Order Management

1. The assignment's objective

The main objective of this assignment is to develop an application for processing customer orders for a warehouse. This is done by following a particular pattern, making sure that certain secondary objectives are met, such as: the product, clients and orders are stored in a database in order to be processed, so a connection to the database is essential. Also, the CRUD operations should be available for all the tables in the database, resulting in an abstraction of them (the class AbstractDAO will be further discussed in chapter 4). What's more, we have to make sure that every product/order introduced is valid (the amount or the price should not be negative) and for each order placed, the stock should be verified and, if the order can be completed(the order quantity doesn't exceed the stock quantity), the stock and the amount the client that places the order should be updated. Last, but not least, for each client, a Receipt (pdf file) should be generated with the products that were bought and with the total amount to pay, and, also lists with the existing Clients/Products/Orders should be generated, also as pdf files. Another important aspect is the parsing (reading and identifying) of the commands: an input file containing on each line a command is given as argument, each corresponding to an operation, so the parsing is necessary in order to ensure full functionality.

2. Problem analysis, modelling, scenarios, use cases

The Order Manager should be implemented in relation to the database: the relationships between the tables in the database should be kept as they are, i.e. for example, when a Product is deleted, all the orders with that product should be deleted, same goes for the client. Also, when an order cannot be processed, a pdf file containing a message that informs the Client that there is an understock should be generated.

For each table, the CRUD operations are, in essence, very similar, so an Abstract Class that implements these operations is very useful. The classes that are in relation with the database having specific methods for operations, for example: the PersonDAO class implements a findByName() method that is very useful when trying to insert a new person(if the person exists => findByName() does not return null, we cannot add it), method that is not typical for every table in the database (some tables might not have a name field).

Another important aspect is the logic behind the CRUD operations, taking into account aspects such as: for a new Order added, there should be a verification done for the client that places the order and the product that is ordered to see if they exist, because if they don't, then an order can not be placed or in order for a person to be deleted, it has to exist at first (these aspects and many more, will be detailed in the 4th chapter).

In order to summarize these aspects, and to present how the application works, a use case will be displayed below that shows the situation in which a new Order is to be added.

Use Case: Order Management

- The parameters corresponding to the person and the product are given for a new order are given
- The Person that placed the Order is searched, as is the Product that was ordered
- If they exist, the quantity ordered is compared to the stock.
- If the stock is not exceeded, a new Order is created and inserted into the database, and a pdf with the order(s) of the person that belongs to this order is generated.
- The stock is updated. (we have to subtract the ordered amount).
- If the stock is exceeded, an UnderStock message is generated in a pdf file informing the Person that ordered the Product that the quantity ordered exceeds the stock available.

Similar Use cases can be presented for other operations (creating, reading, deleting, updating) regarding other objects from the database (Clients or Products).

3. Design (design decisions, UML diagrams, data structures, design classes, interfaces, relationships, packages, algorithms, user interface)

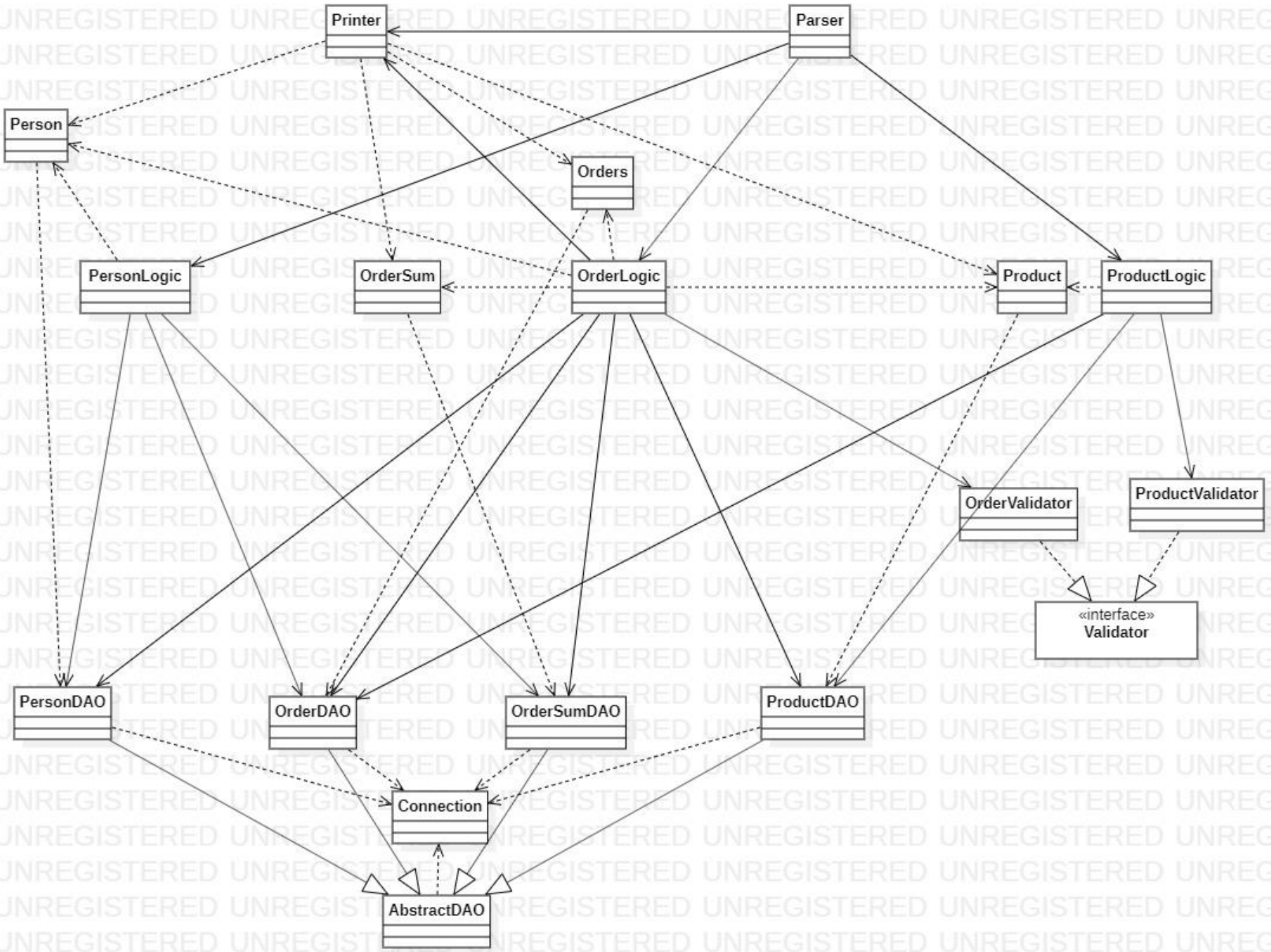
Talking about design decisions, the assignment has 4 Packages: the Model Package, containing classes corresponding to tables form the database (Person, Product, Orders, OrderSum), the DataAccessLayer, where the operations regarding the database are implemented, with classes such as AbstractDAO (for the generic CRUD operations), PersonDAO, ProductDAO, OrderDAO, OrderSumDAO (for the more specific operations in relation to the database), the BusinessLogicLayer with classes that contain the logic that is behind the database operations (such as the logic that is behind adding a new order: the Person that made the order and the product orders have to exist, the stock cannot be exceeded by the order made. After the order was done, the stock has to be updated as does the total sum that pe client(person) has to pay.). The classes in this package are: PersonLogic (the logic behind inserting and deleting a Person), ProductLogic (the logic behind inserting and deleting a Person) and the OrderLogic which was explained a little bit earlier. In the BusinessLogicLayer there exists another package, Validate, containing an interface, Validator, whose method validate() is implemented by the OrderValidator and the ProductValidator classes (they make sure that no order or product are inserted with quantity <=0 or price <= 0). The package in which the connection to the database is done is called Connection. Another important Package is the PresentationLayer, containing two classes: Printer(in which all the pdf files are generated) and parses(in which the input file is read and parsed, identifying the commands and the operations that have to be performed in order for the command to be fulfilled).

In terms of relationships, there is a strong connection between the classes, as the database operations are done on the Java objects defined in the Model package, and in turn, these operations are used in the logic layer, that is later used to print pdf files or to performed the commands that were parsed in the PerentationLayer package. All relationships between the classes can be seen below, in the UML class diagram.

Another import aspect when talking about the design are the classes implemented, and two classes that are very useful in this assignments developing are the AbstractDAO: in which generic CRUD operations are implemented and the Printer class, by having the generic method printAll() that is used to print all the objects contained in a table from the database. These two classes, one by being generic and the other one by containing generic methods are very important due to their practicality(no matter the Object that the methods from the AbstractDAO or the printAll() method are called for, they will work regardlessly => we avoid writing the same code for two or more different Objects).

Another design decision was the addition of a new table in the database and new classes (OrderSum, OrderSumDAO). By adding this additional table, we will store the total amount to pay for a person only once, in this table. If we weren't to have it, then for each order in the orders table we should have kept the total amount to pay for all the person's orders(for instance, if a Person had 20 orders with a total amount to pay 200, we would have kept that information for all the orders => 20 times, from which 19 of them would have been redundant. So, by adding this table, we link the order with the person's id and that id with the amount to be paid for that person, and because of that this information will be stored exactly once for each client)

The UML diagram:


Printer
Parser
Person
Orders
PersonLogic
OrderSum
OrderLogic
Product
ProductLogic
OrderValidator
ProductValidator
«interface»
Validator
PersonDAO
OrderDAO
OrderSumDAO
ProductDAO
Connection
AbstractDAO

4. Implementation

1) The Model Package

In this package, there are 4 classes: the Person class, the Product, the Orders and the OrderSum class. Each of these classes correspond to a table in the database and each Object of one of these Classes has as a unique identifier the id, except for the latter one whose unique identifier is the personId. Other fields are:

For Person class: the name, the address

For Product class: the name, the quantity, the price

For order class: the personId, the productid, the quantity; the personId corresponds to the person that made an order, and the ProductId to the product that was ordered

For OrderSum: the personId, the totalPrice; the personId corresponds to a person that has to pay totalPrice amount of money for all of his/hers commands.

Talking about methods, each class has mutators and accessors for their fileds.

2) The DataAccessLayer

Contains 5 classes: the AbstractDAO class, PersonDAO, ProductDAO, orderDAO, OrderSumDAO.

The AbstractDAO class is very important: here the queries used by all of the classes in order to perform operations in relation to database are created by the following methods: createSelectQuery() for reading, createInsertQuery() for creating, createDeleteQuery() for deleting and createUpdateQuery() for updating. Each of these methods return a string that corresponds to an SQL query that when executed, inserts/selects/deletes/updates a table from a database. Also, 4 generic methods corresponding to the CRUD operations are developed: the findById() method, that returns an Object with the id that was searched, the findAll() that returns all Objects form a table in the database, the delete() that deletes a certain object whose id is equal to the given parameter, and the update() method that has as parameter the field that needs to be updated and the new value that the field will have. All these methods contain a Statement Object that, when executed(with the methods executeQuery() or executeUpdate()) returns, either a ResultSet object, in the case that the executeQuery() method is called, or number of rows affected by the executeUpdate() method. In the case in which a ResultSet object is returned, we need a method that convers objects from a ResultSet to an Object wanted by us (one corresponding to a database table, from the Model package) so, that is exactly what the createObjects() method does.

The PersonDAO, ProductDAO classes contain special operations such as findByName() that return an object of type Person/Product if there exists one with the name equal to the given parameter. These methods are used when deleting or adding a new Person/Product to the table

The OrderDAO class contains methods such as findAllByPersonID(), used to generate a pdf file with all orders from a client or methods of deleting all orders with a certain productid or personID, used when we are deleting a person or a product(it does not make sense to have orders from persons or orders of products that do not exist).

The OrderSumDAO class contains methods for finding an object with a certain personID, used for getting the total amount that a Person has to pay or for deleting an object with a certain personID, used when we delete a person(if the person does not exist, neither should the total sum that he/she has to pay).

3) The BusinessLogicLayer

Has another package within it: Validate that contains an interface (Validator) whose method (validate) is implemented by the two classes ProductValidate and OrderValidate that throw an exception if the Order or Product object is not valid (the quantity or price <= 0)

Other than that, it contains 3 more classes:

The PersonLogic class – contains two methods. The first method is used for inserting a person and for deleting a person. In the inserting method, first, a Person with the same name is searched with the findByNmae method from the PersonDAO class (we can not add two persons with the same name). If it does not exist, the person is inserted. In the deleting method, we search for a person to see if the person we want to delete exists. If it exists, we delete him/her and all the Orders and the OrderSum object corresponding to them (with the same personID).

The ProductLogic class is very similar to the PersonLogic class, and has two methods also for inserting and deleting that work on the same principle.

The OrderLogic class has a single method that is for inserting a new order and it works the following way: it searched for the Person that made the order and the ordered Product by their ids(the fields productid, personid). If they exist, it compares the orders quantity with the stock quantity. If the stock is not exceeded, the order is made, and the stock is updated (subtracting the orders quantity). Also, the amount that the person that made the order has to pay is updated. Then, a pdf file with the person's commands and amount to pay is generated. If the stock is exceeded, a message informing the Person that the stock is exceeded is given into a pdf file.

4) The PresentationLayer- contains two classes: Printer and Parser

The Printer class contains methods for generating pdf files such as: all orders of a Person, an understock message. This methods are used in the OrderLogic class when inserting a new order. Other methods are the generic method printAll() that is used to print in a tabular form all the Product/Persons from the database table. It also generates a pdf file and the printAllOrders() method used to print all orders fronm the orders table. Given the fact that the order contains productid and personId as fields, this method prints for all the objects the Person's and the Product's name instead of their ids, in that way being more clear. The names are searched with methods from the classes in the DataAccessLayer package.

The methods for printing all objects from a table into a pdf file give a name to the pdf file in the following manner:

ObjectsName + exact TimeStamp + ".pdf"- by using the exact TimeStamp we can differentiate between multiple pdf files generated of the same table by the time they were generated.

When generating a pdf file for all orders of a person and the totalPrice that he/she has to pay, the pdf will be named:

Receipt + the person's name + ".pdf". In that way, even if the method is called 20 times for a person, at the end there will be only one pdf file generated for a person containing all the orders corresponding to him/her.

The parser class reads from a file that contains the commands. It contains two methods, one by reading the file line by line (1 line=1 command), then it prepares that line(gets rid of the special characters such as ",:" , gets rid of multiple blank spaces and then splits the line into a list of words). The words list is then given as a parameter to the other method: parse(). The parse() method is very important because this is where the decoding of the command is given and actually where the operation will be performed, by calling methods from the BussinessLogicLayer's classes.

The valid commands are:

- Insert client: name, address
- Delete client: name
- Insert product: name, quantity, price
- Delete product: name
- Order: namePerson, nameProduct, quantity
- Report client
- Report order
- Report product

Depending on the first two (or one in the case of Report) words the parser identifies the command and then calls for the required method from PersonLogic, OrdeerLogic or ProductLogic (depending on the object on which the operation is to be performed; this object is usually identified by reading the second word) to execute the operation that was identified by reading the first word(insert, delete, report).

5. Results

In terms of results, a commands.txt file was given with the following contents:

Insert client: Ion Popescu, Bucuresti

Insert client: Luca George, Bucuresti

Report client

Insert client: Sandu Vasile, Cluj-Napoca

Report client

Delete client: Ion Popescu, Bucuresti

Report client

Insert product: apple, 20, 1

Insert product: peach, 50, 2

Insert product: apple, 20, 1

Report product

Delete Product: peach

Insert product: orange, 40, 1.5

Insert product: lemon, 70, 2

Report product

Order: Luca George, apple, 5

Order: Luca George, lemon, 5

Order: Sandu Vasile, apple, 100

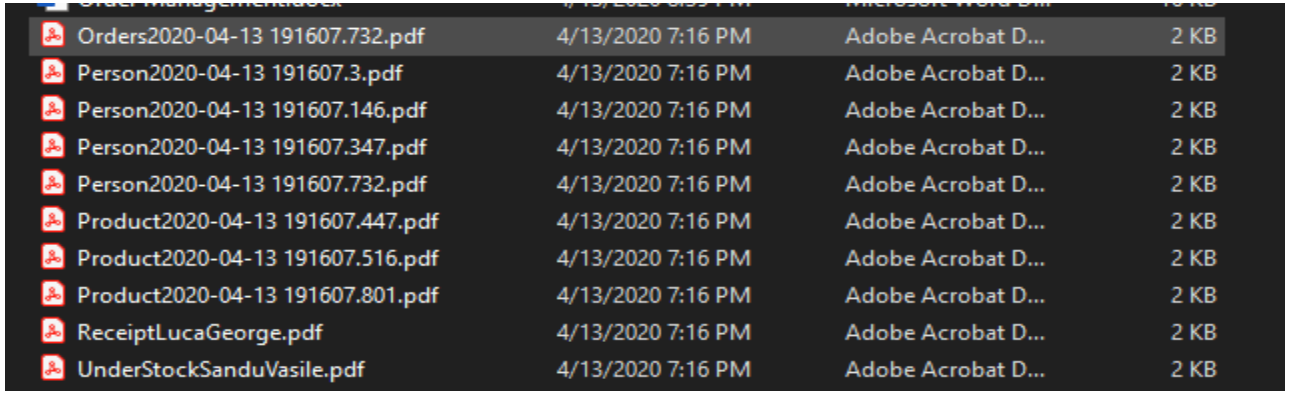
Report client

Report order

Report product

We can notice that there were intentionally placed multiple spaces in order to present the full functionality of the parser to be presented(it works no matter how many spaces are between words).

The pdf files generated are attached to the project:



6. Conclusions

This assignment develops an application that simulates a warehouse-like product management. It can be further developed by adding new features such as: generating a user friendly UI for the client to be able to select more easily the products and to place an order. Another further development would be the addition of more tables and more commands.

7. Bibliography

Database_Connection_Java.pdf

https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-layered-architecture.git

https://utcn_dsrl@bitbucket.org/utcn_dsrl/pt-reflection-example/src/master/