TECHNICAL
UNIVERSITY
OF CLUJ-NAPOCA
ROMANIA

Assignment 3
Order Management

Faculty: Automation and Computer Science

Department: Computer Science

Student: Dragan Iulia-Andreea, **Group** 30423/1

1. Objective

The objective of this assignment is to design and implement an application **OrderManagement** for processing customer orders for a warehouse. **Relational databases** are used to store the products, the clients and the orders. The application will be structured in packages using a **layered architecture** and will contain at least the following classes:

- **Model classes** – data models of the application
- **Business Logic classes** – implement the application logic
- **Presentation classes** – implement the user input / output
- **Data access classes** – implement the access to the database

The application will allow processing commands from a **text file received as command line argument** and perform the requested operations. The data will be saved in the database and the program will generate **reports in PDF format**.

In order to successfully achieve the requested goals, an **objected-oriented programming design**, as well as relational databases were used, through the **MySQL** platform. Thus, the following things were ensured:

- Creating the methods necessary to acquire and correctly process the data from the **file given as input**, so that for each command present in the file the database will be updated / a **PDF file** containing the information from the database will be generated.
- Implementing **model / business logic / data access classes** to manipulate the data that was read from the file to efficiently place it inside the database, as well as to manipulate the data extracted from the database and process it accordingly. In order to achieve this, **reflection** was used, in order to abstract the **SQL commands** and methods necessary to retrieve / write data from / into the database.
- Creating the classes and the methods necessary to generate the **output files**, by acquiring the data from the tables and displaying it in the form of tables, for legibility. This was done by using the **itext library**.

2. Problem Analysis, Modelling, Scenarios, Use Cases

The problem requires that the developer successfully designs and implements an **order management simulator** using relational database, to illustrate retrieving data from tables and introducing it respectively.

From the perspective of the user, the program can be used by **running the .jar file** generated by the developer in the command line. The program will interpret the data from the file and it will act accordingly, by updating the database step by step and by generating a PDF for each report command, as well as **bills** containing information concerning all the orders which were processed / which were failed to be processed in the case in which the required conditions were not satisfied.

In terms of **scenarios and use-cases**, the program behaves as follows:

- When running the **.jar file**, the user also has to write one argument, which will represent **the input text file**. If the argument is not given to the program, it will generate exceptions and the execution will stop; the command is:

```
java -jar program_name.jar input_file.txt
```

- The **output files** will appear under the form of **PDF files**, with the name **“Table_NameTableTimex”**, where x is a variable representing the current **timestamp** of the program in order to ensure that the names of the files will not repeat after multiple executions of the program. The files will be overwritten in case of multiple runs, but they will not overwrite each other during a single run.

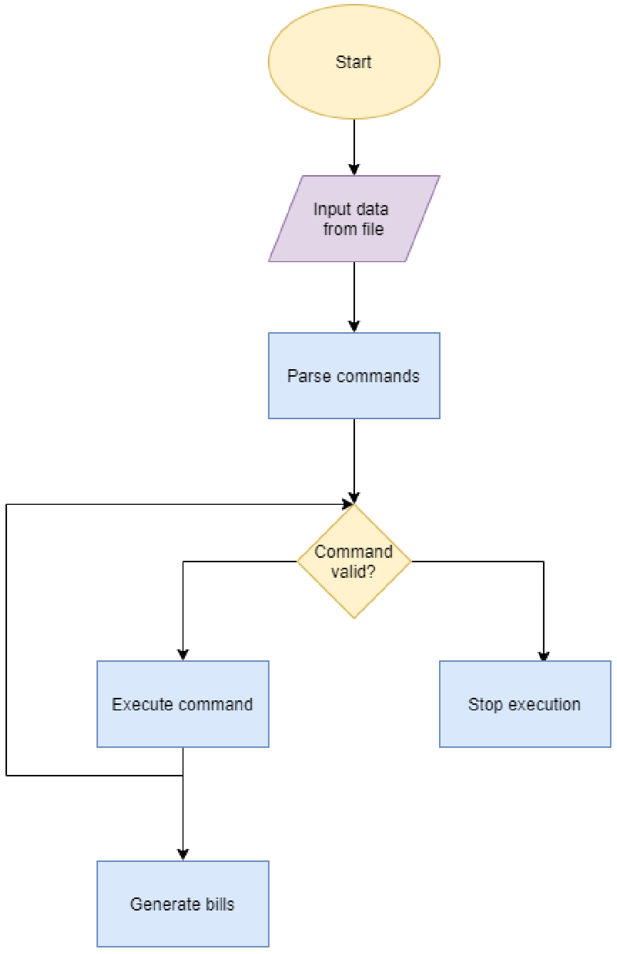
Note about the SQL dump files: The tables in the database come populated with the data from the sample input files. Only the insert operations have been performed, no deletion or update. There is also an empty database.

Please note that the program was implemented in such a way that each time it is run, the **database can be reset at the beginning of the execution** in order to ensure that the generated outputs are strictly based on the commands given as inputs in the file. This is purely to illustrate the functionality of the program on a **predefined set of inputs**. To do that, input the following command when running the program:

```
java -jar program_name.jar input_file.txt reset
```

Since it is considered that **each product ordered by a client belongs to the same order**, the program will generate **one bill / client**, containing all the orders they have placed and the total price. If all orders have been processed successfully, a success message is generated. Otherwise, the program will generate a notification containing the details of the unsuccessful order inside the PDF.

The **flowchart** below briefly illustrates the overall functionality of the program:



3. Design

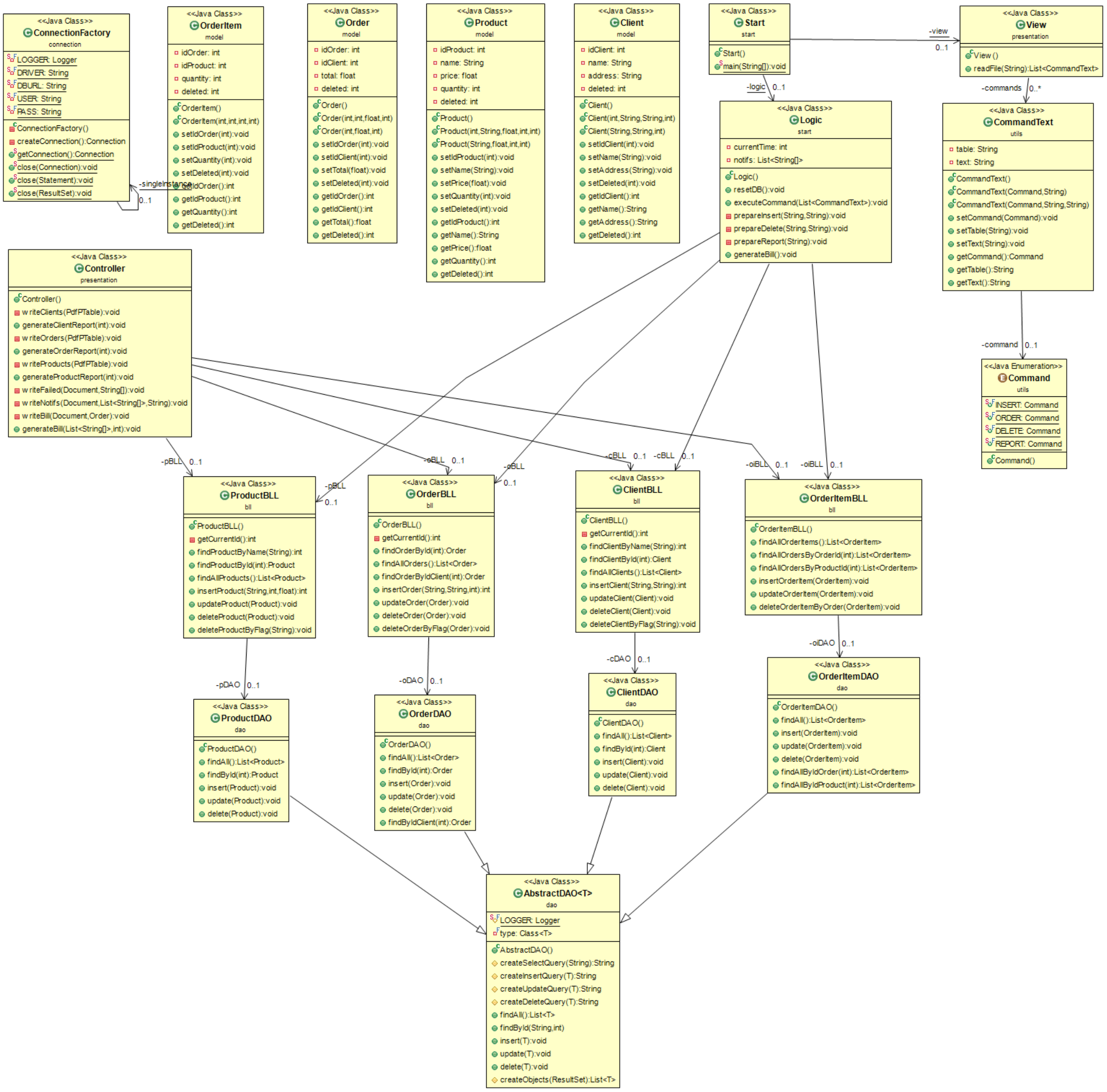
Important!! The layered and reflection **templates** respectively displayed in the support document **Assignment_3_Indications.pdf** and on the **UTCNdsrl bitbucket repository** were used as shell for the program, with slight modifications to the given pieces of code.

The **object-oriented programming design** was exploited as well. Certain key object-oriented design and implementation elements were used in this assignment, such as **encapsulation**, **inheritance** and **generics** (when implementing the data access class through reflection).

In terms of the structure of the project, as required in the objective, the following **packages** were created:

- **BLL (Business Logic Layer)** – contains the classes which implement the **application logic**. It is one layer above the classes within the DAO package, and it contains the classes and methods use to manipulate data from the database. This way, multiple insertions of the same object are avoided, as well as the retrieval of data when the table at hand is empty, so that no **NullPointerExceptions** are generated. This will be elaborated further in the implementation section of the documentation.
- **Connection** – has a single functionality, and that is to **establish the connection between the Java program and the database**.
- **DAO (Data Access)** – contains classes which implement **the access to the database**. They generate the **SQL commands** to retrieve / manipulate data from the database (**SELECT, INSERT, UPDATE, DELETE**). All the subclasses extend the superclass **AbstractDAO<T>**, which was implemented by using the **reflection** feature of the Java language.
- **Model** – contains classes representing **data models** of the application. They are the objects (tables) contained in the database and mostly consist of setter and getter methods, which are then used to access the data from the relational database system.
- **Presentation** – is made of classes which implement the **relationship between user input / output files and the program**. One of the classes has the purpose to generate the list of commands received in the file, while the other generates the output **PDF files**.
- **Start** – contains the **main class** and a class which creates **a link between the main method and the bll package**, by calling the appropriate methods from it, depending on the current command.
- **Utils** – contains an **enumeration** and a **utility class**, whose purpose is to parse the instructions more easily and to make them easier to access.

The **UML diagram** below illustrates the relationships between the classes. It was generated automatically using the **ObjectAid UML Explorer** for Eclipse. (see next page)


<<Java Class>>
ConnectionFactory
connection
LOGGER: Logger
DRIVER: String
DBURL: String
USER: String
PASS: String
ConnectionFactory()
createConnection():Connection
getConnection():Connection
close(Connection):void
close(Statement):void
close(ResultSet):void
-singleInstance
0..1
<<Java Class>>
OrderItem
model
idOrder: int
idProduct: int
quantity: int
deleted: int
OrderItem()
OrderItem(int,int,int,int)
setIdOrder(int):void
setIdProduct(int):void
setQuantity(int):void
setDeleted(int):void
getIdOrder():int
getIdProduct():int
getQuantity():int
getDeleted():int
<<Java Class>>
Order
model
idOrder: int
idClient: int
total: float
deleted: int
Order()
Order(int,int,float,int)
Order(int,float,int)
setIdOrder(int):void
setIdClient(int):void
setTotal(float):void
setDeleted(int):void
getIdOrder():int
getIdClient():int
getTotal():float
getDeleted():int
<<Java Class>>
Product
model
idProduct: int
name: String
price: float
quantity: int
deleted: int
Product()
Product(int,String,float,int,int)
Product(String,float,int,int)
setIdProduct(int):void
setName(String):void
setPrice(float):void
setQuantity(int):void
setDeleted(int):void
getIdProduct():int
getName():String
getPrice():float
getQuantity():int
getDeleted():int
<<Java Class>>
Client
model
idClient: int
name: String
address: String
deleted: int
Client()
Client(int,String,String,int)
Client(String,String,int)
setIdClient(int):void
setName(String):void
setAddress(String):void
setDeleted(int):void
getIdClient():int
getName():String
getAddress():String
getDeleted():int
<<Java Class>>
Start
start
Start()
main(String[]):void
-view
0..1
<<Java Class>>
View
presentation
View ()
readFile(String):List<CommandText>
-logic
0..1
<<Java Class>>
Logic
start
currentTime: int
notifs: List<String[]>
Logic()
resetDB():void
executeCommand(List<CommandText>):void
prepareInsert(String,String):void
prepareDelete(String,String):void
prepareReport(String):void
generateBill():void
-commands
0..*
<<Java Class>>
CommandText
utils
table: String
text: String
CommandText()
CommandText(Command,String)
CommandText(Command,String,String)
setCommand(Command):void
setTable(String):void
setText(String):void
getCommand():Command
getTable():String
getText():String
-command
0..1
<<Java Enumeration>>
Command
utils
INSERT: Command
ORDER: Command
DELETE: Command
REPORT: Command
Command()
<<Java Class>>
Controller
presentation
Controller()
writeClients(PdfPTable):void
generateClientReport(int):void
writeOrders(PdfPTable):void
generateOrderReport(int):void
writeProducts(PdfPTable):void
generateProductReport(int):void
writeFailed(Document,String[]):void
writeNotifs(Document,List<String[]>,String):void
writeBill(Document,Order):void
generateBill(List<String[]>,int):void
-pBLL
0..1
-pBLL
0..1
-oBLL
0..1
-oBLL
0..1
-cBLL
0..1
-cBLL
0..1
-oiBLL
0..1
-oiBLL
0..1
<<Java Class>>
ProductBLL
bll
ProductBLL()
getCurrentId():int
findProductByName(String):int
findProductById(int):Product
findAllProducts():List<Product>
insertProduct(String,int,float):int
updateProduct(Product):void
deleteProduct(Product):void
deleteProductByFlag(String):void
<<Java Class>>
OrderBLL
bll
OrderBLL()
getCurrentId():int
findOrderById(int):Order
findAllOrders():List<Order>
findOrderByIdClient(int):Order
insertOrder(String,String,int):int
updateOrder(Order):void
deleteOrder(Order):void
deleteOrderByFlag(Order):void
<<Java Class>>
ClientBLL
bll
ClientBLL()
getCurrentId():int
findClientByName(String):int
findClientById(int):Client
findAllClients():List<Client>
insertClient(String,String):int
updateClient(Client):void
deleteClient(Client):void
deleteClientByFlag(String):void
<<Java Class>>
OrderItemBLL
bll
OrderItemBLL()
findAllOrderItems():List<OrderItem>
findAllOrdersByOrderId(int):List<OrderItem>
findAllOrdersByProductId(int):List<OrderItem>
insertOrderItem(OrderItem):void
updateOrderItem(OrderItem):void
deleteOrderItemByOrder(OrderItem):void
-pDAO
0..1
-oDAO
0..1
-cDAO
0..1
-oiDAO
0..1
<<Java Class>>
ProductDAO
dao
ProductDAO()
findAll():List<Product>
findById(int):Product
insert(Product):void
update(Product):void
delete(Product):void
<<Java Class>>
OrderDAO
dao
OrderDAO()
findAll():List<Order>
findById(int):Order
insert(Order):void
update(Order):void
delete(Order):void
findByIdClient(int):Order
<<Java Class>>
ClientDAO
dao
ClientDAO()
findAll():List<Client>
findById(int):Client
insert(Client):void
update(Client):void
delete(Client):void
<<Java Class>>
OrderItemDAO
dao
OrderItemDAO()
findAll():List<OrderItem>
insert(OrderItem):void
update(OrderItem):void
delete(OrderItem):void
findAllByIdOrder(int):List<OrderItem>
findAllByIdProduct(int):List<OrderItem>
<<Java Class>>
AbstractDAO<T>
dao
LOGGER: Logger
type: Class<T>
AbstractDAO()
createSelectQuery(String):String
createInsertQuery(T):String
createUpdateQuery(T):String
createDeleteQuery(T):String
findAll():List<T>
findById(String,int)
insert(T):void
update(T):void
delete(T):void
createObjects(ResultSet):List<T>

In terms of the user interaction, the program was designed to accept the following **commands** from the input file:

Insert client: Name, Address – inserts a new client in the database

Insert product: Name, Quantity, Price – inserts a new product in the database

Order: Client_Name, Product_Name, Quantity – creates an order (if possible)

Delete client: Name[, Address] – deletes a client from the database; address is optional

Delete product: Name – deletes a product from the database

Report client – PDF containing information about the clients

Report product – PDF containing information about the products

Report order – PDF containing information about the orders

Please note the following:

- When **inserting a client or a product** into the database, the program will search for **duplicates** (since the name is considered to be unique and the id is incremented automatically, the name will be searched). If a duplicate is found, the program will not generate an exception, but it will **update the current object** with the data received through the insert command.
- **An order is inserted** only if all the requirements are met (client and product exists, and the quantity available in stock is larger or at least equal to the quantity requested in the order). **If the order is not processed**, this information will be stated in the final PDF, acting as the bill, which will generate **notification messages** along with the orders of each client.
- If an **“order” command** offers as inputs an **invalid client / product name**, an **exception** will be thrown and the execution of the program will stop. This is considered to be an invalid command, since the user is trying to access a field that does not exist.
- **The orders are grouped by client**, i.e. each order placed by the same client will be added up to a **single order** (by having the same order id). This way, the **total price** of an order for each client is the sum of all the prices from each order.
- When **a client is deleted**, all the orders associated with them will be deleted as well.
- When **a product is deleted**, all the orders associated with it will be deleted as well.
- **A product with quantity 0 will not be deleted** from the database, as it is still considered existent, but simply out of stock.

The fields are **not fully deleted**. Each table contains an auxiliary column called **“deleted”**, which holds the **value 0 i**n the case in which the item is still in the database, **and 1** in case it is not. This way, the generation of the bill is performed more easily, as the retrieval of data is done by simply **ignoring the “deleted” field**. The field is taken into consideration only when performing operations on the database / generating the reports for each table.

The only time when the information from the database is **truly deleted** is at the beginning of the program, if the **“reset” command** is written as command line argument along with the input file.

The tables within the database itself are **not connected through SQL code**, but **through the java code** from the program. It ensures that no illegal accesses are performed and that the deletion process is done through **cascading** the tables.

4. Implementation

In order for everything to be easier to read and understand, the classes will be presented in **groups**, based on **the package they belong to**. If the classes within a package are too similar, they will not be discussed in detail, only the use of the package will be explained.

For more details, please refer to the **JavaDoc files**.

- **BLL package**
  - **ClientBLL class**
  - **OrderBLL class**
  - **OrderItemBLL class**
  - **ProductBLL class**

These classes are one level above the **DAO classes**, since they encapsulate them and enhance them by adding several conditions before performing the methods within them.

It is ensured that data will no be overwritten by using the **getCurrentId()** method, which retrieves the **largest id present in the database**, so that the insertion is done from there onwards.

Some examples of these would be the **standard find methods**, as well as the **well-defined insert methods**, each specific for each object. Inserting a client or a product is easy, but inserting an order or an orderItem is done by **verifying certain aspects** first (whether the client / the product exists).

The **delete methods** (whether they are true delete or delete by the “delete” flag) ensure that the process will **cascaded** through all the tables which contain a field from the current table, by deleting all its appearances in them.

Please note that **inserting an already existent item** (i.e. its name is already in the database) will lead to the update of the existing item. If a client of the same name is inserted twice, but with different address the address of the client inserted first will be updated. If a product with the same name is inserted twice, its quantity will be increased according to the value in the second insertion.

- **Connection package**
  - **ConnectionFactory class**

Singleton class.

The purpose of this class is to **generate the connection between the program and the database** for each operation performed on the tables of the database, as well as to close the connection once the operation has finished.

It contains methods to **close the connection**, the **resultSet** (when necessary – for the select operations) and the **statement**, for each query.

- ## DAO package
  - **AbstractDAO class**

SuperClass created through reflection and by using the singleton principle.

The **createXQuery()** methods generate the queries corresponding to the operation the user wishes to perform (**select, insert, update, delete**). They each output a string representing they respective query.

findById(String, int) and **findAll()** are self-explanatory. They generate the object / the resultSet corresponding to the id / table to be searched. If no item in the database corresponds to the searched id or if the desired table is empty, the methods return a null.

Insert(T), update(T), delete(T) all act according to their specifications. They do not return anything, but simply modify the information in the database.

createObjects(ResultSet) receives the resultSet as parameter and converts into a list of its corresponding objects.

  - **ClientDAO class**
  - **OrderDAO class**
  - **OrderItemDAO class**
  - **ProductDAO class**

Note: Since **OrderItem does not have its own primary key**, but only foreign keys, this class contains several special methods, **findAllByOrder()** and **findAllByProduct()**, which allow to find all the elements from the database based on one of the foreign keys. This is done since several **OrderItems** can belong to the same order, and several **OrderItems()** can contain the same product.

All these classes extend the **AbstractDAO** class. They contain the standard methods for operating with tables, i.e. **find, insert, update, delete**.

- ## Model package

Contains the following classes:

  - **Client class** – **fields**: idClient, name, address, deleted
  - **Order class** – **fields**: idOrder, idClient, total, deleted
  - **OrderItem class** – **fields**: idOrder, idProduct, quantity, deleted
  - **Product class** – **fields**: idProduct, price, quantity, deleted

Note: The total field inside the Order class represents the **total price** of all the orders placed by the client represented by **idClient**, as stated above.

They act as objects for the tables in the database. Each field in the table is represented by a field in its respective class. Please note the **“deleted” field**, which acts as a flag to determine which rows are considered to be deleted. These classes only contain **setters and getters** for their fields.

- ## Presentation package
    - ### View class

Contains a single method, **readFile(String)** and a list of **CommandTexts** (see the utils package). That list represents all the **commands which have been read** from the input file, in the order in which they were read. The method returns that list, which will be further used in the **Logic class**.

- ### Controller class

The **generateBill()** method generates the **final reports** (bills), containing all the orders which have been performed (**grouped by client**) as well as notifications for the orders which were not processed.

Contains several methods whose purpose is to generate the **PDFs** and the information which is written in them. The **generateXReport()** methods contain the instructions through which the PDF, the table and its header are generated. The **writeX(PdfPTable)** methods are the ones placing the data itself into the tables, by first ensuring that the **"deleted" flag is set to 0**.

All the id fields acting as **foreign keys** (from all the tables) are replaced with their name from their respective table.

Please note that for the Order table, the PDF generated **does not** contain the total amount, but rather the product and the quantity. The total price will only be shown in the **bill PDF**. For this reason, the orders are not grouped by client in this report.

- ## Start package
    - ### Logic class

Contains the methods whose purpose is to **parse the instructions**.

ResetDB() resets the entire database, by emptying all the tables. This method is only called **at the beginning of the execution** of the program and only if the **"reset"** command is entered in the command line, for simulation purposes, such that each result is based on the input file alone.

executeCommand(List<CommandText>) receives the list of commands generated by the **View class** and interprets it based on its fields. Based on the command held in each element of the list, one of the following methods is called: **prepareInsert(String, String)**, **prepareDelete**, **prepareReport**, according to the types of commands which can be read from the input file.

The **prepareX(String, String)** methods choose the appropriate class and method do be performed from the **BLL package**, based on the table which the user wishes to access and the type of operation they wish to perform on it.

generateBill() generates calls for the method of the same name from the **Controller** class.

- ### Start class

Contains the **main method**, which accesses the **View and the Logic classes**, and makes the connection between the data read from the input file and the instructions which are to be performed.

- **Utils package**
  - **Command enum**

Enumeration which contains all the instructions which can be performed: **INSERT, ORDER, DELETE, REPORT**.

  - **CommandText class**

A class containing **three fields: command, table and text**. This class is used when **parsing the commands** read as inputs, in order for the computation of each command to be done more easily, with a few case statements and with legible fields. Only contains **setter and getter methods**.

5. Results

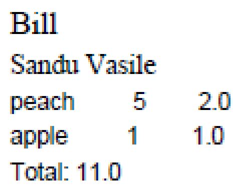
The results are displayed within the **PDFs generated** both by **the report commands and the final bills**. As expected, once **a product / a client is deleted**, the information inside the database itself is deleted as well, both for those objects and for the orders / orderItems.

However, the **bills stay intact**. If an order of a deleted product was requested before the deletion of said product, the generated bill will contain that order. However, when it comes to the **Order table** itself, it can be noticed that once the deletion is performed, so is that respective order.

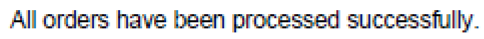
If an order cannot be performed, it is simply **not added to the database**. Despite that, it is still acknowledged by the program and it will generate a bill in which **the failure message** will appear.

In the following example, it can be seen that by placing an order with the product **"peach"**, the end order report will not contain the respective order, but the bill will be generated:

Bill is under the form **(product quantity price)**

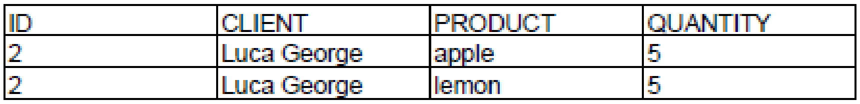


```
Bill
Sandu Vasile
peach    5    2.0
apple    1    1.0
Total: 11.0
```

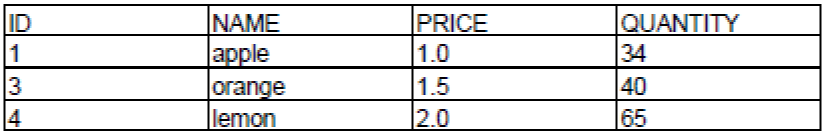


```
All orders have been processed successfully.
```

Order & Product tables after the deletion of the "peach" product:



ID	CLIENT	PRODUCT	QUANTITY
2	Luca George	apple	5
2	Luca George	lemon	5



ID	NAME	PRICE	QUANTITY
1	apple	1.0	34
3	orange	1.5	40
4	lemon	2.0	65

6. Conclusions

In conclusion, the assignment can be viewed as an introduction into communicating with **MySQL databases** using **java programs**, in order for the students to grasp the concept of using databases.

The **layered architecture** was used within the program and it can be noticed through the multitude of packages in which the program was divided. As well as that, certain **reflection techniques** were used to create a generic class (**AbstractDAO**) which contains the methods for accessing the database. The specific queries will be generated automatically by its subclasses.

In order to implement the relational database, **four tables** were created, whose connections and **relationships are established within the java program**.

Object-oriented programming was exploited through the use of **packages**, **classes**, **encapsulation**, **inheritance** and **generics** – through the **AbstractDAO** class.

As **further enhancements**, the tables could be upgraded to display which orders have been fully processed and which orders have been denied inside the database itself. It could also be possible to generate a report PDF containing the history of all the operations performed on the database, and whether they were performed successfully, unsuccessfully, or with slight modifications (in case overwriting occurs).

7. Bibliography

https://www.baeldung.com/javadoc

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

https://api.itextpdf.com/iText5/java/5.5.13/com/itextpdf/text/Paragraph.html

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

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

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

https://bitbucket.org/utcn_dsrl/pt-layered-architecture/src/master/