MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Order Manager

Fundamental Programming Techniques

Horvath Andrea Anett

Faculty of Automation and Computer Science
Computer Science Department
2'nd year, Group 30424

Contents

I. Objectives of the proposed laboratory work

The main objective of the proposed laboratory work was to develop a Java application that allows processing customer orders for a warehouse, together with other maintenance commands like inserting and deleting into and from the database.

As secondary objectives, the following can be considered:

- developing the application in accordance with the object oriented programming principles
- using a layered architecture
- working with relational databases for storing the data for the application (MySQL)
- using reflection techniques to create generic methods
- dynamically generated queries for accessing the database through reflection
- drawing the UML diagrams and writing the code based on them (forward engineering)
- determining all the use-cases
- designing the proper operations for databases to match real life needs
- getting familiar with JavaDoc files
- working with files for reading data
- generating PDF documents for writing data
- running the project as a .jar file

II. Analysis of the laboratory work, modeling, scenarios and use-cases

Analysis

The Order Manager will be implemented to handle actions performed in a warehouse, both for maintenance and for client interaction operations, i.e. ordering.

The development of the application will have as a primary concern the permanent exchange of information between the database and the application. Relational databases will be used to store the products, the clients and the orders. Starting from this, the model that the application will be using is mapped to the fields of the databases. The model will be one of the 5 layers on which the Order Manager will be developed. Having the model of the data, the next layer will implement the necessary queries for the operation to be performed and it will also provide the connection with the MySQL Workbench. The top layer will manage the input and distribute it at the lower layer which will be the one where the following actions are controlled. The input is also validated in a separate layer.

The operation which are to be executed by the Order Manager are:

- add client to the database;
- delete client from the database;
- add product to the database;
- delete product from the database;
- create order for client;
- generate reports.

Modeling

The application will run as a .jar file getting as argument a .txt file containing the commands to be processed. The input text will be parsed, and each command will be executed separately. Before executing, the input is validated and in case of inconsistencies the command will not be executed. Each command is executed using methods generated through reflection. Those methods will be based on SQL queries applied on the database. In order to check the operations, there are commands which generate reports in PDF format.

Concerning the ideas on which the database was developed the following should be highlighted about the tables which were implemented:

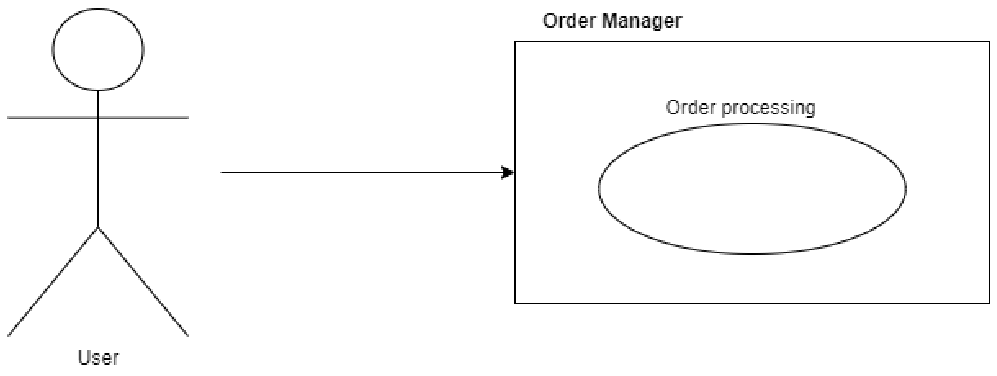
1. Table *client*
    - contains *clientId*: int, *clientName*: varchar and *address*: varchar as fields;
    - the *clientId* is the primary key;
    - the *clientName* is marked as unique, thus there cannot be two clients with the same name (this decision was made since the requirement asks for an order to be made only knowing the name of the client, so two clients with the same name could not be distinguished when making an order).
2. Table *product*
    - contains *productId*: int, *productName*: varchar, *stock*: int and *price*: float;
    - the *productId* is the primary key.
3. Table *ordert*

- contains *orderId*: int, *clientId*: int, *total*: float;
- the *orderId* is the primary key;
- the *clientId* is bound to the *client* table's column *clientId* by a foreign key constraint; when a client is deleted, all the corresponding orders for that client are removed.

4. Table *orderedProduct*
   - contains *orderedProductId*: int, *productId*: int, *orderId*: int and *quantity*: int;
   - the *orderedProductId* is the primary key;
   - this table is used to make a correspondence between the order and the ordered products, gives the possibility of putting two or more products on an order;
   - the *productId* is bound to the *product* table's column *productId* by a foreign key constraint; deleting a product leads to deleting all the tuples having that product in this table;
   - the *orderId* is bound to the *ordert* table's column *orderId* by a foreign key constraint; deleting an order will lead to the deletion of all corresponding tuples in this table.

Scenarios and use-cases

The only use-cases is sketched in the next diagram and detailed below.



In the following use-case the primary actor is considered to be the user.

Use-case: *Order Processing*

Main success scenario:

1. The user enters in a .txt file valid commands and valid data for them.
2. The user opens Command Prompt.
3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command with the right argument (existent .txt file) to run the application.
5. The application verifies the existence of the files.
6. The application parses the input text.
7. The application executes the commands.

Alternative sequence A:

1. The user enters in a .txt file invalid commands or invalid data for them.
2. The user opens Command Prompt.

3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command with the right argument (existent .txt file) to run the application.
5. The application verifies the existence of the files.
6. The application parses the input text.
7. The application does not execute the invalid commands or the commands with invalid data; executes only the valid commands with valid data.

Alternative sequence B:

1. The user enters in a .txt file commands.
2. The user opens Command Prompt.
3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command with a non-existing file as argument to run the application.
5. An exception is thrown and the program exits.

Alternative sequence C:

1. The user enters in a .txt file commands.
2. The user opens Command Prompt.
3. The user changes the current directory with the directory in which the project is located.
4. The user introduces the appropriate command with the right argument (existent .txt file) to run the application.
5. A system error occurs and the file cannot be opened.
6. An exception is thrown and the program exits.

III. Design

1. Development approach

The Order Manager application is developed as before mentioned having a layered architecture, the structure obeys the object oriented paradigm. Thus, the levels or the layers of the app can be distinguished in 5 packages: *model*, *dataAccess*, *bllValidators*, *businessLogic* and *presentation*.

The package *model* includes the classes:

- *Client*: contains the attributes of a client resembling the fields of the table *client*;
- *Product*: contains the attributes of a product resembling the fields of the table *product*;
- *Ordert*: contains the attributes of an order resembling the fields of the table *ordert*;
- *OrderedProduct*: contains the attributes of an ordered product resembling the fields of the *orderedProduct* table.

The package *dataAccess* includes the classes:

- *ConnectionFactory*: implements the connection with the database;
- *AbstractDAO*: contains the generic methods for accessing the database: queries, generic basic operations like insert, delete, update, list and conversion of result set to object list;
- *ClientDAO*, *ProductDAO*, *OrderDAO*, *OrderedProductDAO*.

The package *bllValidators* includes the classes:

- *ClientValidator*: contains methods which validate client related input;
- *ProductValidator*: contains methods which validate product related input;
- *OrderValidator*: contains methods which validate order related input;

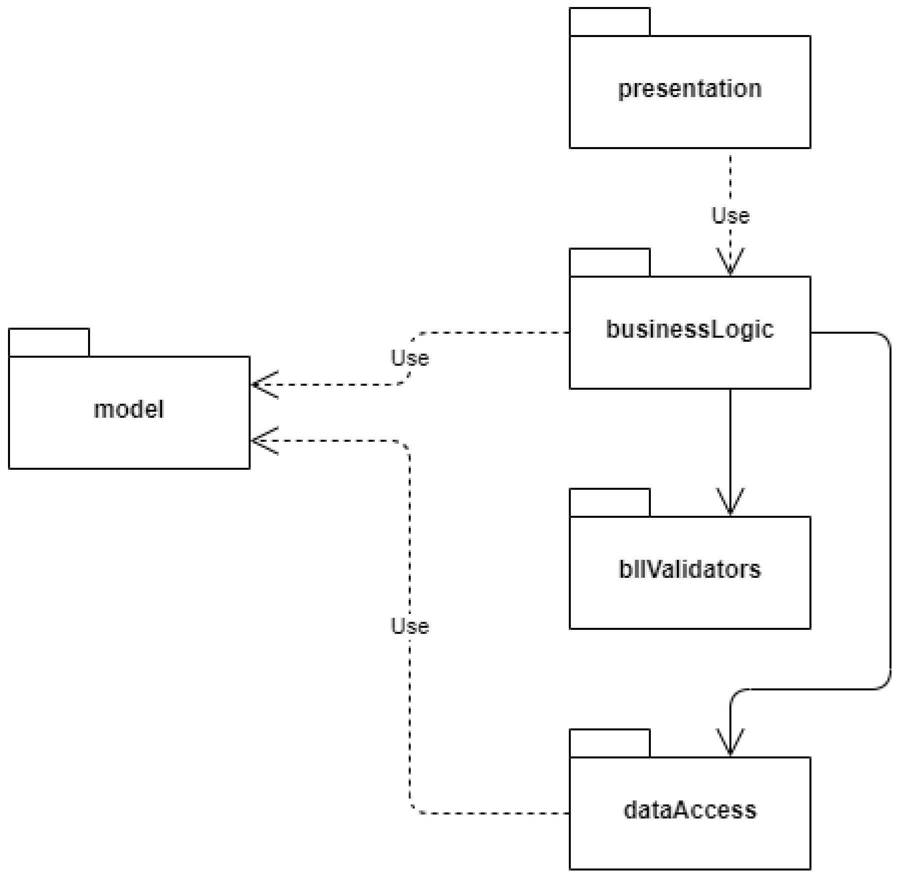
The package *businessLogic* includes the classes:

- *ClientBll*: implements the client related commands;
- *ProductBll*: implements the product related commands;
- *OrderBll*: implements the order related commands;
- *PDF*: generates PDF reports in table or message form.

The package *presentation* includes only the class *Controller* in which the input text is parsed and every command transmitted to the corresponding business logic component. This class also contains the main method in which the constructor of the controller is called.

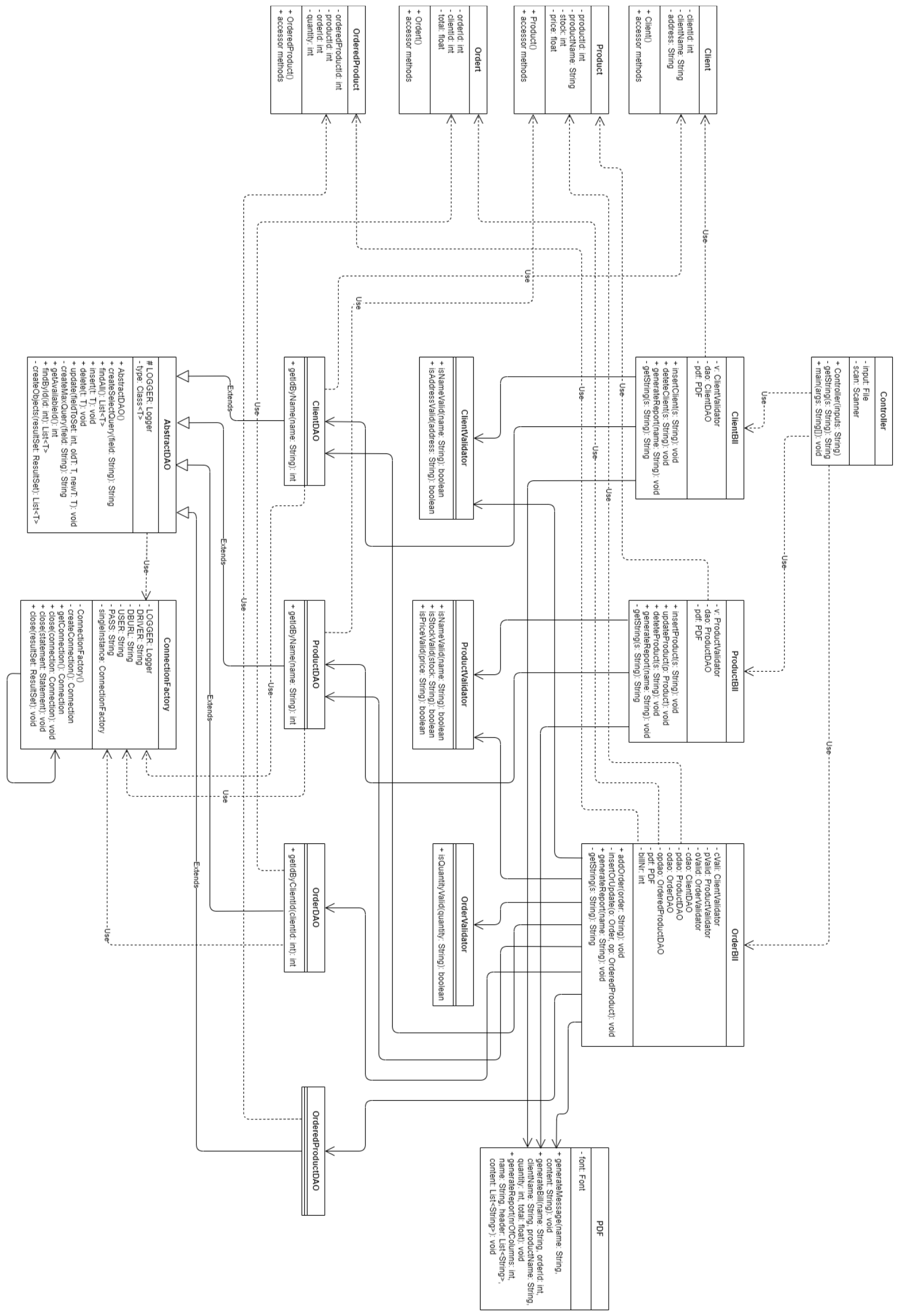
2. <u>Structure and UML Diagrams</u>

2.a. Package Diagram



The *presentation* calls methods from the *businessLogic* to handle the commands, the *businessLogic* uses *bllValidators* in order to validate the input and methods from *dataAccess* to execute the commands. The *businessLogic* and the *dataAccess* also use the *model* for having access to the contents of the tables.

The content of the packages is shown in the class diagram from 2.b.


Client
- clientId: int
- clientName: String
- address: String
+ Client()
+ accessor methods
Product
- productId: int
- productName: String
- stock: int
- price: float
+ Product()
+ accessor methods
Ordert
- orderId: int
- clientId: int
- total: float
+ Order()
+ accessor methods
OrderedProduct
- orderedProductId: int
- productId: int
- orderId: int
- quantity: int
+ OrderedProduct()
+ accessor methods
Controller
- input: File
- scan: Scanner
+ Controller(inputs: String)
- getString(s: String): String
+ main(args: String[]): void
ClientBll
- v: ClientValidator
- dao: ClientDAO
- pdf: PDF
+ insertClient(s: String): void
+ deleteClient(s: String): void
+ generateReport(name: String): void
- getString(s: String): String
ClientValidator
+ isNameValid(name: String): boolean
+ isAdressValid(address: String): boolean
ClientDAO
+ getIdByName(name: String): int
AbstractDAO
# LOGGER: Logger
- type: Class<T>
+ AbstractDAO()
+ createSelectQuery(field: String): String
+ findAll(): List<T>
+ insert(t: T): void
+ delete(t: T): void
+ update(fieldToSet: int, oldT: T, newT: T): void
- createMaxQuery(field: String): String
+ getMaxId(): int
+ findById(id: int): List<T>
- createObjects(resultSet: ResultSet): List<T>
ProductBll
- v: ProductValidator
- dao: ProductDAO
- pdf: PDF
+ insertProduct(s: String): void
+ updateProduct(p: Product): void
+ deleteProduct(s: String): void
+ generateReport(name: String): void
- getString(s: String): String
ProductValidator
+ isNameValid(name: String): boolean
+ isStockValid(stock: String): boolean
+ isPriceValid(price: String): boolean
ProductDAO
+ getIdByName(name: String): int
ConnectionFactory
- LOGGER: Logger
- DRIVER: String
- DBURL: String
- USER: String
- PASS: String
- singleInstance: ConnectionFactory
- ConnectionFactory()
- createConnection(): Connection
+ getConnection(): Connection
+ close(connection: Connection): void
+ close(statement: Statement): void
+ close(resultSet: ResultSet): void
OrderBll
- cValid: ClientValidator
- pValid: ProductValidator
- oValid: OrderValidator
- cdao: ClientDAO
- pdao: ProductDAO
- odao: OrderDAO
- opdao: OrderedProductDAO
- pdf: PDF
- billNr: int
+ addOrder(order: String): void
- insertOrUpdate(o: Order, op: OrderedProduct): void
+ generateReport(name: String): void
- getString(s: String): String
OrderValidator
+ isQuantityValid(quantity: String): boolean
OrderDAO
+ getIdByClientId(clientId: int): int
OrderedProductDAO
PDF
- font: Font
+ generateMessage(name: String, content: String): void
+ generateBill(name: String, orderId: int, clientName: String, productName: String, quantity: int, total: float): void
+ generateReport(nrOfColumns: int, name: String, header: List<String>, content: List<String>): void
Use
Extends

IV. Implementation

1. model package

For each of the following classes the variables represent the attributes of a client/product/order/ordered product and they match the fields in the corresponding table.

The classes only contain the constructor (the simplest one), getters and setters in order to have access to the private instance variables (all of them) since their only purpose is to provide a model for the tables entries.

1.a. Class Client

Instance variables: *clientId*: int, *clientName*: String, *address*: String.

1.b. Class Product

Instance variables: *productId*: int, *productName*: String, *stock*: int, *price*: float.

1.c. Class Ordert

Instance variables: *orderId*: int, *clientId*: int, *total*: float.

1.d. Class OrderedProduct

Instance variables: *orderdProductId*: int, *productId*: int, *orderId*: int, *quantity*: int.

2. dataAccess package

2.a. Class ConnectionFactory

Instance variables: *LOGGER*: logger, *DRIVER*: String, *DBURL*: String, *USER*: String, *PASS*: String, *singleInstance*: ConnectionFactory.

Some of the variables are constants and they represent the requirements to obtain a connection to the MySQL databases.

Besides the constructor, the class contains the following methods that will allow us to use the database:

```
private Connection createConnection() {...}

public static Connection getConnection() {...}

public static void close(Connection connection) {...}

public static void close(Statement statement) {...}

public static void close(ResultSet resultSet) {...}
```

The first two will be used to make and to use a connection, and the last three to close the connection, a statement or a result set obtained after executing a query.

2.b. Class AbstractDAO

Instance variables: *LOGGER*: logger, *type*: Class<T>.

This class is written using reflection techniques, and contains the methods for accessing the database. The queries for accessing the database for a specific object that corresponds to a table will be generated dynamically through reflection. The type of the object is specified by *T*.

The constructor `public AbstractDAO() {...}` initializes the class extending this one with a specific type.

The method `public String createSelectQuery(String field) {...}` is used to obtain a generic **select** query which will have the following form: SELECT * FROM table WHERE field = ?. The table will be the corresponding one for which the class is extended, i.e. for *CliendDAO* the table will be *client*, and the *field* is specified in the argument.

The method `public List<T> findAll() {...}` is used to display the entire content of a table. It starts by getting a connection to the database and forming a **select** query without a condition so that it will return the entire table. The result will be returned to a ResultSet which will be converted into a list of corresponding objects, using one of the following methods and returned.

The method `public void insert(T t) {...}` performs the insertion of a new instance in a table. It gets as argument the object to be inserted. It also starts by getting a connection to the database. An **insert** query of the following form will be generated: INSERT INTO table (column1, column2, ...) VALUES (value1, value2, ...). In order to construct this statement, firstly, the fields of the specific type are obtained and completed in the positions of *columnX*, and then the positions of *valueX* are completed with as many '?' as the number of fields. After this, the positions of the fields from *t* are set to be accessible and the '?' are replaced with the values of the fields of *t*. In the end, the query is executed.

The method `public void delete(T t) {...}` performs the deletion of an instance from a table. It gets as argument the object to be deleted. It begins with getting a connection set and then building the **delete** query. The query will have the following form: DELETE FROM table WHERE field = ?. Instead of *table* the name of *T* will be put and the first attribute of class *T*, which is usually the id, will be chosen to be the condition. The '?' will be replaced with the value of *t* for the chosen, i.e. first, attribute of *T*. The query is then executed.

The method `public void update(int fieldToSet, T oldT, T newT) {...}` is used to modify an instance of a table. It gets as arguments two objects of type *T*: the one to be modified, and the one which has the desired values. It also gets as argument the number of the filed we want to modify. A connection is set and an **update** query is formed. The query will have the form: UPDATE table SET field1 = ? WHERE field2 = ?. In this statement, the *table* is replaced just as in the previous methods. On the position of *field1* is put the name of the field with number *fieldToSet* and on *field2* is put the name of the first field. Then in place of '?'s the values of the corresponding fields from *oldT* and *newT* are placed. The query is in the end executed.

The method `private String createMaxQuery(String field) {...}` returns a string corresponding to a **max** query of the following form: SELECT MAX field FROM table.

The method `public int getAvailableId(){...}` is useful when we aim to insert in a table. The method finds based on the previous method the maximum of the ids and returns this value incremented by 1.

The method `public List<T> findById(int id) {...}` is used to obtain an element with a specific id. It uses a select query where the condition will be the id. Then the query is executed and the result set is converted into a list of objects of the right type.

The method `private List<T> createObjects(ResultSet resultSet) {...}` gets as argument a result set which is converted to a list of objects. This method was used in some of the above described methods.

All the methods which executed some query, also close the connection, the statement and the result set in a finally block.

2.c. <u>Class ClientDAO</u>

This class extends the previous class. Thus, it customizes all the methods presented in AbstractDAO with the type Client.

It also contains a specialized method `public int getIdByName(String name){...}` which is used for obtaining the id of a client when we know its name. This is useful since all the commands for a client are given with the name and we need to find that client's id. It uses a **select** query in which the condition will be *clientName*. The result will return only one client since the name is unique and the method will return the id of the result.

2.d. <u>Class ProductDAO</u>

This class extends AbstractDAO and it customizes all the methods presented there with the type Product.

As in ClientDAO, there is only one method, the same, just changed for products instead of clients.

2.e. <u>Class OrderDAO</u>

This class extends AbstractDAO it customizes all the methods presented there with the type Ordert.

It contains a singular method similar to the ones in ClientDAO and ProductDAO, `public int getIdByClientId(int clientId){...}`, which is used to obtain the order of a client when we know its *clientId*. It uses a **select** query where the condition is the *clientId*. The result will return only one order since all the orders of a client will be updated in the same order(only in table *orderedProducts* will be inserted the items, in table *ordert* only the total will be updated; all the ordered products of a client will be placed in the same order).

2.f. <u>Class OrderedProductDAO</u>

This class extends AbstractDAO it customizes all the methods presented there with the type OrderedProduct.

This class does not contain any specific methods.

3. <u>bllValidators package</u>

3.a. <u>Class ClientValidator</u>

Contains only two simple methods.

The first one `public boolean isNameValid(String name) {...}`, checks whether a string representing a possible name of a client is valid or not by checking if it contains only letters using a regular expression.

The second one `public boolean isAddressValid(String address) {...}`, checks in the same manner if a possible address given as a string is valid. The method checks here for letters, digits and the characters '.', ',', '-'.

3.b. Class ProductValidator

Contains three validation methods, similar with the ones in ClientValidator.

The method `public boolean isNameValid(String name) {...}` is identical with the one checking the name of the client.

The methods `public boolean isStockValid(String stock) {...}` and `public boolean isPriceValid(String price) {...}` check for numbers of type int, respectively float, given as strings, using regular expressions. They verify that the string contains only digits, the second one allowing also '.' for float numbers.

3.c. Class OrderValidator

Contains only one method because only one field requires verification. The method is `public boolean isQuantityValid(String quantity) {...}` and checks just as in ProductValidator for an integer number. The method returns *true* if the string only contains digits.

4. businessLogic package

4.a. Class PDF

Instance variables: *font*: Font.

In this class methods for generating PDF documents are implemented.

For instance, the first method `public void generateMessage(String name, String content) {...}`, constructs a file which contains a message given as argument, having the name also specified as argument.

The second method `public void generateBill(String name, int orderId, String clientName, String productName, int quantity, float total) {...}` is used to generate bills after an order is executed successfully. The name of the PDF is specified in the first argument, the others representing the necessary data to display on the bill. The information will be organized in a table having as headers Order Id, Client name, Ordered product, Quantity, Total. This columns will receive the values from the arguments.

The last method `public void generateReport(int nrOfColumns, String name, List<String> header, List<String> content) {...}` is used to construct a PDF with a table customized for the content we want to display, i.e. the clients, the products or the orders. The table is built with *nrOfColumns* columns having the headers specified in the list received as argument. The content of the table is also given as a list of strings.

4.b. Class ClientBll

Instance variables: *v*: ClientValidator, *dao*: ClientDAO, *pdf*: PDF.

In this class are developed methods which implement the required commands on clients: insert client, delete client, and generate report.

An additional method which is written and used here is `private String getString(String s) {...}` which removes the extra spaces in a string and also removes the space in the front and in the end of a string.

The method `public void insertClient(String s) {...}` receives as argument the client to be inserted in the following way: from the command which has the form " Insert client: Ion Popescu, Bucuresti ", the *s* given as parameter is the entire line form ':'. A client *c* is created and its attributes will be set in the following steps. The name is obtained by taking the substring of *s* which contains the characters until it reaches ',', it is processed with the *getString* method and is validated. If it is valid, then the name of *c* is set. The address is the rest of the string and is processed and verified in the same manner. If this is valid, the address of *c* is set. If the method did not returned until this point, it means the client is valid and can be inserted. Before this, an id is set with the *getAvailableId* method. Then the *insert* method is called. If the client's name does already exist in the table, the insertion is not performed. Otherwise, the command will be executed successfully.

The method `public void deleteClient(String s) {...}` receives as argument the client to be deleted in the following way: from the command which has the form " Delete client: Ion Popescu, Bucuresti ", the *s* given as a parameter is the entire line from ':'. A client *c* is created and its name and address are set in the same way as for insertion. The method then searches in the table for a client with this name and if the address of the found client with the one we want to delete match, than the client can be removed from the table.

The last method `public void generateReport(String name) {...}` corresponds to the command "Report client". It firstly forms a list of all the clients by calling the *findAll* method from ClientDAO and then a list of the headers of the tables, which will be the fields of Client. The content of the report table will be obtained by transforming each field of each client in the list in strings and putting it into a list of strings. In the end, the generateReport method is called.

4.c. <u>Class ProductBll</u>

Instance variables: *v*: ProductValidator, *dao*: ProductDAO, *pdf*: PDF.

In this class are developed methods which implement the required commands on products: insert product, delete product, and generate report.

In this class the same method `private String getString(String s) {...}` as in the previous class is implemented.

The method `public void insertProduct(String s) {...}` implements the insertion operation. It receives as argument a string of the form "apple, 20, 1", representing the name of the product, the stock and the price. The method creates an object *p* of type Product. Its name, stock and price are taken from the input string and validated. Then, we verify if the product already exists in the table. If the *getIdByName* method returns 0, it means there is no such product in the table and the insertion is executed with the next available id. If the call of *getIdByName* returned an existing id for that product, *p* gets the value of that id and an update for that product is required. Thus, the method `private void updateProduct(Product p) {...}` implements an update for the product with the same id as the product received as argument. The method increases the old stock with the stock of the product to be inserted and the price is set to be the new one, the one of *p*. In this case, the method *update* from ProductDAO is called twice, once for changing the stock, and once for changing the price.

The method `public void deleteProduct(String s) {...}` implements the command *Delete product*. The argument represents the name of the product to be deleted. The method checks the validity of the product and if the validator returns an inconsistency, the method ends. Otherwise, the product with the id corresponding to the given name is deleted.

The last method in this class is `public void generateReport(String name) {...}` which is similar to the one with the same name in class ClientBll. The difference consists in the fields which are to be inserted in the table.

4.d. <u>Class OrderBll</u>

Instance variables: objects of each type of validators, objects of each type of DAOs, *pdf*: PDF, *billNr*: int.

The first method, just as in the previous two classes is `private String getString(String s) {...}`, having the same functioning.

The first method to implement a command here is `public void addOrder(String order) {...}` which performs the execution of an order. The method gets as argument a string of the form "Luca George, apple, 5 " which will be parsed and validated. Two objects are created in this method, *o* of type Ordert and *op* of type OrderedProduct since changes in these two tables are required. Firstly, we check the validity and the existence of the client and the product. If any of these conditions is not fulfilled, the method terminates. Otherwise, the next attribute which is the quantity is also validated. A further condition is imposed to this, namely, to be smaller or at most equal with the stock. If the order is too large, a PDF representing the bill will be generated with the under-stock message. If the quantity is a proper one, the next step is to compute the total price: the quantity * the price of the product. After this, the method `private void insertOrUpdate(Ordert o, OrderedProduct op) {...}` is called. This method decides whether a new order has to be inserted, or it is required an update to a previous one. Thus, if the client has not performed an order before, the current order will be inserted in table *ordert* and *orderedProduct* with the next available ids. In case the client has previously made an order, the current command will get the id of the previous order, update the total price for it by adding the new cost to the old total and only in *orderdProduct* table, an insertion will be executed. Returning from this method, the stock of the product is decremented with the bought quantity and a PDF bill is generated.

The other method in this class is the `public void generateReport(String name) {...}` which works the same as the other *generateReport* methods from ClientBll and ProductBll.

5. <u>presentation package</u>

<u>Class Controller</u>

Instance variables: *input*: File, *scan*: Scanner.

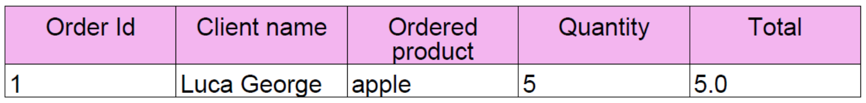
The constructor of this class has the role of the parser. It gets the name of the input file as an argument and processes the content of the file. It reads line by line from the file and detects the commands and distributes them at the right Bll method.

The last method here is the main method `public static void main(String[] args) {...}` , its arguments being the input file. An object of this class is initialized and only the constructor is called.

V. Results

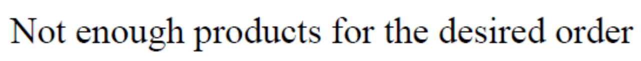
For verifying the correctness of the execution, the program was tested with the input file provided in the assignment requirements. The output is visible in the generated PDFs: 3 bills, 4 client reports, 2 order reports and 3 product reports.

Example of bill:



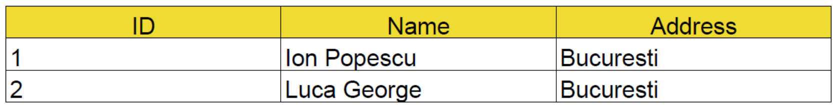
Order Id	Client name	Ordered product	Quantity	Total
1	Luca George	apple	5	5.0

Example of bill with under-stock message:



Not enough products for the desired order

Example of report:



ID	Name	Address
1	Ion Popescu	Bucuresti
2	Luca George	Bucuresti

V. Conclusions

The Order Manager fulfills all the requirements of the laboratory work.

For future improvements the following can be considered:

- Graphic interface for entering the input data;
- Adding new commands;
- Improving the structure of the commands to allow multiple clients with the same name.

VI. Bibliography

1. https://mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/
2. https://ibytecode.com/blog/jdbc-mysql-create-database-example/
3. https://dzone.com/articles/layers-standard-enterprise
4. http://tutorials.jenkov.com/java-reflection/index.html
5. https://www.baeldung.com/java-pdf-creation
6. https://www.vogella.com/tutorials/JavaPDF/article.html
7. https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html
8. https://dev.mysql.com/doc/refman/5.7/en/using-mysqldump.html
9. http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/Assignment_2/