**3rd Assignment, PT**

**Andriesă Maria**

**-Order Management-**

**1.Objectives**

**Main objectives**

Problem specification: Consider an application OrderManagement for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders.

For storing products and customer order processing we need to establish a connection with a database management system. So our objectives are the next ones :

* Analyze the application domain, determine the structure and behavior of its classes and draw an extended UML class diagram.
* Implement the application classes. Use javadoc for documenting classes.
* Use reflection techniques to create a method createTable that receives a list of objects and generates the header of the table by extracting through reflection the object properties and then populates the table
* Implement a system of utility programs for reporting such as: under-stock, totals, filters, etc .

Other classes and packages can be added to implement the full functionality of the application.

Secondary objectives

* Draw extended UML diagram
* Document classes using Javadoc
* Generate header of table using reflection
* Utility programs for reporting

**2.Analysis ,Modelling, Use cases Scenarios**

2.1 Analysis

The application I implemented provides a comfortable environment for database management within the windows of the application. In order to be more efficient when introducing data into the tables and managing the tables, the graphical interface facilitates the interaction with the database and enables even unexperienced users to work with a database. Otherwise, the user would have to learn to write SQL queries and apply them to the database and this requires too much work, effort and energy spent on introducing large amounts of information in the database.

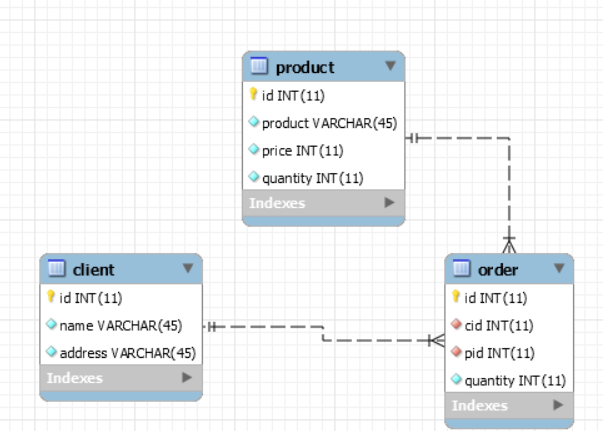
A relational database model implies that we need to have tables which are connected relationships. Each table is characterized by an ID, which is incremented automatically. Through the ID, we can distinguish one object from another. The other fields are not usually unique to an entry and we can have more identical fields on a column.

2.2. Modeling

The modelling of this application has been done using the 3-tier Architecture, imitating a server client solution, with every part being played by our personal computer. The base classes are the tables which are entities that have a 1 to 1 relationship with the classes Client, Product, Order from the application. Each entity models a real life element. Each of the model classes contains fields which correspond to a column in the table with the same name as the class. These classes make the model, the architecture being composed of the packages presentation, business\_logic, data\_acces, The before mentioned packages represent different layers of the application, facilitating a further scalability and organization of the program.

Two main tools are required for the implementation of the problem: MySQL and Java. For each database model, there needs to be a corresponding class in the Java application. The two tools are strongly related, communicating towards a common goal, which is an easy to use application without the need to write queries by hand.

The database tables are quite intuitive: client, product and order. All these tables support the create, read, update and delete functions (CRUD) that any database management system allows. These operations are mirrored in the Java data abstraction layer, allowing us to perform operations on the table data. The diagram of the database is displayed below:



For each entity in the database I created a class having a 1:1 relationship between the two structures. Each class from the model has as many fields as there are in the associate database table, with the exact same name and only getter methods.

2.3. Use Cases

As per the use case definition, the control lies in the hands of the user which is, in this case, the administrator of the database. The application has been created with the user’s experience in mind and provides a little guide for using every ability of the application correctly. The administrator can be any person in charge of introducing data and handling the data inside the database. And there are a few scenarios that the administrator is going to pass through when using the application, in order for it to function correctly.

2.4. Scenarios

Scenarios represent the tracking of steps taken by the user when uses the application. Their purpose is to predict any unwanted situation which can be caused by the user and to prevent them. Next, due to the similar behavior of all the entities in the database I will present the operations that can be done on the Client entity which has buttons as handles in our interface.

**Scenario #1- Inserting a client**

1. User enters the application
2. In the Client tab he writes the name and the address of the newly inserted client
3. He doesn’t write an id because it is automatically generated.
4. The user presses the insert button.
5. If the client’s name is correctly written, i.e. contains only letters, the new client is inserted in the database and its information is displayed in the table in the tab.

**Scenario #2 –Editing a client**

1. User enters the Client tab in the application.
2. To edit a client’s information, the user has to write the client’s id and all the possible information that has to be changed for a client.
3. If any field is left empty, a notification will pop up.
4. If the input is correct, then the update is proceeded to be applied in the database and the client’s new information appears in the database immediately.

**Scenario #3 – Deleting a client**

1. User enters the Client tab in the application.
2. To delete a client, the user has two options: first, to select the client from the table by clicking the row with the needed client to be deleted and then press the delete button.
3. Secondly, to write the client’s to be deleted id in the id field and then press the delete button.
4. The client’s data will be deleted from both user interface table and the database.

3. Design

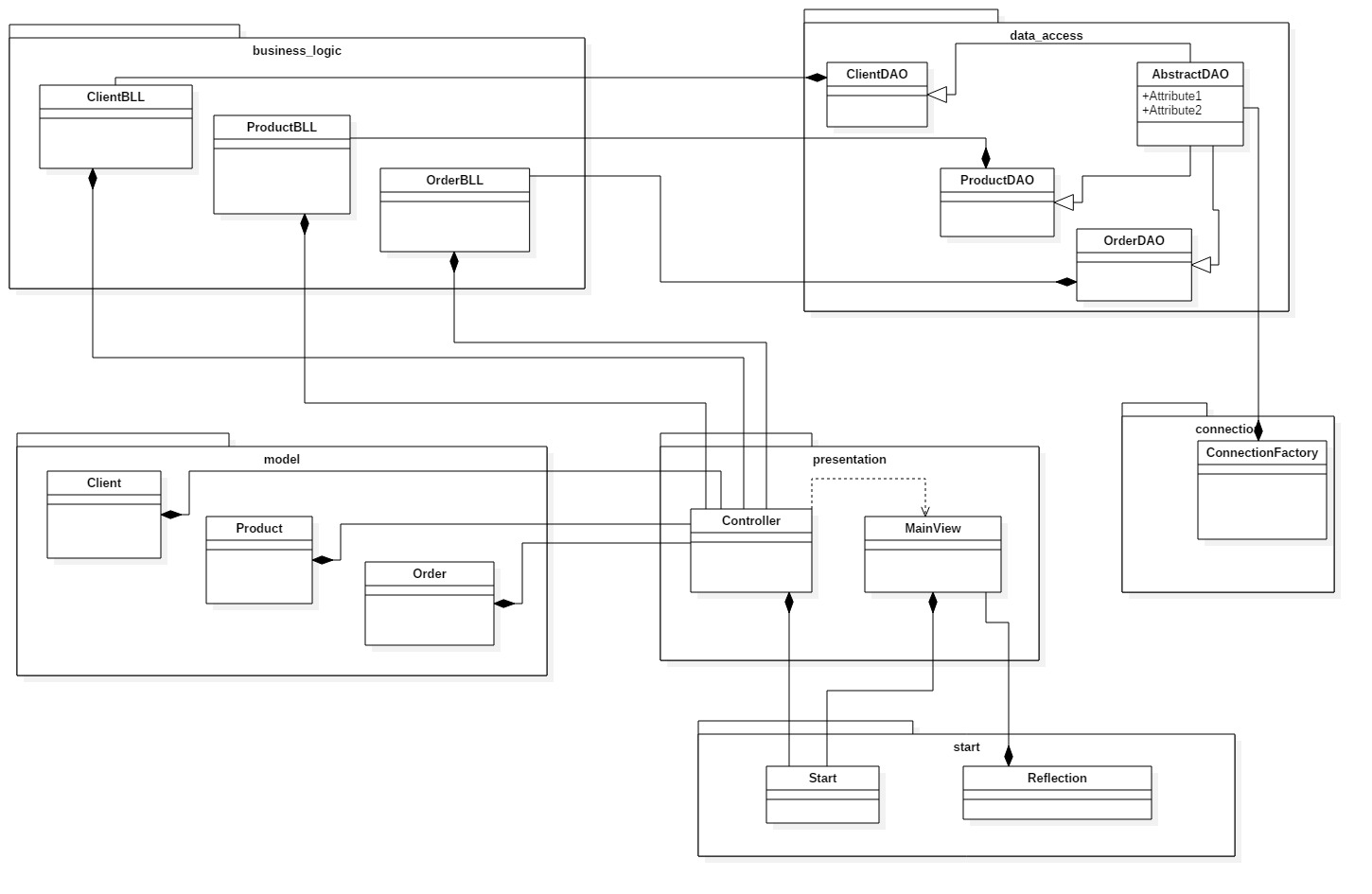
The layered architecture is easy to understand. We have three packages (presentation, businesslogic and dataAccess) that reference the model package. The data access references the connection package, since it uses a database connection to perform the CRUD operations on the table data. The model package models the real-world correspondents of the problem into classes which are easy to work with.

The flow of the 3-tier architecture works as follows: the presentation layer calls the business layer for operations, the business layer then uses the data layer to call the SQL queries, which the connection package connects with the local database. A diagram of the layered architecture will be displayed below.

\*\*\*\*\*\*pune poza de la georgi aici\*\*\*\*\*\*\*\*

3.1 Class Diagram

The class diagram has relationships that highlight the 3-tier architecture. Grouped from top to bottom, we have the business logic, on its right we have the data layer. It is shown how these two layers communicate with each other. Beneath, we have the models, the presentation and the connection layer.



3.2 Decisions

The following paragraphs will depict the decision taken in the design of the relevant classes and the packages to which they belong.

Model

The **client** class models the person which is able to buy a product in real life.

The **product** class models the product to be bought by a real life client.

The **order** class highlights the many-to-many relationship between the clients and the products. Many clients can order many products, and many products can be ordered by one client.

All these classes consist solely on getters and implicit and explicit constructors.

Connection

The **ConnectionFactory** class, in the connection layer, models the database connection of itself and it has as attributes the characteristics of the server. In order to perform the CRUD operations on the database, we need a connection, a statement and a result set. The connection is the application-database communication. The statement is the SQL query applied on the database and the Result Set is the result obtained after applying the SQL query on the database.

The class provides us with methods to create connections and close connections, statements and result sets.

Data Access Level

The Abstract class in this package handles the CRUD (Create, Read, Update, Delete) operations generically on the database. It applies an SQL query on the database table itself and provides our application with the query results from the database. The ClientDAO, OrderDao and ProductDAO classes inherit the AbstractDAO and only provide it with a type to replace the generic type.

As for the algorithms, we will take as an example the “findByID()” method in the abstract class. An SQL statement is applied on a table and a result set is obtained. Then, the result set is parsed and the result is added to a data structure which will be returned by the method.

AbstractDAO is the one who builds the queries with the parameters needed. It uses the principles of Reflection and, by using methods to extract the fields and methods of an entity without having to know their names or order. By extracting the class from a ParameterizedType, it doesn’t have to use the actual name of the class,it needs only an instance of it, passed as a parameter from the business logic level .It uses the ConnectionFactory to establish the link with the database and executes the built queries to do the CRUD operations.

**public** AbstractDAO() {

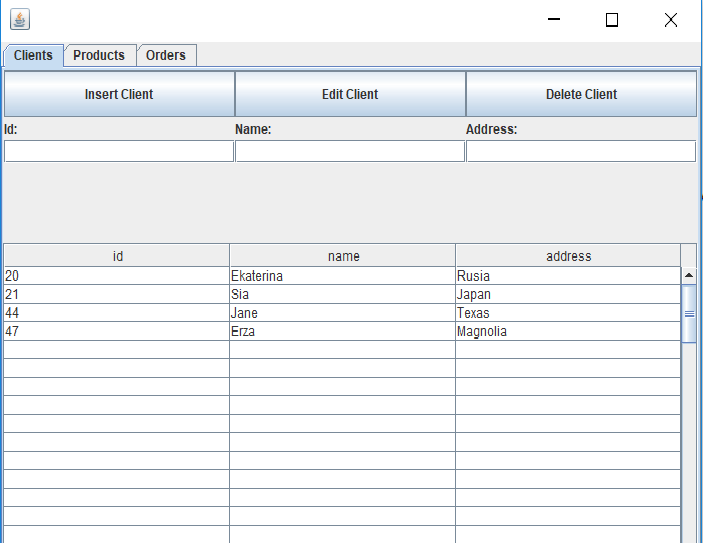
**this**.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0]; }

Business Logic Level

This package deals with the application logic. All its methods are wrappers over the data access methods. For example, the editClient() method in the ClientBLL method calls a ClientDao() update method, which ,through inheritance, is accessed from within the AbstractDAO class.

Presentation

This package contains the view and controller of the application, modeled after the MVC concept. I have a single class for the view, from where all the operations can be done. On the top of the window we have three “labels” which allows us to change what we see: clients, products or tables. The class has various methods for adding button action listeners, getters for the text fields and tables, inserting rows into tables, removing table rows, removing tables and clearing table fields.



The table contents are displayed at all times, so the user can see the existing data. There are text fields for the insert/edit operations, whose contents are taken to the backend when the user presses the corresponding button.

The controller has the purpose to create listeners as inner classes which control what a button does and adds content to the view or updates it as each action happens.

1. Implementation and testing

As this application had to be tested at each step, the testing was done gradually, after each layer was completed. I first tested the database connection and, not surprisingly, it had bugs due to the SQL script used to create the database and the foreign keys. Then, I implemented the flow for a single entity from one end to another, to make sure the CRUD operations work properly and no bugs are met towards obtaining the desired result. Then, I added the view and tested the front end and back end together to see if data is gatherd correctly from the User Interface and the action listeners are implemented properly.

1. Results

My opinion is that my application is easy to be operated by any user, whether he is specialized in the shopping field or not. The user does not need to know any details about the back end implementation, he only needs to input data and press buttons and the window is refreshed on the spot. No ID-s need to be entered by the user, since the database has the id set as an auto incremented field. The whole application is similar to Microsoft office, including the top labels for selecting the pages. It provides the user with tables and ways to add/edit/delete data.

1. Conclusions

What I learned

This application was the first time I used the reflection concept. I did not know it was possible to find out information about an object’s field names in such an easy manner. I have, since then, discovered it is quite an intuitive technique.

I have also learned about the layered architecture and how easy and beautiful it is to divide the classes on packages that suggest what each class does and handles.

Further development

The application can be further developed in quite a few ways. For example, the User Interface could include some dropdown menus for selecting the data, rather than writing it by hand. It could also be more appealing to the user and more eye catching.

Bibliography