization cases

In order to fulfill the program’s purpose, we will be asked to write into an input file the parameters that we will need as it follows:

-there will be 4 major commands: Insert, Delete, Report, Order; for each of them the application needs to do something specific

-the Report command can be used on either of the tables, case in which a PDF output file should be generated that shows the current state of the specific table from our database

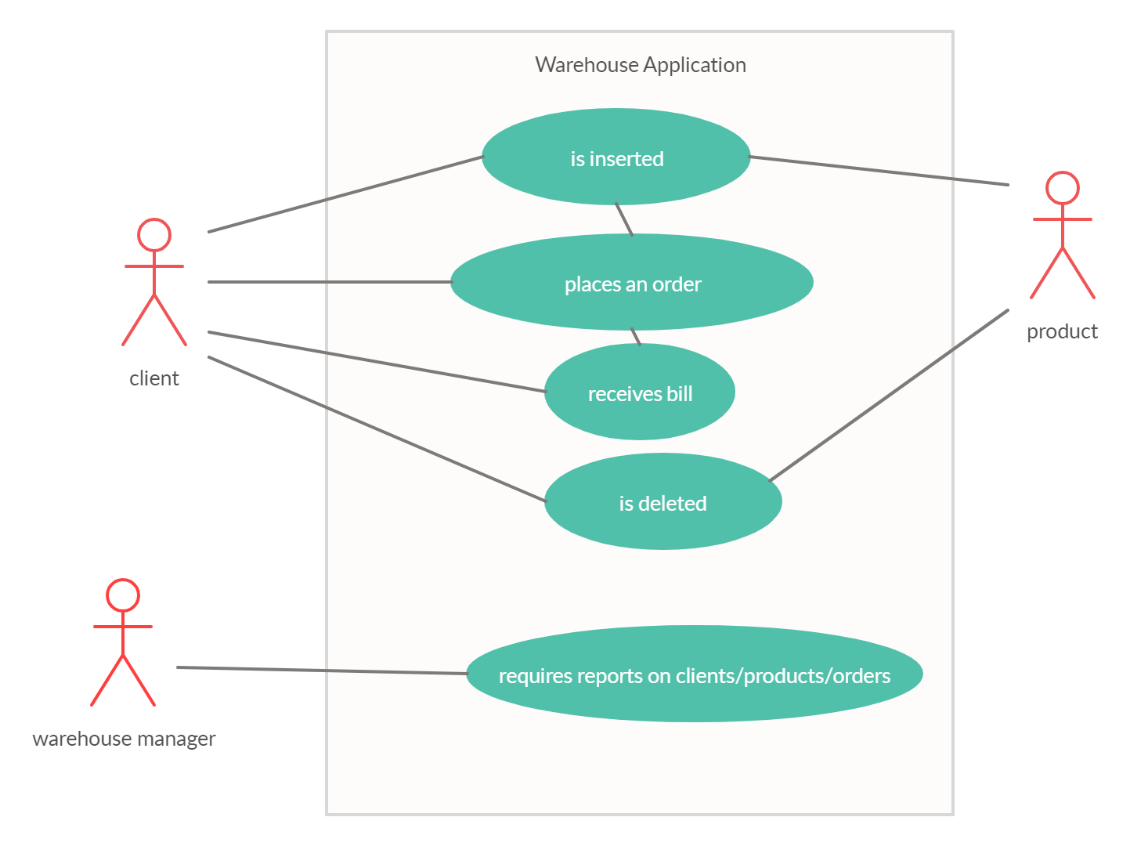
-the Insert & Delete commands can be used on either a client or a product having the specified functionality

-the Order command is used to insert a new order into the order table; each time this happens, a ‘bill’ is created as a PDF output file that contains the name of the person that made the purchase, the product it bought, the quantity of the product as well as the total price (for example if she/he bought 5 apples that each cost 1.0, the total value of 5.0 will be displayed as the price payed)

-each command comes with its “parameters” as a string that follows the character “ : ” used to separate the command from the details of the command; For example, ‘Insert client’ has the following string attached on the same line in the input file: name of the person, address of the person; ‘Insert product’: name of the product, quantity of the product, price; ‘Delete client’: name of the person, address of the person; ‘Delete product”: name of the product; ‘Order’: name of the person, name of the product, quantity ordered

-Report commands are standalone ones, needing no additional information

To show better how the program is supposed to work, I will present some use-case diagrams next (they will be modeled as lists, showing the steps involved in the execution of each case);



Use Case: process of placing an order

Primary Actor: Client

Main Success Scenario:

-The client is inserted into the client table

-The client orders something that is in stock

-A bill is generated containing the order details

-The client either chooses to place another order, case in which the steps are repeated from the step 2 onwards, or it is deleted from the database, case in which he/she can no longer place orders

Use Case: the operations that can be performed on a product

Primary Actor: Product

Main Success Scenario:

-The product is inserted in the product table, its stock being now available for any client to order from

-The product is ordered, case in which the stock decrements with the quantity ordered

-The product can be deleted from the database, case in which the clients can not purchase any more products from the specific deleted stock

-In case the product is already in the database (say we want to add apples, but we already have the product apple), the stock should be updated with the new value (old value + the quantity of the new insert)

Use Case: process of viewing reports

Primary Actor: Warehouse Manager

Main Success Scenario:

-The manager requires the reports for either the clients currently inserted in the warehouse’s tables, or for the products currently in stocks, or for all the orders placed so far

Use Case: process of placing an order

Primary Actor: Client

Alternative Sequences:

-The client places an order for an item with a quantity greater than the one currently available in stock; in this case, instead of placing the order correctly, it will not be placed and a message “Out of stock” will be displayed in an output file

1. Project design (design decisions, UML diagrams, data structures, classes design, relationships, interfaces, packages, algorithms, graphical user interface)

In the following chapter I will discuss how I have split the problem into an object oriented one and the data structures I used for implementing the project, alongside the UML diagrams specific to this application.

Following the model provided to us by our laboratory teacher, Ms. Pop, I organized my project gravitating towards the 4 big elements I presented earlier too: businessLayer, dataAccessLayer, Model, Presentation.

Beside these packages, I also have the Start one which contains the main class, where the application starts.

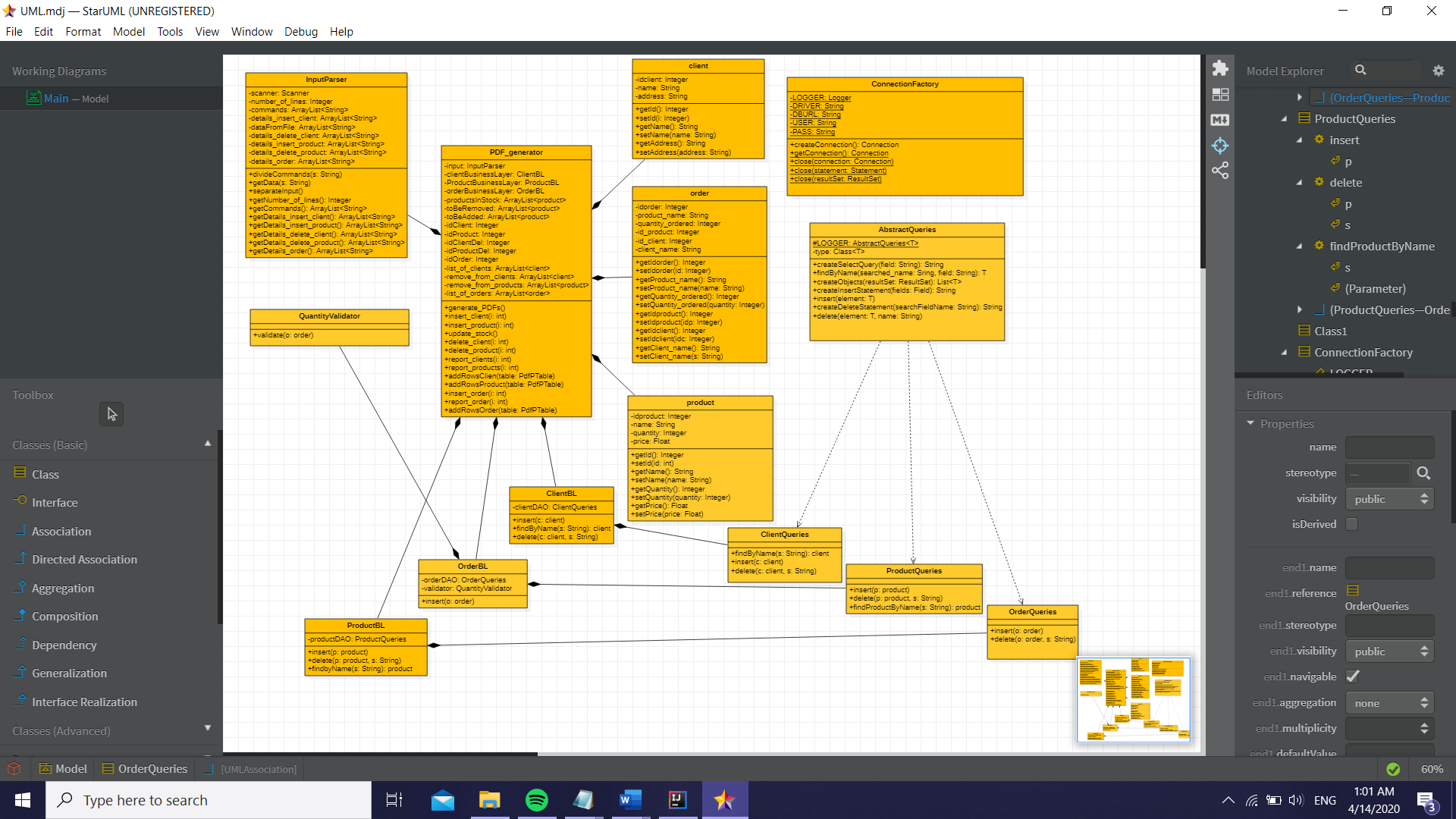
-Presentation package contains the following classes: InputParser & PDF\_generator (which has more than 300 lines of code but I honestly couldn’t fit everything (plus the Javadoc descriptions) into that range, I will try to further more think of ways I can truncate the code); This package deals with, as the name says, presentation of the application, more so the way the input and output are dealt with;

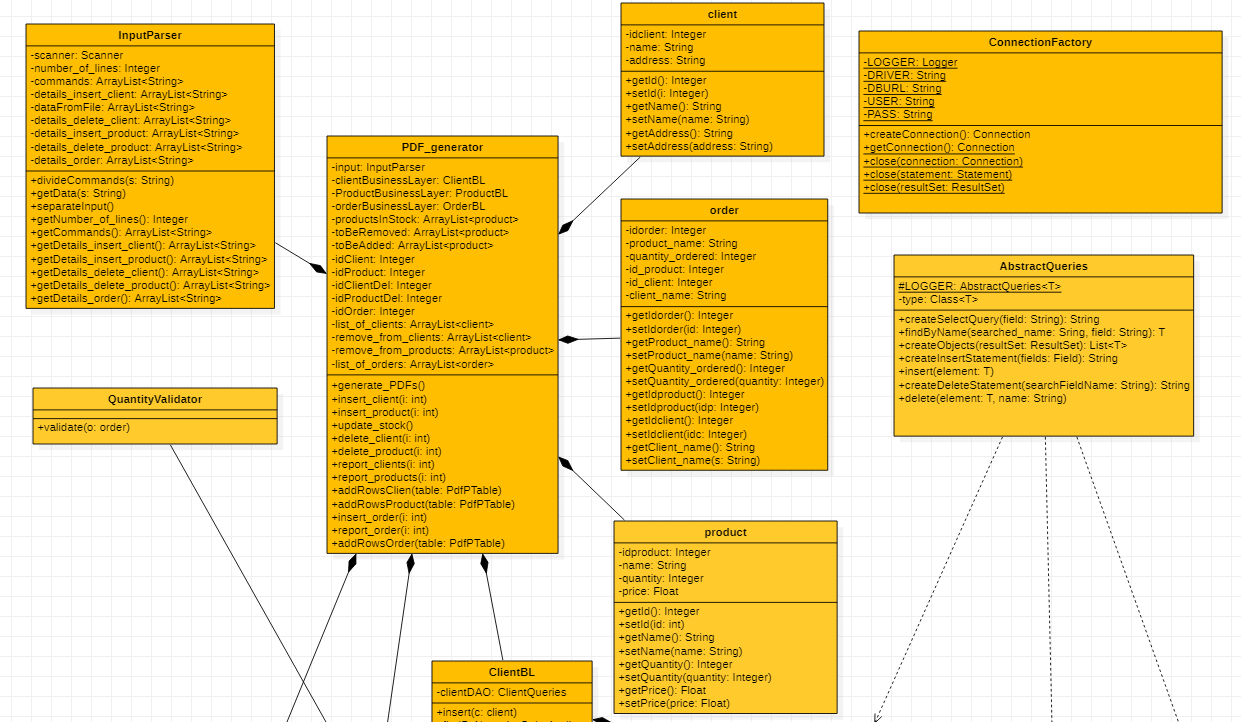
-Model package contains the classes for creating the objects of type client, product and order; these are the same objects as the three tables contained in the database and each class has the same parameters as the ones described as the tables’ columns.

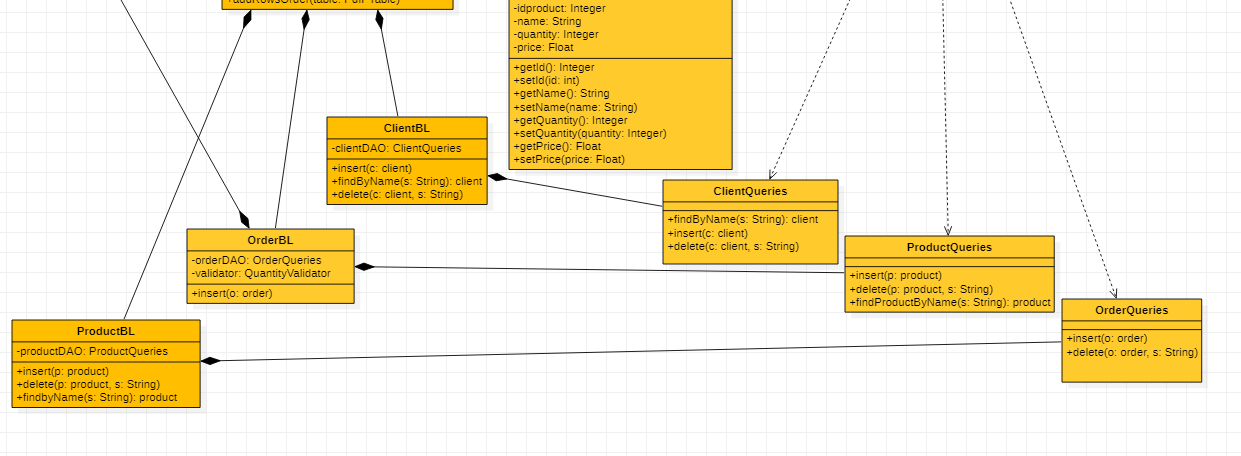
-dataAccessLayer contains the following classes: AbstractQueries, ClientQueries, ProductQueries, OrderQueries (which contain the queries implemented; AbstractQueries is the main class that implements them, but because I chose to use reflection technique, the other mentioned classes extend the first one for their specific objects) and ConnectionFactory (which determines the database connection);

-businessLayer contains the following classes: ClientBL, OrderBL, ProductBL (the classes that encapsulate the application logic, being the link between dataAccessLayer and Presentation) and QuantityValidator (the class that validates the quantity in case of an order);

Next, I will attach the UML diagrams specific to my project, along with the relationships between them (the following UML diagram is made using the StarUML app):







1. Implementation

In this chapter I will elaborate on each class I created, explaining the most important methods that each implement and why I chose them the way I did.

* client class

This is the class that creates objects of type client. The specific parameters are the ID, the name and the address of one specific person. Because we are going to use this object to insert it into a table, the ID must be unique for each client, idclient being the primary key of that table.

The methods implemented by this particular class are pretty simple, only getters and setters and the overwritten toString method, all of them designed to help us further into our implementation.

* product class

This is the class that creates objects of type product, as the one before it. The specific parameters are the ID, the name of the product, the quantity and the price. Once again, because we are going to use this object to insert it into a table, the ID must be unique for each product, idproduct being the primary key of that table.

The methods implemented by this particular class are once more only getters and setters.

* order class

This is the class that creates objects of type order. The specific parameters are the ID, the name of the product in the order, the quantity ordered, the ID of the client that made the order, the ID of the ordered product and the name of the client. The ID should be once more a unique one, idorder being the primary key of that table.

The methods implemented by this particular class are once more only getters and setters.

* InputParser class

This method will be used to read from the input file given as argument. For doing that we will use a Scanner object and a File object which will get as a parameter the input file. While the scanner can read from the file, we will add each line read to an ArrayList<String> named dataFromFile and we will increment the variable number\_of\_lines which we will use later to parse through the commands.

The method divideCommands() gets as parameter a string which will certainly contain the character “ : “ and it will divide this string using string.split in order to get the first part which we will consider as the command (can be either Insert X, Delete X, Report X, Order);

The method getData() does almost the same thing, but now it will keep the other part of the string, the ‘details’ of clients/products/orders. For example, for the line “Insert client: Pop Alex, Cluj-Napoca”, the method will add the string “Pop Alex, Cluj-Napoca” to an ArrayList<String> named details\_insert\_client. We will use the information in all of the string created this way later, when we insert/delete in tables.

The method separateInput() puts together the first two methods.

The rest of the methods represent only getters and setters.

* PDF\_generator class

This class is the one that performs all the operations on tables as well as generating the desired PDF outputs. I have to admit that I did not respect the 300 lines of code/class requirement here, but I will furthermore work on this project and I will try to adapt this class such that it will fit the criteria.

The method generates\_PDFs() is the ‘main’ one, combining all the other ones presented later; it iterates through the commands and depending on which one is selected, a specific operation will be performed.

The insert\_client() will do the following: if the command is “Insert client”, it will take the first string saved in the details\_insert\_client ArrayList and it will separate the string using the regex “, “ into parts that will fit for the name and address of the person. It will attribute a unique ID and then it will insert it into the table using a function presented later in the clientBusinessLayer class.

The method insert\_product() does the same thing, only with products; but this time, an object having the same name can be inserted, which means that we will need to update the stock. To do that, I keep into an ArrayList<product> the old product that needs to be deleted and in another one the one that needs to be added, having the quantity = old + new inserted one.

The method update\_stock() uses the two ArrayList<product> mentioned earlier to update the stock.

The method delete\_client() works similarly with the insert one, only this time the client is not added to the table, but instead it is deleted using another method of the clientBusinessLayer class we will get to a little bit later on.

The method delete\_product() works in the same way, only deleting a product; but because the details taken from the command give only the name of the product and we also need the other identifiers, we will parse through all the products in stock to get the rest of the information needed.

The methods report\_clients(), report\_products() and report\_order() all work in a similar way, creating a PDF output file displaying the current tables in the database. In order for these methods to work, 3 new ones are needed, to add new rows to the table: addRowsClient(PdfPTable table), addRowsProduct(PdfPTable table), addRowsOrder(PdfPTable).

The last method, insert\_order() is the one that should insert orders into the table; unfortunately, it doesn’t work accordingly, this is something I will definitely work on for next week. Because the order details only present everything except for the client and product ID, we should take them from the already generated and populated tables using findByName (this seems to be my problematic function, I was not able so far to figure out exactly what’s going wrong, but I will keep trying to do that). Also, once an order is placed, the quantity of the stock should decrease if the validate function invoked does not generate the “Out of stock” PDF.

(As an update, I got to fix the findByName function, now linking the tables in the correct way but only after the second program run, and that’s because I chose to insert all the product at the end of the application running, and this causes problems when a product has to be retrieved from the product table which is at that moment empty; I will surely think about another way to solve this further)

* ConnectionFactory class

This is the class that establishes the connection between the database and the application; I created it using the steps from the support presentation and it has the following methods:

-createConnection() which will generate the connection with the database

-close(Connection connection) which will end the connection

-close(Statement statement) which will close the statement

-close(ResultSet resultSet) which will close the result set

* AbstractQueries class

This is the class that implements the queries that will be used to perform operations on the database. Because I chose to use reflection, it will be an abstract class that will be extended by other 3 classes, each one of them for a specific type of object. That’s why it has as a parameter Class<T> type.

The method createSelectQuery(String field) returns a string that is created using parts needed to create a query; in the end it will look something like this: SELECT \* FROM some\_table WHERE field =?;

This query will be further used in the next methods.

The method findByName(String searched\_name, String field) is similar to the one presented in the support given, but instead of searching for the ID, it is searching for a given string, returning the first object that corresponds to the given requirements.

The method createObjects(ResultSet resultSet) is the one that creates the object according to the specific fields of the object; This too is a method implemented using the help of the presentation.

The method createInsertStatement(Field[] fields) is the one used to create a string, that will be returned at the end, underlining the insert query needed for a specific table. The ID portion is skipped, since it will be autoincremented.

The method insert(T element) will be used to insert a specific type of object into their table. It uses the previously defined method to get the query for insertion, then, depending on the type of the field, it will choose the kind of variable to insert. Again, it skips the ID.

The method createDeleteStatement(String searchFieldName) defines, like the create one, a query used in the following method in order to delete from a specific table.

The method delete(T element, String name) deletes from the table the object that has the required name given by the second parameter. It uses the query created previously.

* ClientQueries class

This class extends the one previously mentioned, implementing the functions insert(), delete(), findByName(); It implements all of these functions for object of type client.

* OrderQueries class

This class extends the AbstractQueries one, implementing the insert() and delete() methods for objects of type order.

* ProductQueries class

This class extends the abstract one just like the previous two, implementing insert(), delete() and findByName() for objects of type product.

* ClientBL, ProductBL, OrderBL classes

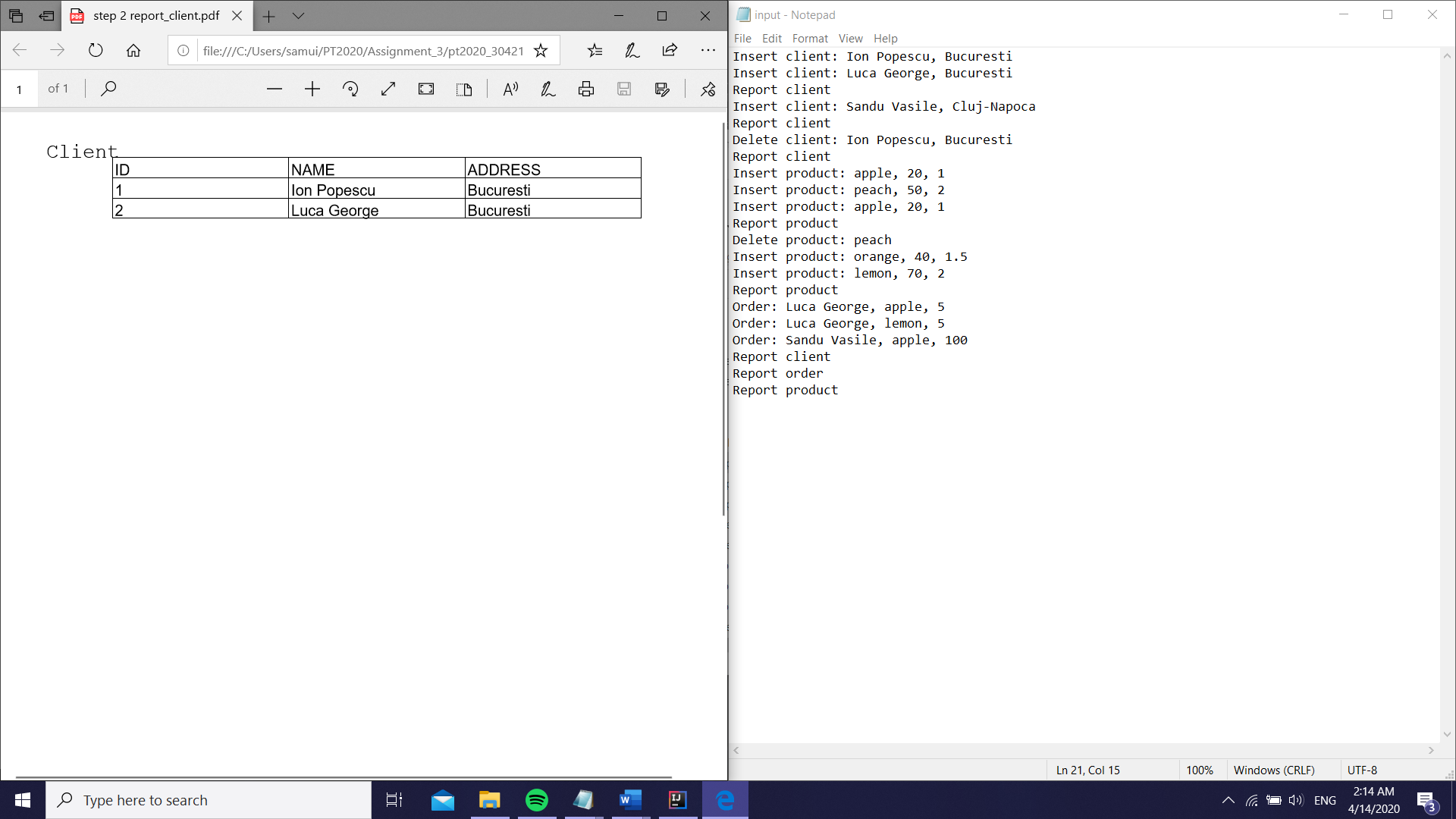
These classes are similar, all of the taking as parameter an object of type XQueries (X=Product/Client/Order) and implementing once again their methods, doing the link between the query part and the presentation part.

* QuantityValidator class

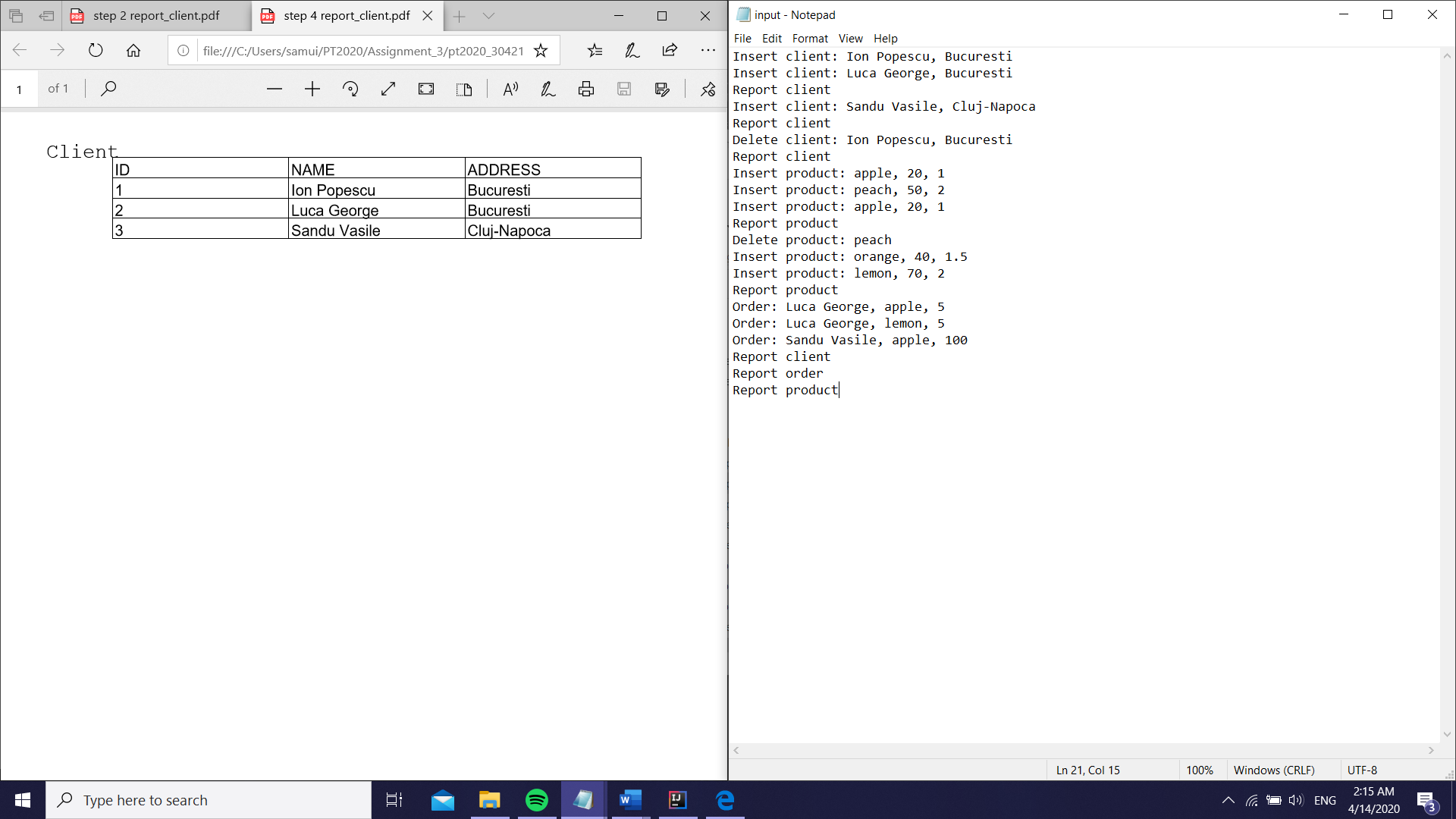
This class has only one method that checks if the quantity from an order is smaller than the stock quantity for that specific product. If it is not, it will create a PDF with the message “Out of stock” and the order will not be proceeded. In order to find the stock of a specific product, we need to find that product using the findByName function once again.

1. Results

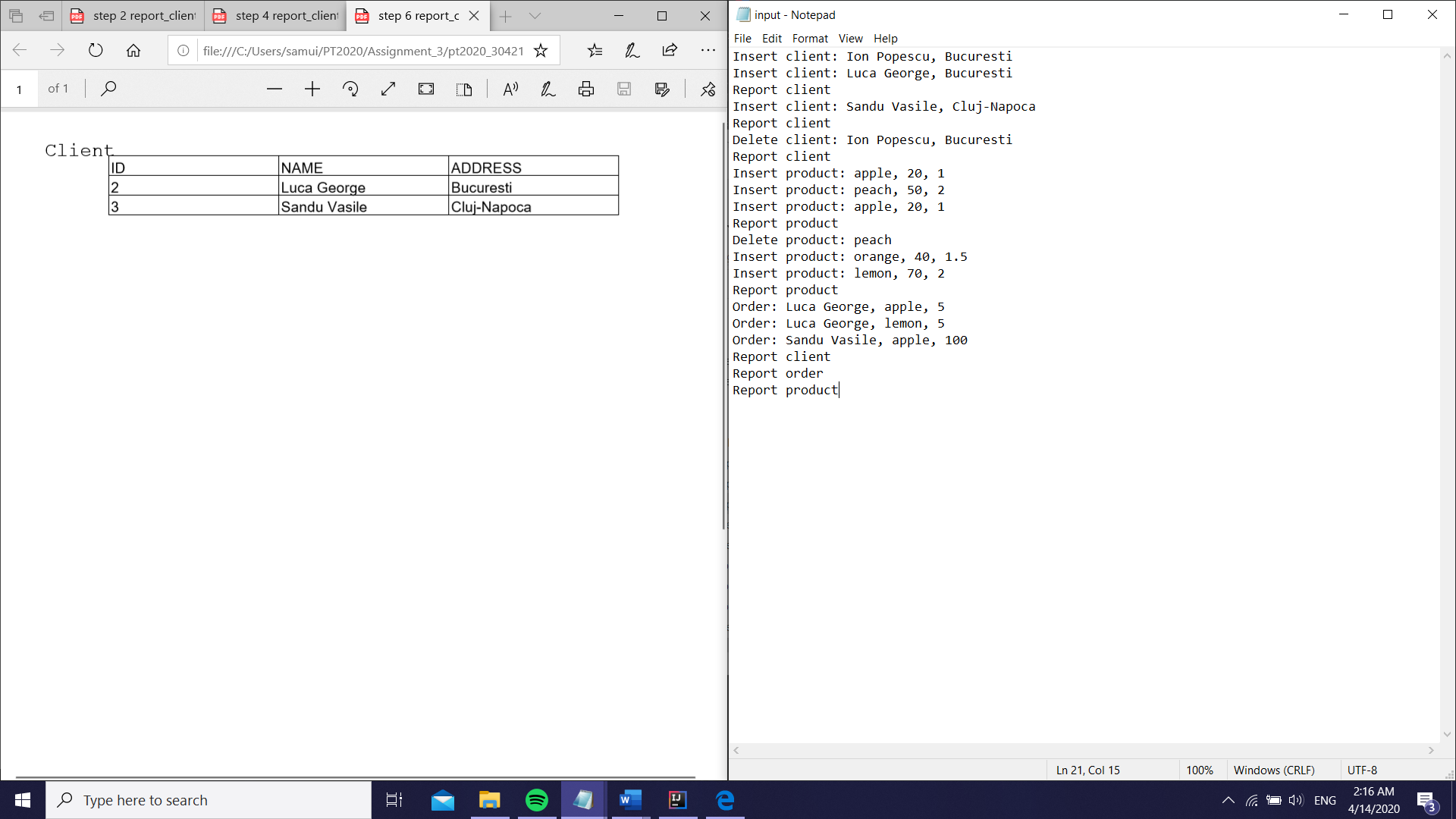
I will attach some screenshots with some of the output, and in addition to that I want to point out that I used the console of IntelliJ to display at each moment of time the current operation that is performed (Here also could be easily observed the fact that the application presents some problems at the level of the findByName method).



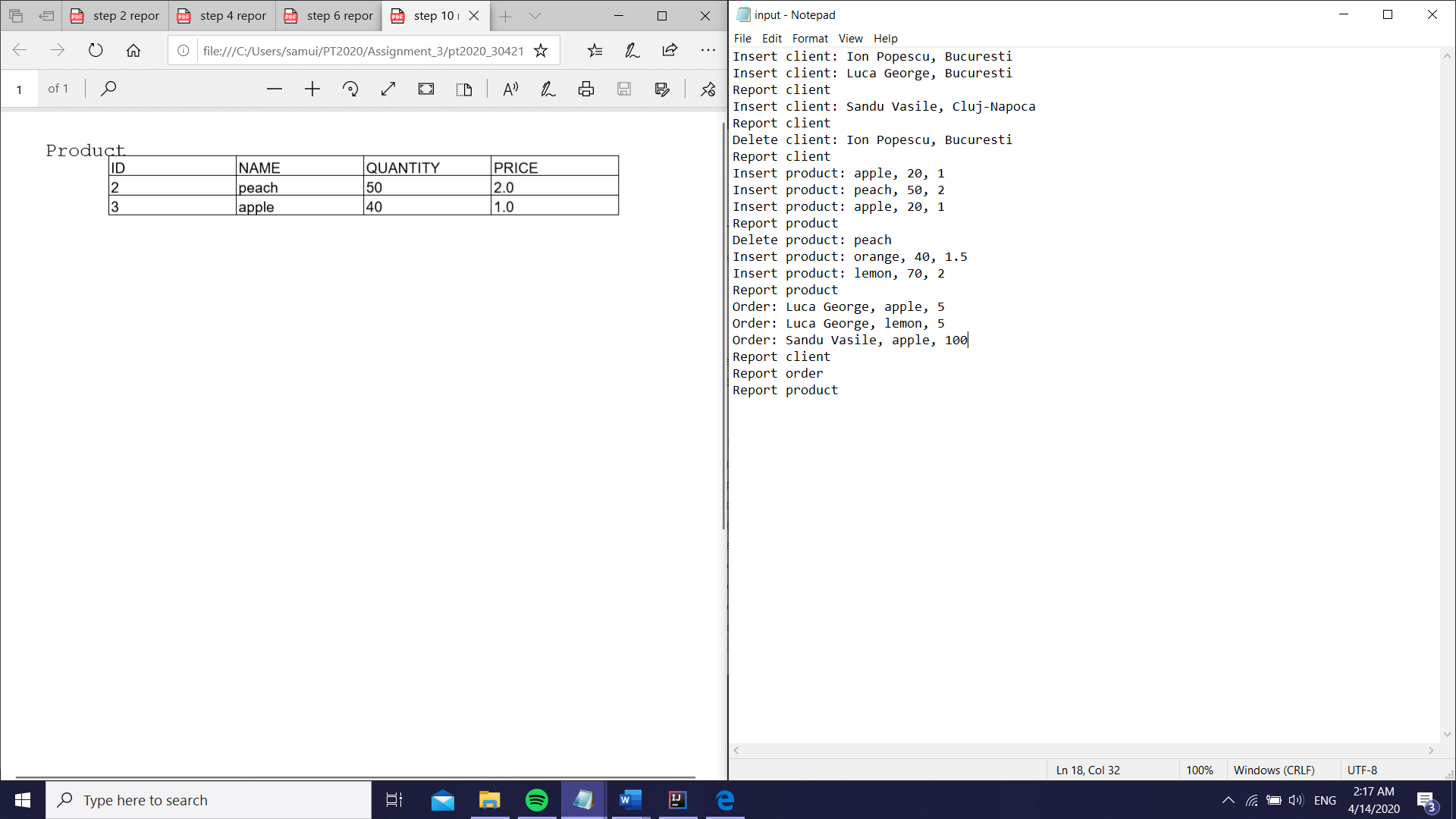
The output in the moment 2(indexed from 0), for the command Report Client;



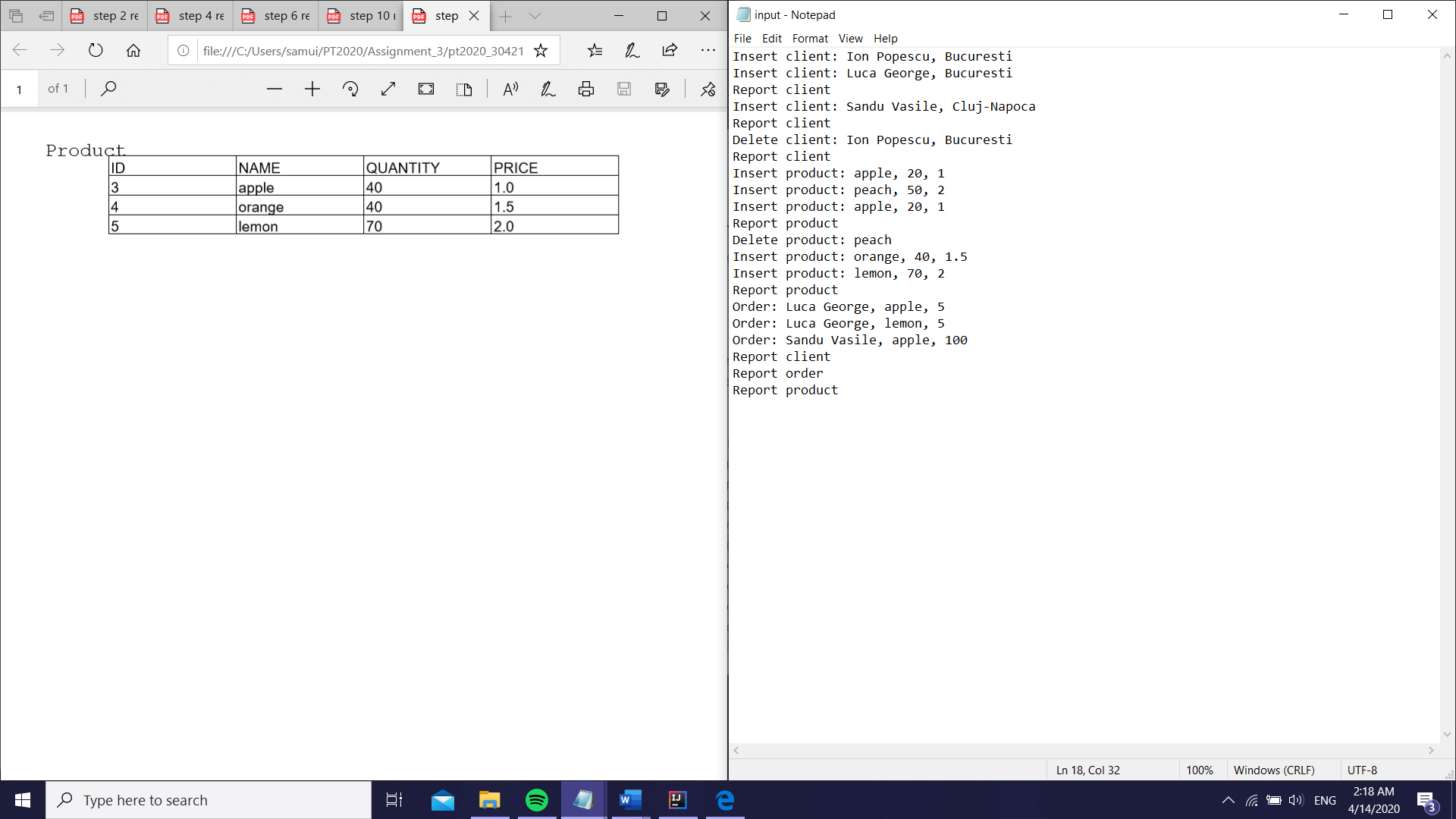
The output at moment 4, for the second Report Client;



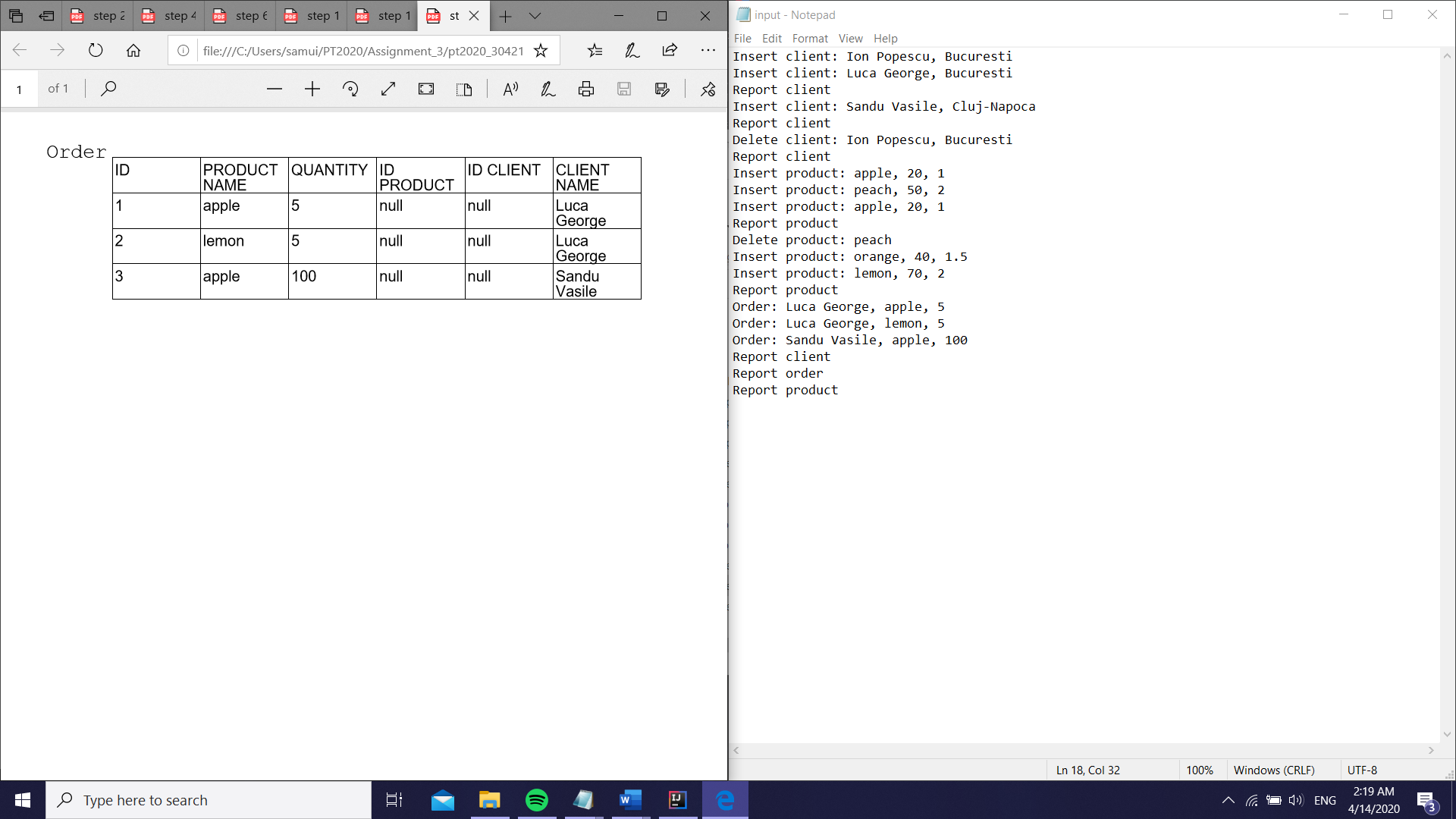
The output at moment 6, after the deletion of the first client;



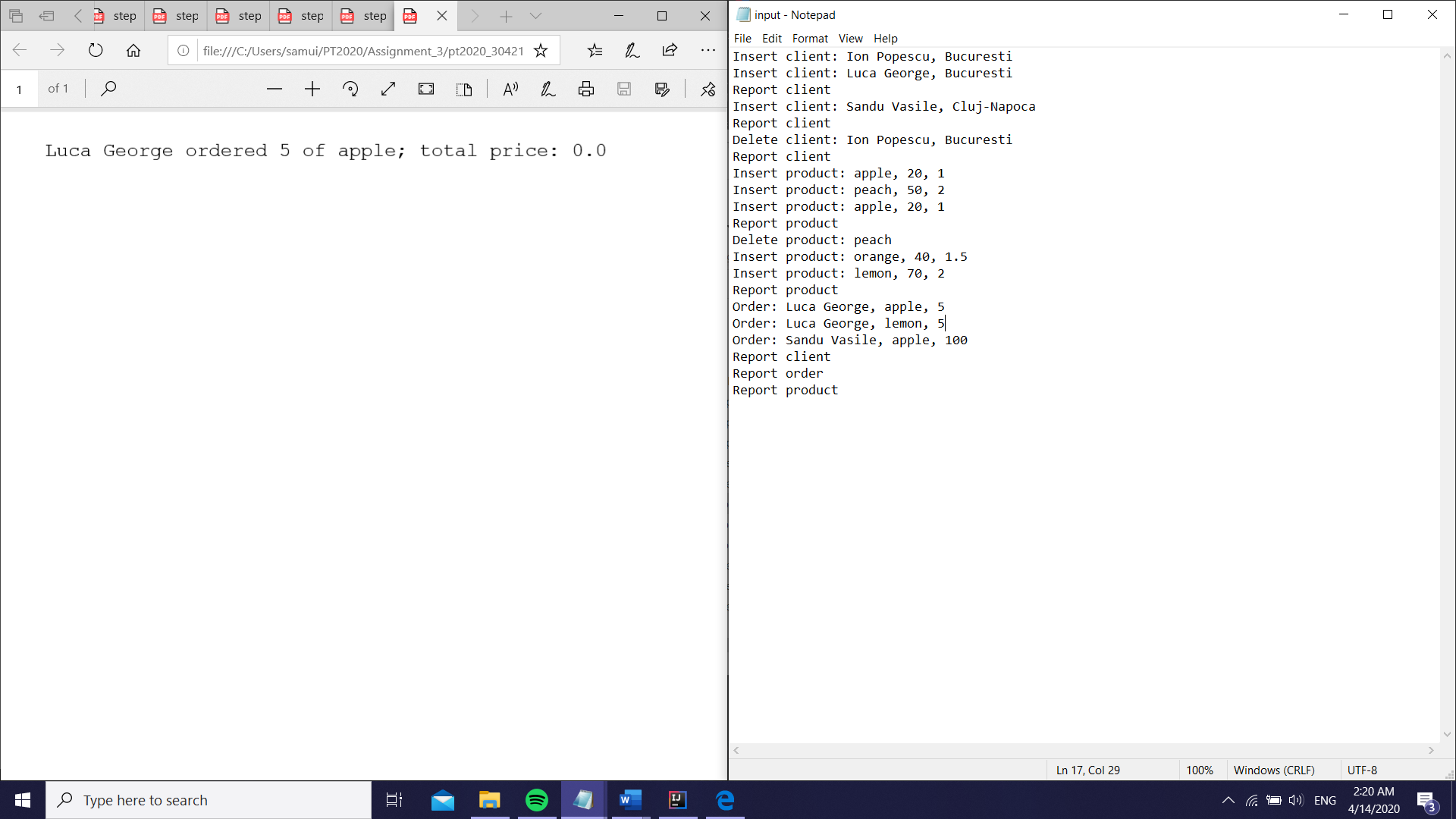
The product table, at the first report; it can be observed that the quantity is updated after the second insert;



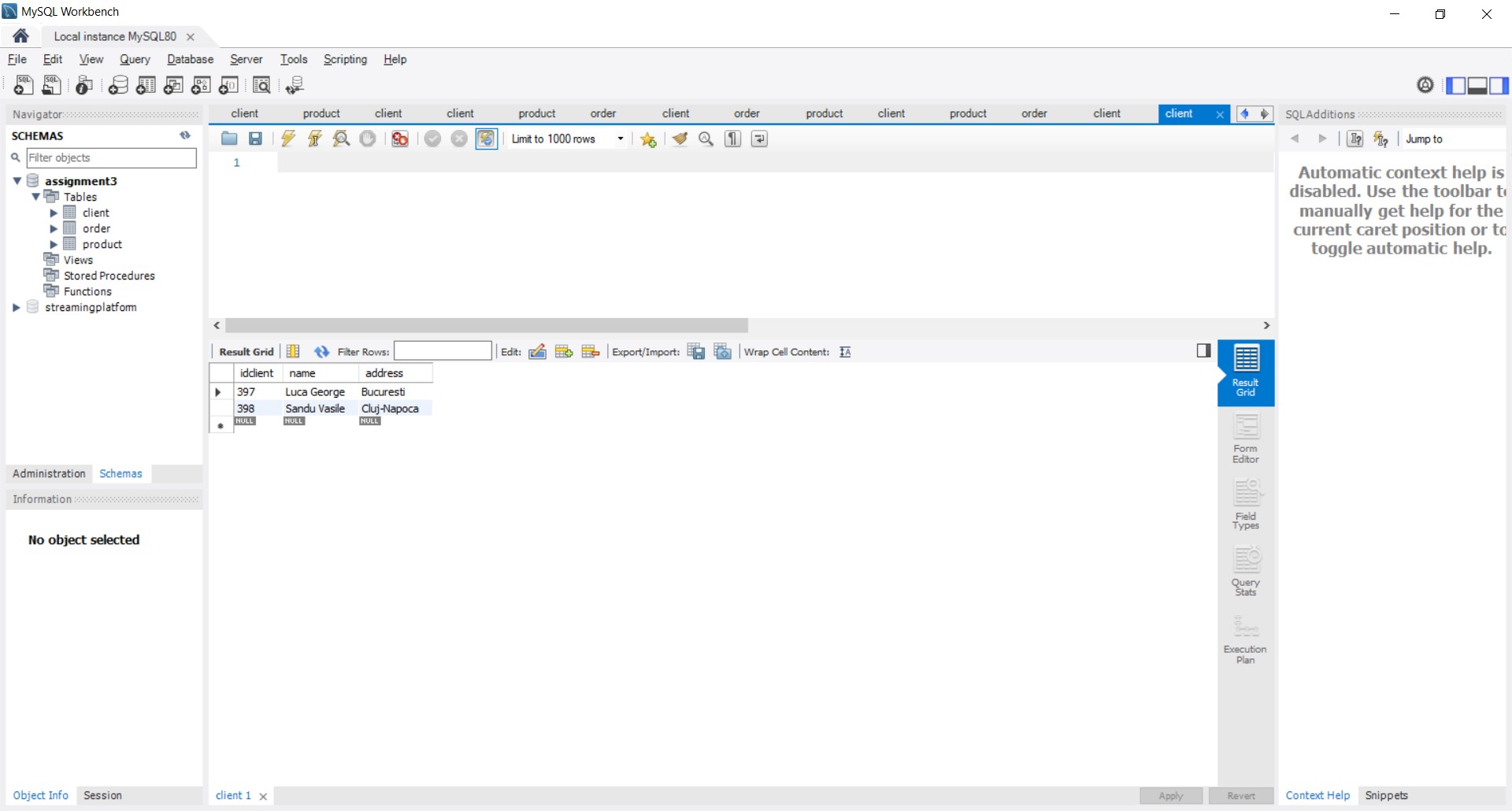
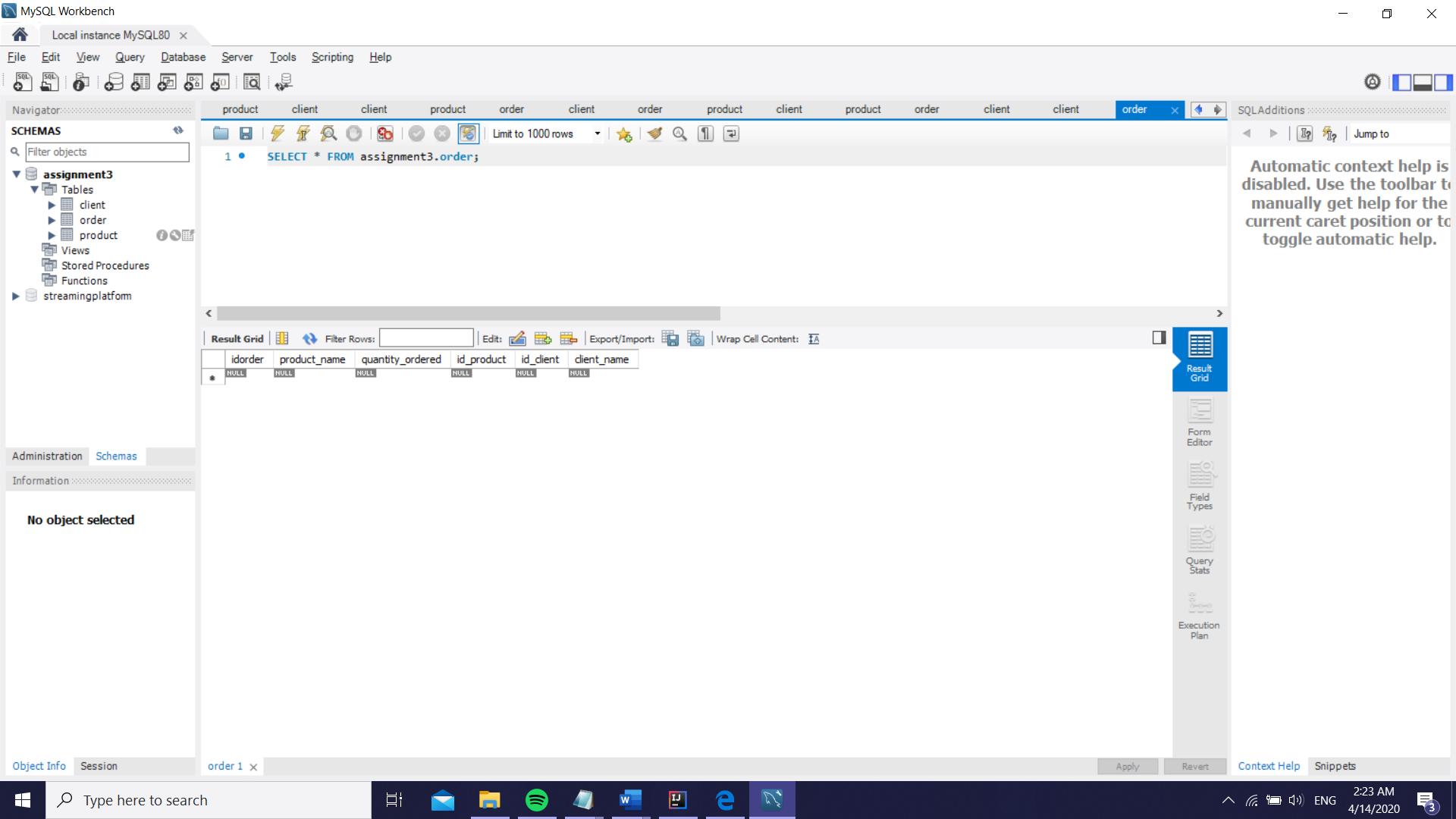
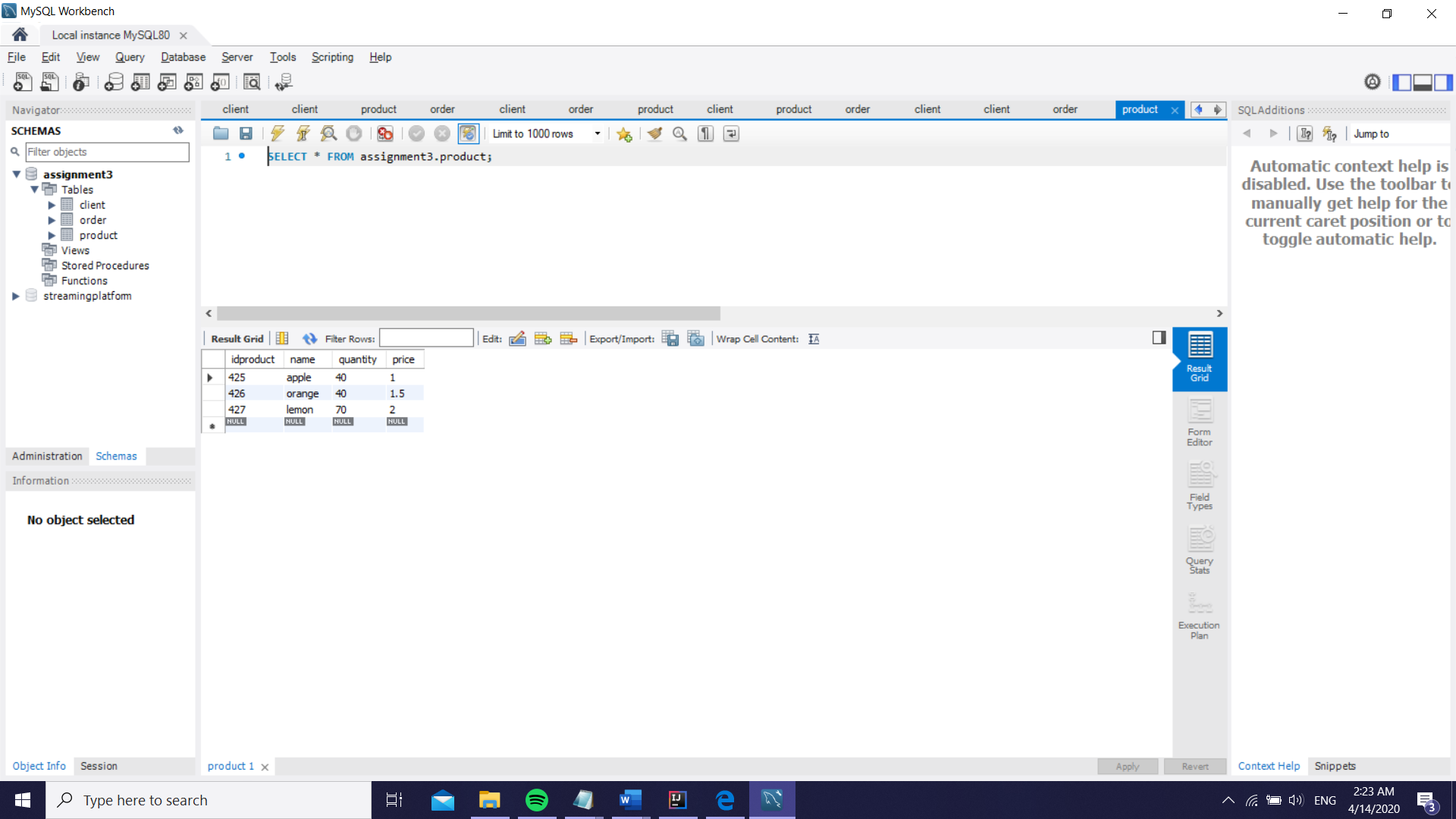
The product table after the changes made before the second report;



This is the report order at the end of the file, but as it can be seen, the IDs are null due to the error I get at the findByName method;

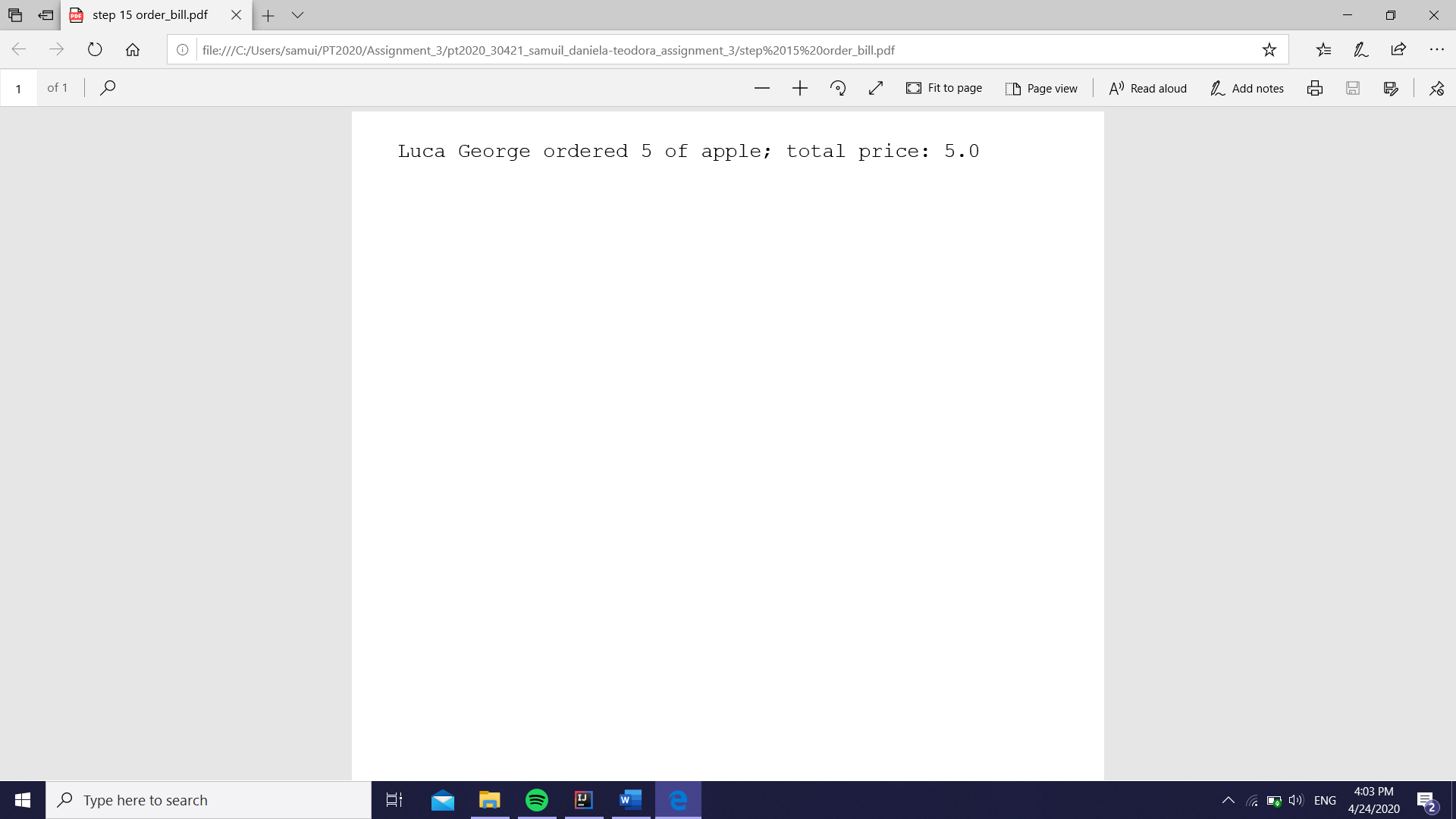


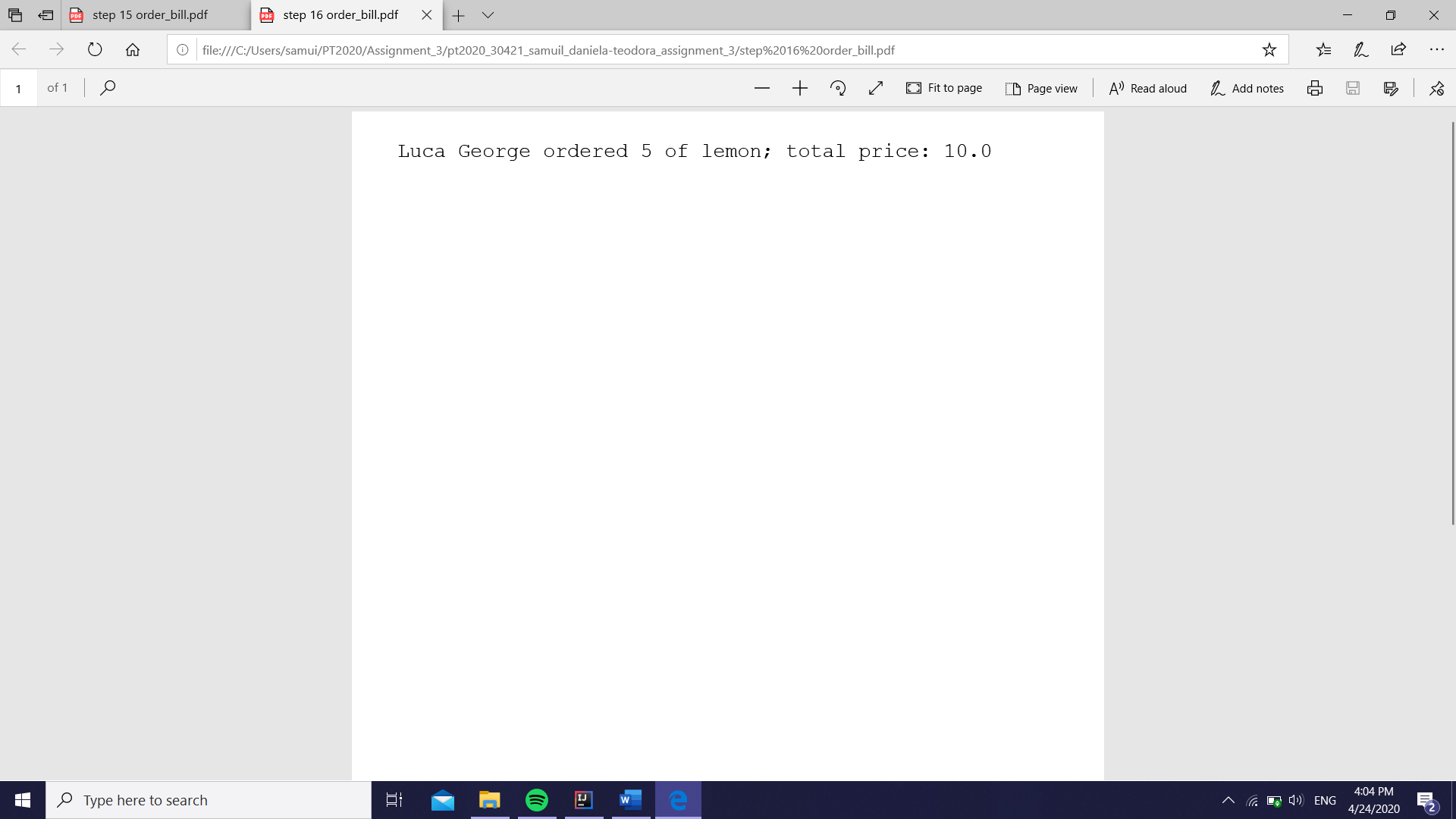
This should be a bill example, but because of the issue, the “Out of stock” message will not be displayed and the price can not be computed (I need the product’s price so I can compute the total);

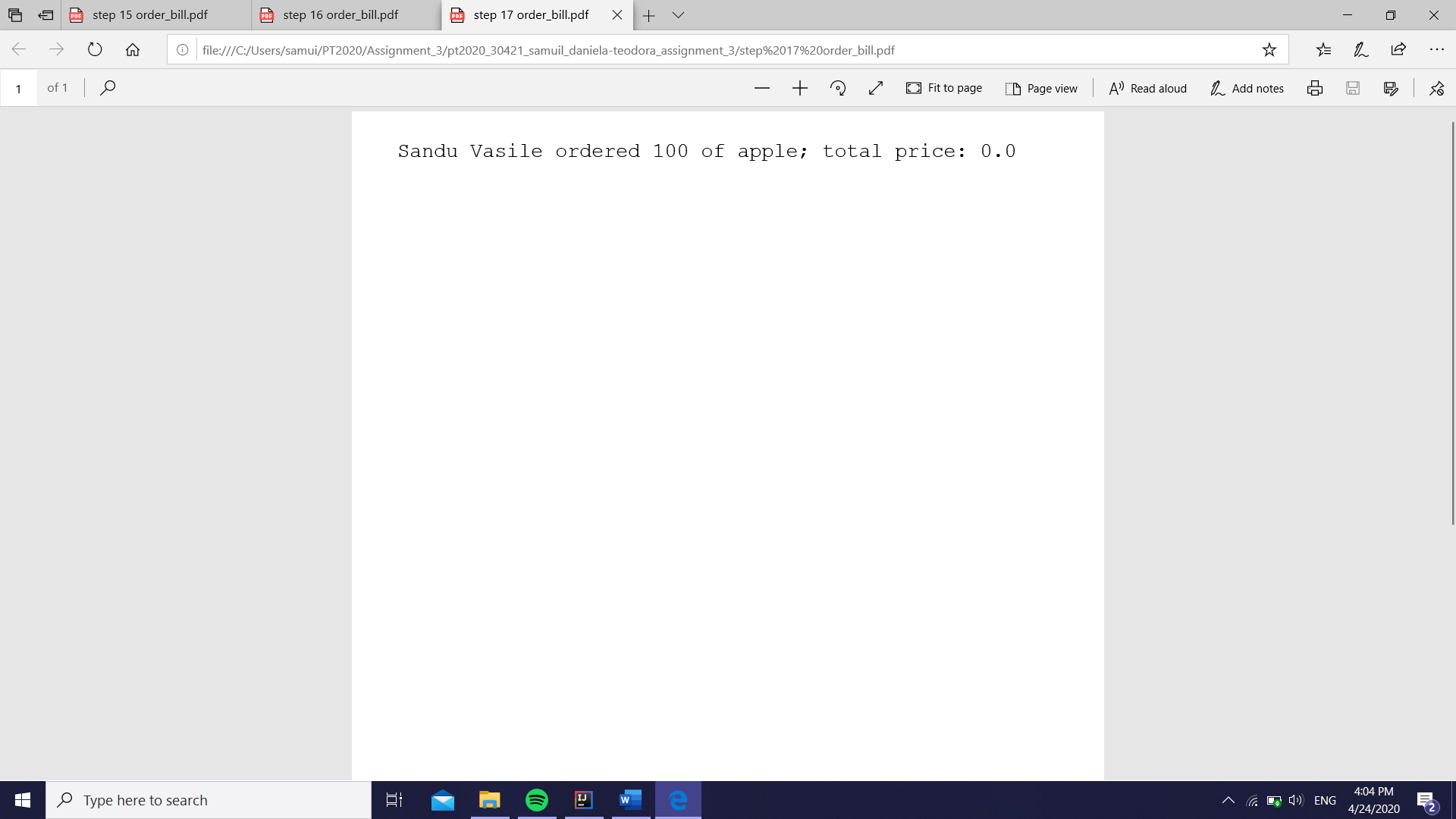
  

The pictures above illustrate the way my database looks like after the completion of all the commands;

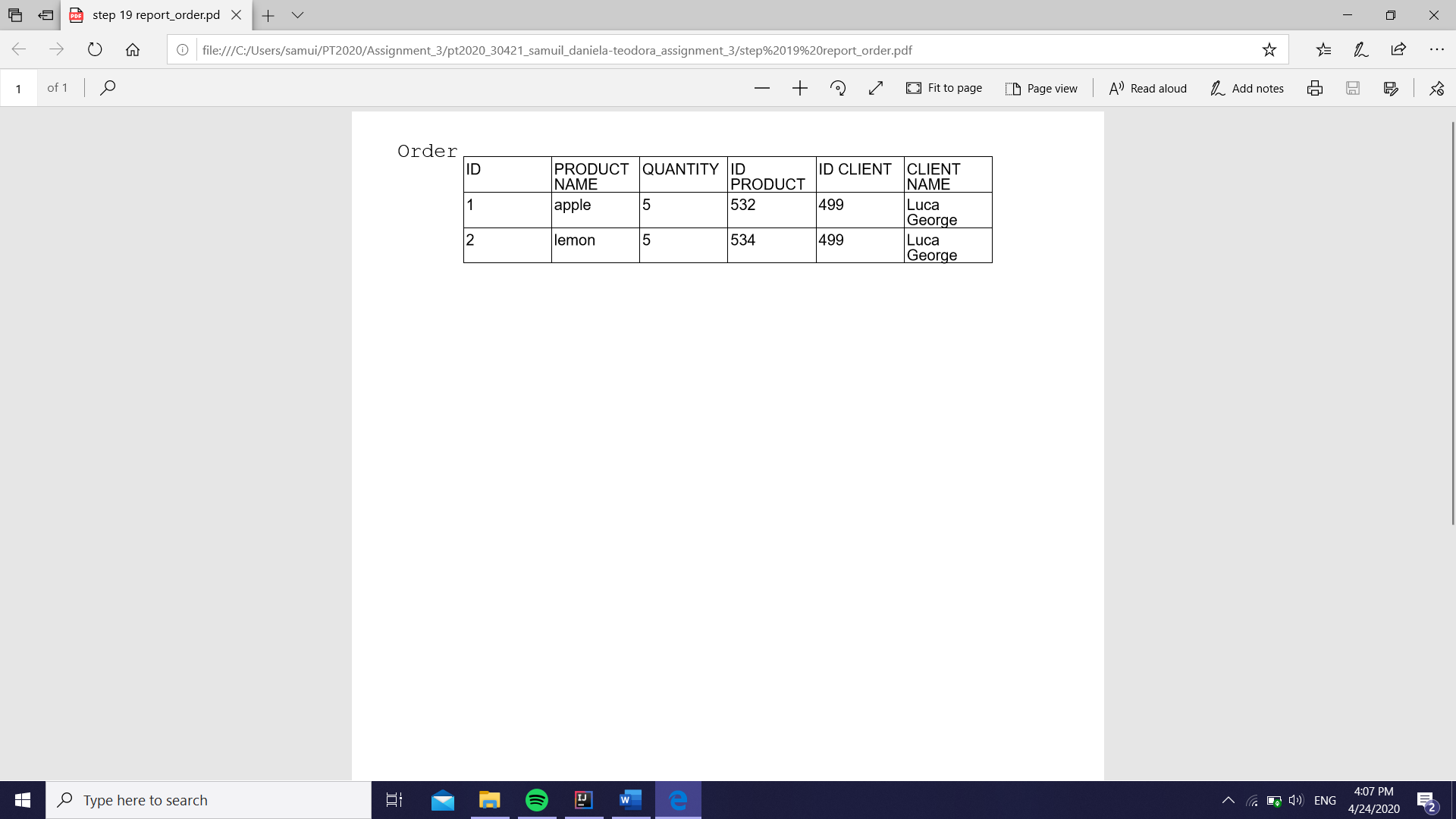
As a futher update regarding the project, I will attach some more screenshots showing that the application works accordingly, just that is doesn’t add the orders in the table ( I received an SQL syntax error for insertion but I couldn’t find the root of it, and it was confusing because as I tried to do a generic insert, for the client and product tables it worked correctly but for the order one it gave me a warning). Though, to show that the program still work in the desired way with the orders and stock, I implemented some additional lists to keep the orders and the products and in such way I can display them in the pdf even though the orders do not fill the table.



The bills showing the proper total price this time



Because the insertion in the table is not completed, thus the creation of the “out of stock” pdf does not happen; but instead, to show that the order is not placed, I chose to still show a bill type PDF but having the total 0 (meaning that the order was not able to be placed)



The order report, showing futhermore that the last order could not be placed

1. Conclusions

In conclusion, I found this project the most interesting so far and really practical because of the usage of databases. Also, it required a lot of new knowledge about linking a database to the project, about Javadoc, but mainly about the way in which a database can be manipulated using object-oriented programming. I plan on working on this specific project a little bit more as I am not as satisfied with how it turned out (I mainly want to resolve the orders problem) but I believe it was really good practice. I will most probably send the assignment again next week with a hopefully improved version.

\*I couldn’t entirely fix the order problem, I will still try to figure out why it wouldn’t insert the orders into the table, but I worked a little bit on the improvement of the project\*

1. Bibliography

I will attach some links I felt were useful for my work within this project, as well as some mentions I thought I should make:

* The main support I based my application on is found in Ms. Pop presentation:

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

* In order to connect my database to my project I followed the instructions linked below:

<http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>

* For inserting in a table that has the autoincrement field, I used the directions provided in the following link:

<https://www.codeproject.com/Questions/1042128/How-to-auto-increment-user-id-in-mysql-in-java>

* To understand better the layering architecture, I used the site:

<https://dzone.com/articles/layers-standard-enterprise>

* For more information in the reflection technique, I used:

<https://www.oracle.com/technical-resources/articles/java/javareflection.html>

* For the PDF output files creation, I found support in the following explication:

<https://www.baeldung.com/java-pdf-creation>

* In order to correctly generate the Javadoc files, I followed the steps presented here:

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

* To generate the SQL dump file, I followed the instructions here:

<https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>

* For a little guidance on the delete statement, I read the information below:

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