**PROGRAMMING TECHNIQUES**

Assignment 3: ORDER MANAGEMENT

Omer Miriem, group 30421

**Contents:**

1. Problem’s Objective
2. Problem Analysis
3. Design
4. Implementation
5. Results
6. Conclusions
7. Bibliography
8. **Problem’s Objective**

Consider an application **OrderManagement** for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application should be structured in packages using layered architecture and should use (minimally) the following classes:

* **Model classes –** the 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 should allow processing commands from a text file given as argument, perform the requested operations, save the data in a database, and generate reports in pdf format.

Implement a parser to read command in the Presentation layer and a pdf generator to generate reports.

1. **Problem Analysis**

* **General overview**

This application should be able to fulfill all the requirements in order to generate pdfs that keep track of orders, clients and products. These are stored in a relational MYSQL database, along with the information about the users which have access to the system. This way, all the data is easier to retrieve and access from different computers.

Order management refers to storing orders and performing operations on them. They constitute a virtual representation of an operation in the real world. A virtual product listed in the application is a real object and the customer is a person that needs a service. They must be physically stored in a warehouse.

To manage all the orders in a proper way, this application will be done via pdf files.

Order management process

The first step in all of this is the moment when the client decides to place an order. Keeping track of these incoming orders may be relatively simple when stating up. But it gets a lot trickier as more and more orders start coming in from various clients.

It allows a business to coordinate the entire fulfillment process- from order collection, inventory to service availability.

Order management in real life

Steps:

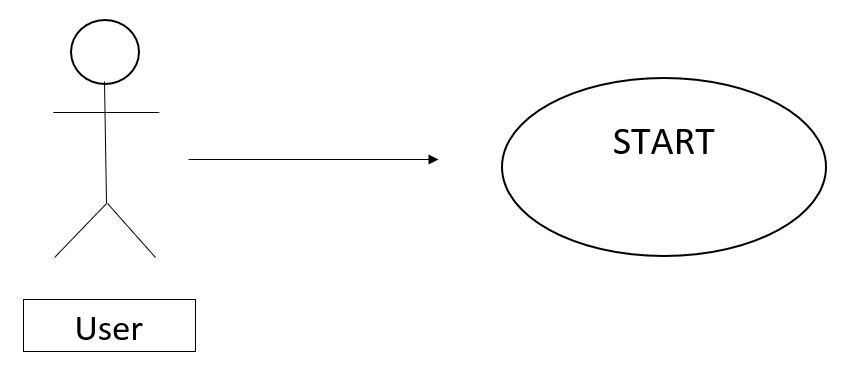
1. Placement: the client places the order; a sales team member checks the details and confirms the order
2. Fulfillment: a warehouse employee confirms shipping details, generates and invoice and fulfill the order
3. Inventory management: inventory levels are monitored as they fluctuate with the demands of the business

Why order management is important

Order management touches virtually every system and process in the supply chain. This results in costly manual processes to complete and deliver the order without errors.

* **Use Case**

**-**methodology used to identify, clarify and organize system requirements



The user starts the program with predefined parameters. The program itself will do the job regardless of the user interaction.

* **Scenarios**

Scenario 1: SUCCESS

The input file introduced by the user is correct. The application will perform the requested operations, save the data in a database and generate reports in pdf format.

-in case that there are not enough products, the order will not be created and the PDF document representing the bill will not be generated.

Scenario 2: ERROR

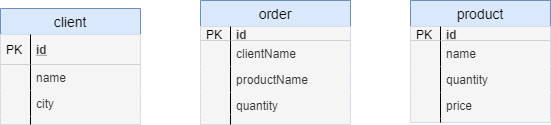
If the user does not introduce the correct name for the input file or if there are some mistakes( the way the commands are written in the text file), the application will not perform the requested operations.

Another error scenario can appear if the application fails to connect to the database.

1. **Design**

We need to closely examine what is happening in this system before we can implement a model for it. We will use object-oriented programming techniques to implement the various elements-the clients, the product, the order etc.

* Database schema



Relational databases were used for storing the data for the application, containing the following tables:

-client:

* + id: Primary key, NOT NULL, Auto-increment
  + name: VARCHAR(45)
  + city: VARCHAR(45)

-order:

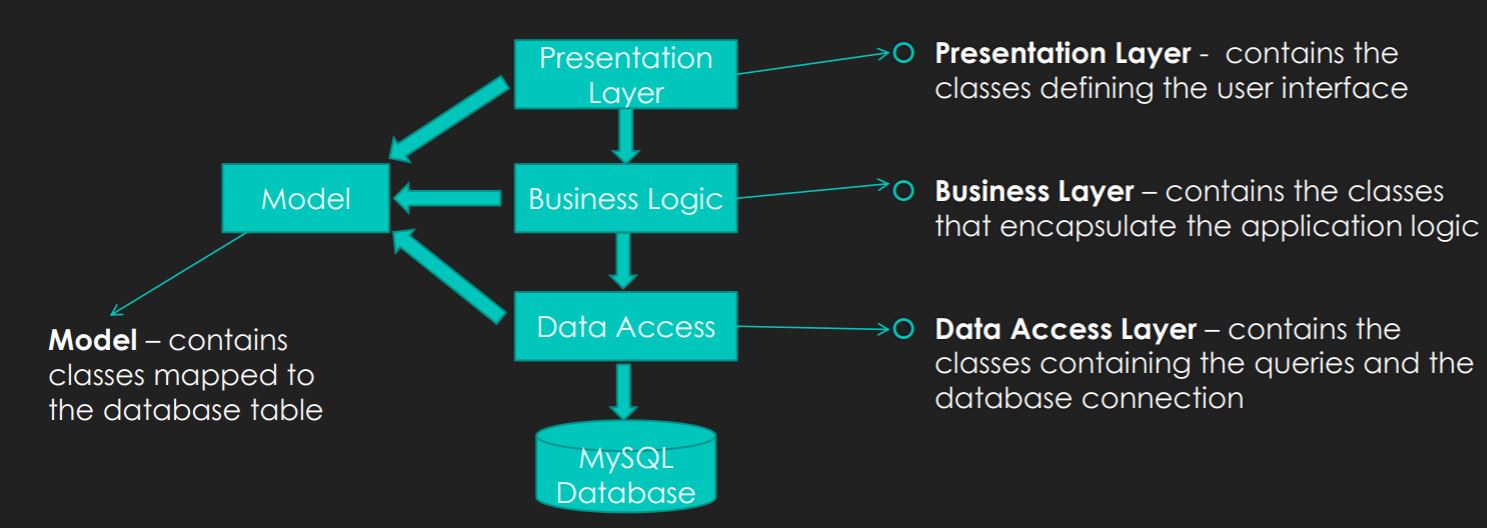
* id: Primary key, NOT NULL,Unique, Auto-increment
* clientName: VARCHAR(45)
* productName: VARCHAR(45)
* quantity: INT(11)

-product:

* id: Primary key, NOT NULL, Unique, Auto-increment
* name: VARCHAR(45)
* quantity: VARCHAR(45)
* price: FLOAT()

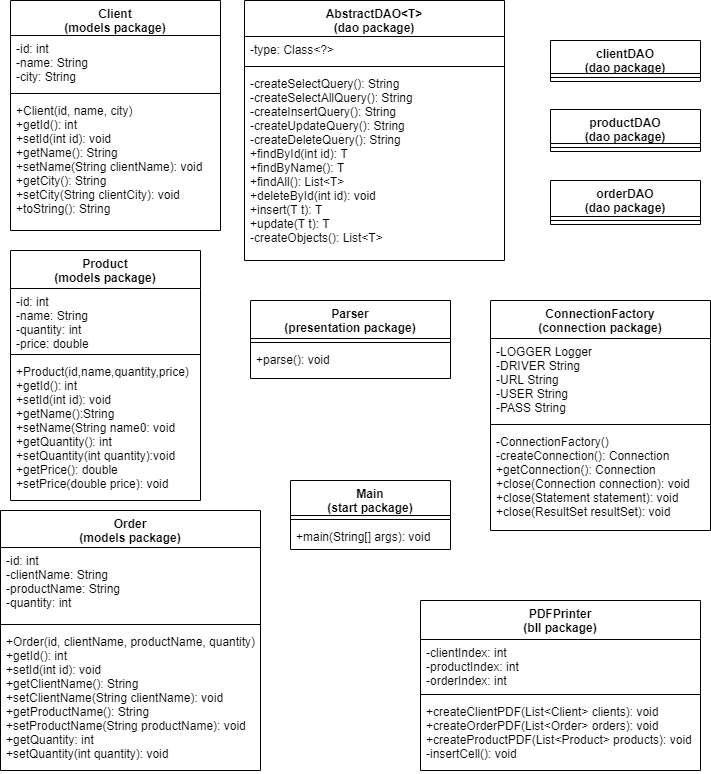
The packages used for creating this application are of type Layered Architecture.

I split my application in different layers, each layer having a special purpose.



My packages contain the following classes:

* models package: Client, Product and Order classes
* presentation package: Parser class
* bll (Business Logic): PDFPrinter class
* dao (Data Access): AbstractDAO, ClientDAO, OrderDAO, ProductDAO class
* connection: ConnectionFactory class
* start: Main class
* **UML Diagram**



**Data Structure**

The most important data structure in this application is the ArrayList, which is used to store the products, the clients and also the orders.

**Class Design**

The whole idea of splitting the program into classes is based on a general rule named divide and conquer. This paradigm can be used almost everywhere: you divide a problem into smaller problems and then you solve these little, simple and well-known problems.

Dividing the program into classes is one of the type division which started to become common in last decade. In this programming paradigm, we model our problems by some objects and try to solve the problem by sending messages between these objects.

At the base of the implementation of this program is the technique of Reflection. Reflection is an API that is used to examine or modify the behavior methods, classes, interfaces at runtime.

Through this technique, we can invoke methods, during running, regardless of the access specifier used with them.

* start
* Main: starts the application
* connection
* ConnectionFactory: is responsible with the connection with the database, creating a statement. It also has methods to close the connection/ the statement.
* models
* Client: models the client table from the database
* Product: models the product table from the database
* Order: models the order table from the database
* dao
* AbstractDAO: this class is implemented based on the properties of the Reflection technique. In order to facilitate the writing of the class methods, it has an instance variable of the class type which refers to the type of the generic variable T.
* private String createSelectQuery()
* private String createSelectAllQuery()
* private String createInsertQuery()
* private String createUpdateQuery()
* private String createDeleteQuery()
* public T findById(int id)
* public T findbyName(String name)
* public List<T> findAll()
* public void deleteById(int id)
* public T insert(T t)
* public T update(T t)
* public List<T> createObjects(ResultSet resultSet)
* ClientDAO, ProductDAO, OrderDAO extend AbstractDAO class

At the time of the extension, the data type T is replaced with Client, Product and Order respectively. The use of the reflection technique makes it possible to use methods of the parent class, in the classes they inherit regardless of the instance variable of the classes.

* bll
* PDFPrinter: the class that contains the methods to create PDF reports for Client, Product and Order
* presentation
* Parser: implements a parser that reads commands from a text file and generated pdf files(for Client, Product and Order Report)

1. **Implementation**

This chapter will provide a closer look at the role and the implementation of each class, with its attributes, constructors and methods. Some of the details can be also found in the JAVADOC files generated.

**Client Class(models package):** it resembles the Client table from the SQL database. The attributes of the Client Class represent the header of the corresponding table.

This class will contain the following Fields:

-id: a Integer data type variable, which represent the id of the client

-name: a String data type variable, which represents the name of the Client

-city: a String data type variable, which represent the city(address) of the Client

**Product Class (models package):** it resembles the Product table from the SQL database. The attributes of the Product Class represent the header of the corresponding table;

This class will contain the following Fields:

-id: a Integer data type variable, which represent the id of the Product

-name: a String data type variable, which will represent the name of the Product

-quantity: an Integer data type variable, which will represent the stock of the Product (the number of available existing products)

-price: a Double data type variable, which will represent the price of the Product

**Order Class(models package):** it resembles the Order table from the SQL database. The attributes of the Order Class represent the header of the corresponding table;

This class will contain the following Fields:

-id: a Integer data type variable, which represent the id of the order

-clientName: : a String data type variable, which represents the name of the Client placing the order

-productName: : a String data type variable, which represents the name of the Product ordered

-quantity: an Integer data type variable, which will represent the quantity wanted

For every model class some getters and setters are written to have access to their attributes

The ConnectionFactory class is a singleton class, it’s constructor is private and all attributes are static, and they are part of the connection (like url, user and password of the database).

Methods are implemented for creating and closing a connection, a statement and result set.

private ConnectionFactory() {  
 try {  
 Class.*forName*(*DRIVER*);  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
}

private Connection createConnection() {  
 Connection connection = null;  
 try {  
 connection = DriverManager.*getConnection*(*URL*, *USER*, *PASS*);  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, "ERROR to open");  
 e.printStackTrace();  
 }  
 return connection;  
}

public static void close(Connection connection) {  
 if (connection != null) {  
 try {  
 connection.close();  
 } catch (SQLException e) {  
 *LOGGER*.log(Level.*WARNING*, "ERROR to close connection");  
 }  
 }  
}

In the AbstractDAO class queries are build up with StringBuilders with reflection techniques, two exemples are Select and Insert queries:

private String createSelectQuery(String field) {  
 StringBuilder sb = new StringBuilder();  
 sb.append("SELECT ");  
 sb.append(" \* ");  
 sb.append(" FROM ");  
 sb.append("sql." + type.getSimpleName());  
 sb.append(" WHERE " + field + " =?");  
 return sb.toString();  
}

And insert:

private String createInsertQuery() {  
 StringBuilder sb = new StringBuilder();  
 sb.append("INSERT ");  
 sb.append(" INTO ");  
 sb.append("sql." + type.getSimpleName());  
 sb.append(" VALUES( ");  
 for (int i = 0; i < type.getDeclaredFields().length; i++) {  
 sb.append("?,");  
 }  
 sb.setLength(sb.length() - 1);  
 sb.append(")");  
 return sb.toString();  
}

These queries are used when a statement is prepared to be executed. When we execute a query, the connection is instantiated than the statement is prepared using a string containing the query (returned by one of the previous methods). Then the parameters are set, and the statement is executed, than the result is stored in a result set.

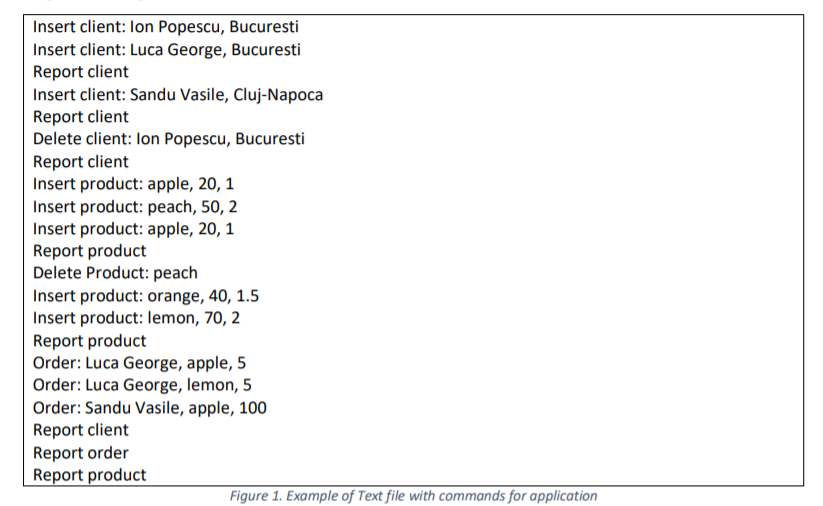
public T insert(T t) {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 String query = createInsertQuery();  
 try {  
 int index = 1;  
 connection = ConnectionFactory.*getConnection*();  
 statement = connection.prepareStatement(query);  
 for (Field f : type.getDeclaredFields()) {  
 PropertyDescriptor propertyDescriptor = new PropertyDescriptor(f.getName(), type);  
 Method method = propertyDescriptor.getReadMethod();  
 Object s = method.invoke(t);  
 statement.setObject(index, s);  
 index++;  
 }  
 statement.executeUpdate();  
 return t;  
 } catch (SQLException | IntrospectionException | IllegalAccessException | IllegalArgumentException  
 | InvocationTargetException e) {  
 *LOGGER*.log(Level.*WARNING*, "DAO:insert " + e.getClass() + e.getMessage());  
 } finally {  
 ConnectionFactory.*close*(statement);  
 ConnectionFactory.*close*(connection);  
 }  
 return null;  
}

The **ClientDAO**, **OrderDAO** and **ProductDAO** classes extend the generic AbstractDAO class and specify the needed operations.

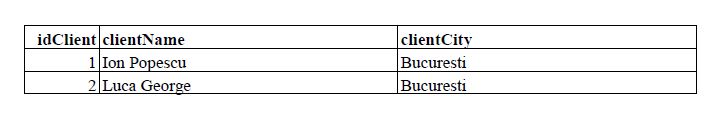
**Parser** class(presentation package): was created in order to implement the User input. The data will be extracted from a text file using Parser, which will be implemented here.

1. **Results**

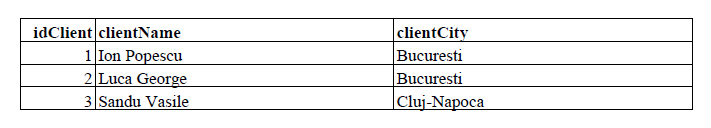
Example of text file with commands for application:



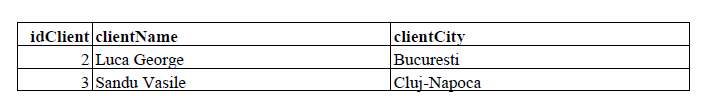
The first report client ( after inserting client Ion Popescu, Bucuresti and Luca George, Bucuresti):



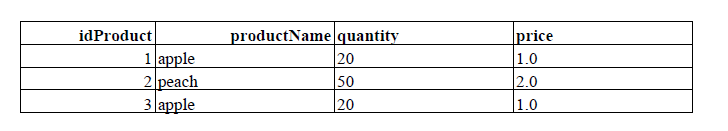
The second report client ( after inserting client Sandu Vasile, Cluj-Napoca):



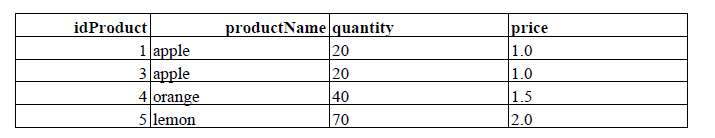
The 3rd report client ( after deleting client Ion Popescu):



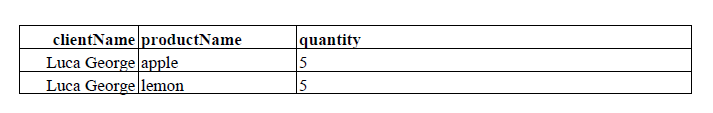
The first report product ( after inserting apple, peach and apple with the quantity and price as in the input file):



The second report product ( after deleting peach and inserting orange and lemon):

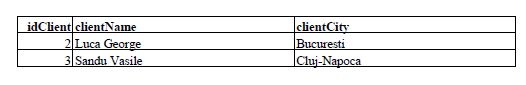


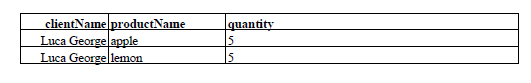
The first report order ( after placing order for Luca George and Sandu Vasile). The number of products Sandu Vasile has in the order is greater than the number of products in stock. In this case, the application will not generate its order in the PDF.



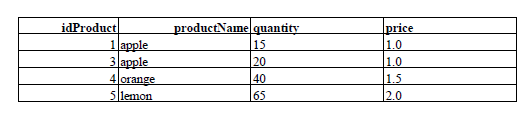
Finally, the last 3 report:

Report client, Report order and Report product



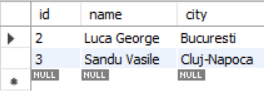


The product stock has been updated after the orders.

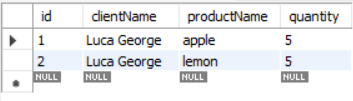


Final SQL tables:

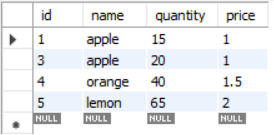
Client table:



Order table:



Product table:



1. **Conclusions**

This project was a good exercise in remembering the OOP concepts learned in the first semester, but also learning new ones. I found it very useful and challenging.

This assignment helped me understand how important is the connection between databases and java.

I learned how to access a database through java code and some operations such as select, update and delete from the database.

1. **Bibliography**

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

<https://www.mysamplecode.com/>

<https://mkyong.com/jdbc>

<https://www.boraji.com/>

draw.io- for diagrams