

**Assignment 3**

**ORDER MANAGEMENT**

**Faculty:** Automation and Computer Science

**Department:** Computer Science

**Student:** Cofaru Vlad-George , **Group** 30423/1

1. **Objective**

The objective of this assignment is to design and implement a system for **Order Management**  capable of processing customers orders to a warehouse of products .

The system is going to use a Relational **database** in order to store the defining data of the customer , a report of all the orders that have been placed in the warehouse , a catalogue of the products that are currently in stock and also a table that is used to track how much did each client spend.

The program is also able **to generate in a .pdf** a format a report of the current situation in each of the tables , representing all the active customers , all the orders that have been placed or the products available in the warehouse with currently updated stocks . Further more for each order the system will generate a bill describing that certain order , or in the case an order can not be processed , an error message is generated.

The system will begin by reading from a file a sequence of commands that will be later interpreted by the program .

In order to document the clasSes of the program , a Javadoc file was also created. Javadoc is a documentation generator created by Sun Microsystems for the Java language for generating API documentation in HTML format from Java source code. The HTML format is used for adding the convenience of being able to hyperlink related documents together

Furthermore , the project also had to presented in a “ .jar “ format . A JAR file allows Java runtimes to efficiently deploy an entire application, including its classes and their associated resources, in a single request. JAR file elements may be compressed, shortening download times.

In order to achieve this goal , an object oriented programming approach was taken ,applying a **three tier architecture** model to the classes of our project . In order to reduce to a minimum the amount of code needed to be written a **Reflexive technique** was applied , technique witch allowed us to write a generic class that contains the common functionality that each of the model classes should have , in this way the model classes themselves will have only specific functions and queries . More details will be presented in fourth chapter of the documentation , **Implementation** .

1. **Problem Analysis, Modelling, Scenarios, Use Cases**

The problem requires that the developer designs and implement an **order management system** for a warehouse , capable of storying information about the clients , the products that are currently in the warehouse and the orders that have been placed .

In order to model this real life scenario , a database was used to store the needed information . More precisely a **Client** table was crated containing the data of the customer , the name of that person , and its address ( in order to simply the project only the city was taken into account) . A **Products** table was used to model the warehouse , containing a list of products that are currently available , the quantity of that certain product and the price per unit . An **Orders** table was created to manage and store the orders that each client has placed , containing the name of the customers that placed the order , the product that is ordered and the quantity that is wanted . In order to keep a better management of the orders each client has placed , an additional table was created , **TotalOrders** witch keeps track of how much each client has spend .

As an additional functionality, the program is able to generate pfd reporst of the current state of each of the tables mentioned above , further more for each successful order a pdf Bill is generated containing the name of the buyer and the list of products that the client has ordered . In case an order can not be placed , an error message describing the order that caused the problem will be generated

Regarding the structure of the program , a **Three lair architecture** was used in order to manage the distribution of the classes in packages. The first layer consists of the **Presentation Layer** which defines the user interface ( but in our case since a GUI was not implemented it represents the parser of the input file ) . The next layer is the **Business Logic Layer** witch contains the classes that encapsulate the application logic. And finally a **Data Access Layer** responsible for all establishing the connection with the database and all further interactions with the tables.

In creating the Data Access Layer , a powerful programing technique was used in order to reduce the amount of repetitive code that needs to be written , namely **Reflexive technique .**  This technique allowed us to created a generic class and write only once the generic queries that all the classes will need .

The input of the program will come from a text file containing a set of instructions that will be executed in a sequential order. The program is able to understand the following sets of actions

* Insert client : Name , City

--- witch will create a new client and will insert it in the database .

* Insert Product : Name, Quantity , price

--- witch will create a new product and will insert it in the database with the mention that is the product already exists , its quantity will be updated and no new product will be inserted .

* Delete client:Name

**---** this is the instruction that will delete the customer with the name

“ Name ” from the database .

* Delete product: ProductName

**---** this is the instruction that will delete the product with the name

“ ProductName ” from the database .

* Order: ClientName, ProductName, quantity

--- this will create an order for the the client “ ClientName” , for the product “ ProductName ” with the quantity “ Quantity”

* Report client

---This will generate a pdf report of the current situation of the table Client from the database

* Report products
* ---This will generate a pdf report of the current situation of the table Products from the database
* Report Orders

---This will generate a pdf report of the current situation of the table Orders from the database

From the user perspective, the problem is quite simple: provide and input file with the parameters mentioned above and the program will interpret the command generating the required output pdf files . When running the jar file configuration the calling of the program is the same

For example, if the user wishes to insert a new client Ion Popescu from Bucuresti in the database , to remove George Vasil form Cluj , to add 20 apples with the price of 1 in the warehouse and to place an order for 10 apples in the name of Ion Popescu the input file should look like this :

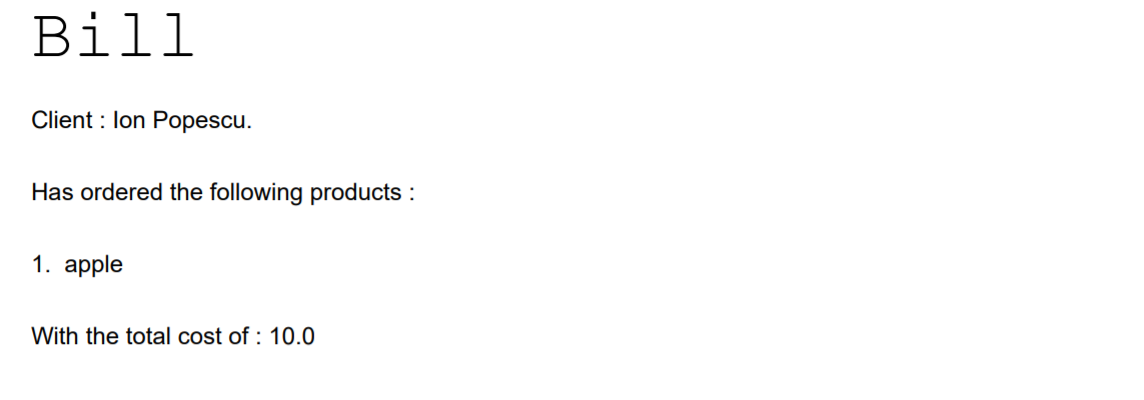
Insert client: Ion Popescu, Bucuresti *The client was added*

Delete client: George Vasil, Cluj *The client was removed*

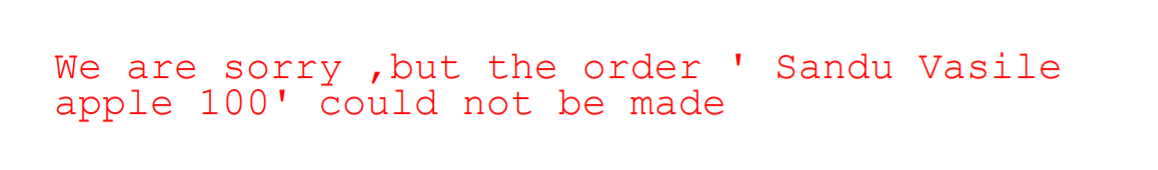
Insert product: apple , 20 , 1 *The products was added*

Order: Ion Popescu , apple, 10 *The order was placed ,*

If the order was successful the user will see a pdf file of the form Bill\_Ion Popescu.pdf containing the details of the order .



In the case an order could not be processed a pdf file of the form Bill-ClientName-error.pdf will be generated. In the case of Sandu Vasile who tried to order 100 apples (because of not enough available stock ) the file Bill\_Sandu Vasile\_error.pdf was generated containing :



1. **Design**

The object-oriented programming design was exploited. Certain key aspects of the object-oriented design and implementation elements were used in this assignment, such as encapsulation, generics and composition .

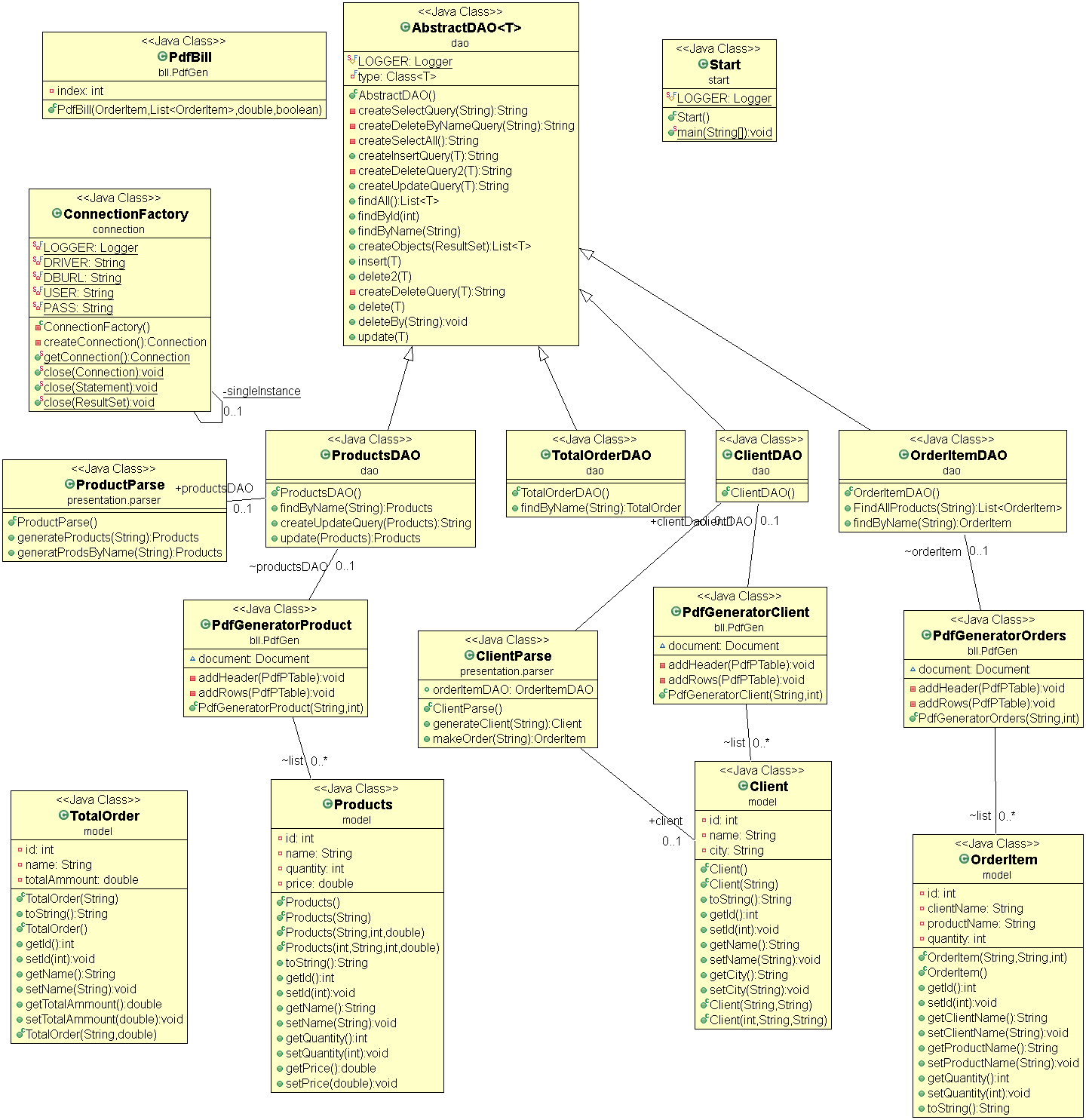
In designing this project the capabilities of a **Reflexive technique** were used . In computer science, reflection is the ability of a process to examine, introspect, and modify its own structure and modify the behavior of methods, classes, interfaces at runtime. This technique is especially powerful where more classes should be able to perform the same operations , but the structure of these classes is not the same , thus through the use of a reflection we reduce the amount of repetitive code we need to write .

This technique was mostly exploited in the **Database Access Classes** ( DAO ). This class being responsible for performing all the operations regarding the manipulation of data from the databases mentioned above . Thus creating an **AbstractDao < T >** generic class , all the common functions of the model classes , Crud ( create , read , update , delete ) and the find methods , were written only once , instead of having to write it once for each model class .

The solution was designed using a **Three layer architecture**  having the project divided in the following packages

* **Bll.pdfGen**  , represents the business logic level of the project , having the utility of generating certain pdf reports when required by the control unit .
* **Dao( data access classes)** is responsible for all the interactions of the program with the database
* **Connection**  is the class responsible for establishing a connection with the database
* **CotntrolParser**  this is the class which plays the role of the controller , while also containing the classes necessary for achieving the parsing of the input file

The UML diagram below illustrates the relationships between the classes. It was generated automatically using the ObjectAid UML Explorer for Eclipse.

****

1. **Implementation**

In order for everything to be easier to follow , the classes will be presented in groups , based on their functionality .

* **Model classes**
* **Client**

The purpose of this class is to model a real life client . Since our program implements in order management system , the client was characterized only by a simple set of identifiers , in our case , an *id* which is useful when thinking of the client as an entry in the database , id thus being an unique identifier . A *name* which helps us to describe the client , and a *City*  which helps us to know where to sent a certain order to.

In terms of methods , this class is pretty straight forward , meaning that it contains only **getters and setters** for each field , to provide a way of accessing the fields without working directly with them , and the constructors .

* **Products**

The purpose of this class is to model the products witch are currently in the warehouse . In terms of fields , this class contains an *id*  which is used to uniquely identify items in the database. The *name* of the product witch has the role of identifying the product . The *quantity* witch shows the current amount of that product that is currently in the database , and the *price* which in our case represents the price per a single unit of the product

In terms of methods , this class is pretty straight forward , meaning that it contains only **getters and setters** for each field , to provide a way of accessing the fields without working directly with them , and the constructors .

* **OrderItem**

The purpose of this class is to model the orders that a client can place . In order to achieve this goal the class has , an *id* field which is used to uniquely identify the order in the database. A *clientName*  having the role of describing witch client in the database wanted to place an order , a *productName*  which shows what product from the database the client wanted to buy , and a *quantity* showing the amount that the clients wishes to buy of a certain product .

In terms of methods , this class is pretty straight forward , meaning that it contains only **getters and setters** for each field , to provide a way of accessing the fields without working directly with them , and the constructors .

* **TotalOrder**

This class was used to track how much each customer had spend from all its orders , to help give a better view of how much money was earned , and how much each client payed . To achieve this the class has an *id* field which is used to uniquely identify the Total order in the database , a *name* field representing the client and a *totalAmmount* representing the total amount that the client had spend .

In terms of methods , this class is pretty straight forward , meaning that it contains only **getters and setters** for each field , to provide a way of accessing the fields without working directly with them , and the constructors .

* **Connection**

The purpose of this class is to establish a connection with the database in order to be able to apply the required queries by the program . Regarding the methods this class is capable of creating a connection with the method  *createConnection()* and is also able to close it when all the operations are done using close(Connection connection)

* **DAO**

This class is responsible for all the interactions with the database . In order to reduce the amount of code written a reflexive technique was used when creating the generic class **AbstractDao<T> ,** witch allowed us to write the common code for the CRUD operations and find only once **.**

Regarding the methods of this class , they can be divided in to types. Ones the prepare the execution of an instruction by creating the query that our database will use , These methods are used to create the SQL language specific instruction , and with the help of the reflexive technique , at runtime the specific details of each class is introduced in the instruction . The methods that are a part of this category are : ***createSelectQuery(String field), createDeleteByNameQuery(String field), createSelectAll() , createInsertQuery(T t) , createUpdateQuery(T t)***

The second category of methods from this class are the ones that actualy execture the query generated above and return a result . Those class follow a generic structure , a connection to the database is establish , a proper query is created using the methods specified above and it is then executed . The function then will with the help of the ***createObject***  method will return an object in the case of : ***findById(int id) , findByName(String name), insert(T t),delete(T t) and update(T t)*  ,** or a list of objects in the case of ***findALL .***

Because of the using of the reflexive technique all that our classes had to do in order to use the generic methods was to extend the **AbstractClass** mentioned above, so in this manner we created such a class for each our model classes : **ClientDAO , ProductDAO, OrderItemDAO, TotalOrderDAO.** In these classes , if needed only specific queries had to be written .

The only class were a separate query had to be written was ***FindAllProducts(String name)***  from ***OrderItem*** table witch had the role to return all the products that a certain client ordered.

NOTE : due to some errors in the sql language same generic queries had to be rewritten in their respective DAO clases , adding the ltrim() sql function to the instruction .

* **PdfGen**

This class has the role of generating the pdf reports needed by the application .

* **PdfGeneratorClient , PdfGeneratorClient , PdfGeneratorClient**

These classes are very similar , thus they have been grouped together . They have the role of generating pdf reports of the current situation of the tables in the database . They receive as an input the name of the file where the reports will be generated and an integer representing the column number of each table. The method calls first of all a **findAll**  to get the list of elements that will be displayed and then in creates the table with that information in a pdf format .

* **PdfBill**

This class has the role of generating the pfd bill that the clients will receive after placing an order . This method receives as parameters an orderItem , a list of profucts that the client had already ordered , a total price and a Successful variable . In case Successful is true the bill will display the clients name , the products he orderd and the total price in a pdf file of the form *ClientsName.pdf.*  In the case Successful is false , the method will generate an error message instead of the bill describing the problem , in a pdf file named *ClientsName\_error.pdf.*

* **ControlParser**

This package contains the classes responsible for parsing the input file , and the class FileParser witch acts as a controller in this program .

* **ClientParser**

This class is an utility class responsible for generating a client by parsing the input received as a string .

For this purpose the method **generateClient** was created, which received as an input a string , it parses it and generates a clients with the same data as the string .

The other method used by this class is **makeOrder**  which received as an input a string , it parses it and generates an Order with the same data as the string

* **ProductsParser**

This class is responsible for receiving a string input with the information about a product , parsing that information and generating a product object with the same data as the input string

* **FileParser**

This is the class that receives the input file , goes through it line by line and executed each instruction. This class consists of two methods .

The method ***executeInstruction(String line).*** This functios recives as a parameter an input string representing the line which was read from the file . First of all the line is split in two parts *instruction* and *data. ( e.g* in the line “Insert client: Ion Popescu, Bucuresti “ the *instruction*  field will have “ Insert client” and the *data* will contain Ion Popescu, Bucuresti, the actual information about the client ) . Then the data information is send to the specific parser mentioned above in each case to generate an object . Then depending on the instruction the program will perform a certain set of actions . For the commands that are more simpler like insert or delete , the program will simply call the required method from the abstract dao class.

But there are some special cases that had to be treated separately

The **insert product** command first of all creates a new product with the help of *producParser* , then it checks if the product already exists in the database or not. In the case it is found , its quantity is incremented with the quantity of the object we are trying to insert , the products in NOT inserted again . In the case the product is not already in the database , a new product is added .

The **Report**  command calls the required PdfGen in order to create reports of the databases.

The **Order**  command is responsible for the case in witch a client wats to place an order . First of all in creates an object of type OrderItem. It checks if the products that the client wants to buy is in the database . If the product it is found the program continues by checking if the quantity that the client wants to buys is smaller that the quantity that is in the warehouse at that moment , if this is the case , the order it is seen as successful , so it will be processed by decrementing the quantity that is in the warehouse and by generating a pdf bill , detailing the order. If the product is not in the database or if the quantity wished to be bough is bigger that the quantity in stock , a pdf file describing the error will be generated .

Finally the method **ReadData** is responsible for reding line by line from the input file and calling the execution function described above.

* **STAR**

This class represents the main class .

1. **Results**

The result is a program, witch is able to simulate an order management system for warehouse.

The Program uses a database for the storage of the information, being able to interact with the database and its contents . The commands of the program are given in the form of an input file the program latter parsing the file to understand each instruction. Above managing the orders , the program is also able to generate pdf reports about the situation in each database . After each successful order a bill will be generated, or in the case an order fails , an error message will be created instead.

Besides the documentation itself the program also comes with a javaDoc document to help the understanding of its structure ,

The program can simply be run from the program itself , or using the .jar configuration , specifying only the input file as an argument :

java -jar PT2020\_30423\_Vlad-George\_Cofaru\_Assignment\_3.jar commands.txt

1. **Conclusion**

To complete with , this project helped me further develop my java oriented programing style , giving me a glance at what the reflexive technique is and its capabilities . The project was also an exercise towards forming a better style of structuring code , following models like the Three layer architecture

1. **Bibliography s**

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

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

<https://www.geeksforgeeks.org/reflection-in-java/>

<https://en.wikipedia.org/wiki/Reflexive_technique>

<https://www.microsoft.com/en-us/sql-server/sql-server-2019>

<https://www.w3schools.com/sql/sql.asp>