DOCUMENTATION

ASSIGNMENT 3

STUDENT NAME: GRAD LAURENȚIU-CĂLIN
GROUP: 30425

Contents

1. Assignment Objective 3
2. Problem Analysis, Modeling, Scenarios, Use Cases 3
3. Design 5
4. Implementation 6
5. Results 6
6. Conclusions 7
7. Bibliography 7

1. Assignment Objective

Consider an application Orders Management for processing client orders for a warehouse. Relational databases should be used to store the products, the clients, and the orders. The application should be designed according to the layered architecture pattern and should use (minimally) the following classes:

- Model classes – represent the data models of the application.
- Business Logic classes – contain application logic.
- Presentation classes – GUI related classes.
- Data access classes – classes that contain access to the database.

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

2. Problem Analysis, Modeling, Scenarios, Use Cases

The assignment asks to implement an application that manages a simple warehouse. It replicates a real-life scenario: any on-line shop must keep track of what products it sells, the available quantity, the clients that regularly buy, their order, and a history of the bills that have been issued. This kind of work is predisposed to creating errors if it is handled by people because of the several things one should keep in mind. However, it is possible to simplify it if an application is designed. A graphical user interface helps even more with the management due to the intuitive interaction between it and an unspecialized user. These are the main guidelines I followed while implementing this assignment. I tried to create an intuitive graphical user interface that does not require computer science knowledge.

The assignment aims to familiarize one with the layered architecture and the reflection techniques provided by the Java programming language. In addition to this, one must work with a relational database management system to ensure data preservation. I chose PostgreSQL.

To store the data (clients, orders, bills, and products), I designed a simple relational database composed of four tables: *client*, *order*, *log*, and *product*. Each table has a primary key called ‘id’. The *order* table has two foreign keys to the *product* and *client* tables because the order is implemented as a one-to-one relation between one client and one product. For simplicity only one product can be part of one specific order, placed by one client, however, the same client can place multiple orders that request different products. This does not resemble a real-life scenario, where one order houses multiple products, but such a functionality is not requested by the text of the assignment.

The *client* relation has four fields: a *name*, a *phone number*, an *address*, and an *id* (unique identification number). These fields are not exhaustive for a client entity, but model accurately what information a shop may require for each person that wants to buy something from them (supposing the shop ships orders remotely).

The *product* relation has four fields, too: a *product name*, a *stock*, a *price per unit*, and an *id* (unique identification number). The fields store only the relevant information that one requires to know when products are inside the warehouse: how they are called, how much of each product is available to sell and what price one unit of that product costs.

The *order* relation has four fields: a *client id*, a *product id*, a *quantity*, and an *id* (unique identification number). The first two fields are linked to the *client* and *product* tables by means of foreign keys. For each order that is placed, an entry in the *log* table is generated, that the assignment calls *bill*. The *log* is linked to the *id* field in the order table (also, by means of a foreign key) and it stores the amount of money the order is worth. The order stores the quantity the client wants to buy of a certain product and when the order is placed, the price of the order is computed and, together with the *id* of the related order, stored in the *log* table.

The database used while developing this project is stored in the same directory as this file in the form of an SQL dump file.

The application is developed and designed to run in a Java Integrated Development Environment such as IntelliJ Idea, Eclipse IDE etc. Once the project is run a pop-up window will appear which is called the *Start Panel* (housed inside the *Main Frame* of the application). Four buttons redirect the user to the specific view that executes the create, read, update, and delete operations on the database. The product, the client and the log views have the same structure: on the left side there are buttons that open the input panels for the CRUD operations. In the right part of the window there is the place where input data is inserted, if necessary. In the lower part of the window the *BACK* button redirects the user to the starting view, while the *EXECUTE* button executes the SQL statement. If the data does not comply with the expected format, the execution of the statements fail, and warnings are displayed in the console. If the execution is successful an info message is displayed in the console, as well. The order view is somewhat special because the panel where the order can be created contains two selectors that let the user choose only between existing clients and products. However, a quantity input field is provided. All input fields must respect hard-coded regex patterns.

NOTE: At any time, the application can be closed if the close button (X) in the top right corner is pressed.

The user diagram describes the dependencies between the user's interactions and the system represented by the Warehouse Management System Application. The user can manage the database as described in the above paragraphs.

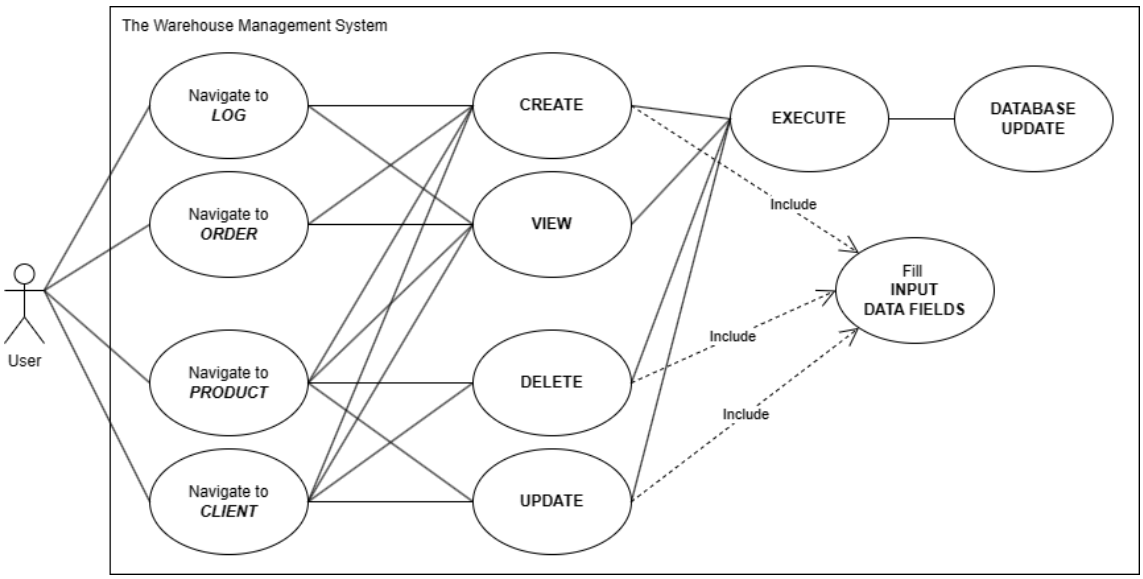

Figure 1. Use case diagram.

3. Design

The application tries to follow the layered architecture pattern. Four packages have been developed: *presentation*, *business logic*, *model*, *data access*. Each package represents a layer, and each one is independent of the others. This aspect favors the modularity and the portability of the application: any layer can be redesigned using different technologies without affecting the other layers if the connections between them are preserved. Further development capabilities are enhanced as well.

The *presentation* package contains all the components of the graphical user interface. The package contains two directories: *utility* and *classes*. The *utility* directory contains classes that are responsible for the swapping between views, sending requests to the *business logic* layer, and general color palette. The classes are not redundant or exhaustive and some could be merged by means of reflection or generics. In the *classes* directory the view objects are placed. These are what the user can see and interact with. To me, their design is intuitive and simple. The structure of the GUI is based on a main frame, that houses more panels due to a card layout. Each such panel is created by smaller panels that store different components such as the operation buttons, the back button, the main panel. The layout of the main panel is also a card layout enabling the application to work in a single window. However, based on the input data a pop-up window may appear and indicate that the operation failed and the cause of failure.

The *business logic* layer stores immutable classes, one for each relation in the database (implemented as Java records). In addition to these, in the *utility* director regex-based validators are placed. Multiple types of validators have been created: id validator, person name validator, address validator, phone number validator etc. They are used to check the input data that is coming from the *presentation* layer. If these input strings represent valid fields of a database relation they are stored into such objects. Following this, they are converted into entities or data transfer objects.

The data transfer objects are part of the *model* package. They represent immutable classes (Java records) that store data that complies with a database relation format (client, order, product, or bill). The difference between them and the classes instantiated in the *business layer* package is that the *model* records store data in the desired format (integer, string, Boolean etc.). The classes in the *business layer* package store data as valid strings.

The *data access* package contains a *utility* directory where the connection to the database object is stored, an enumeration of the possible SQL queries that are used and an abstract class that implements those general CRUD operations/queries. The other directory, called *classes*, stores the specific data abstract object, one for each database relation. Here specific queries such as *find by id* (that is not required to be implemented for all database tables). Other specific queries related to one specific database relation can be instantiated in these DAO classes.

The class diagram is available in the same folder as this file.

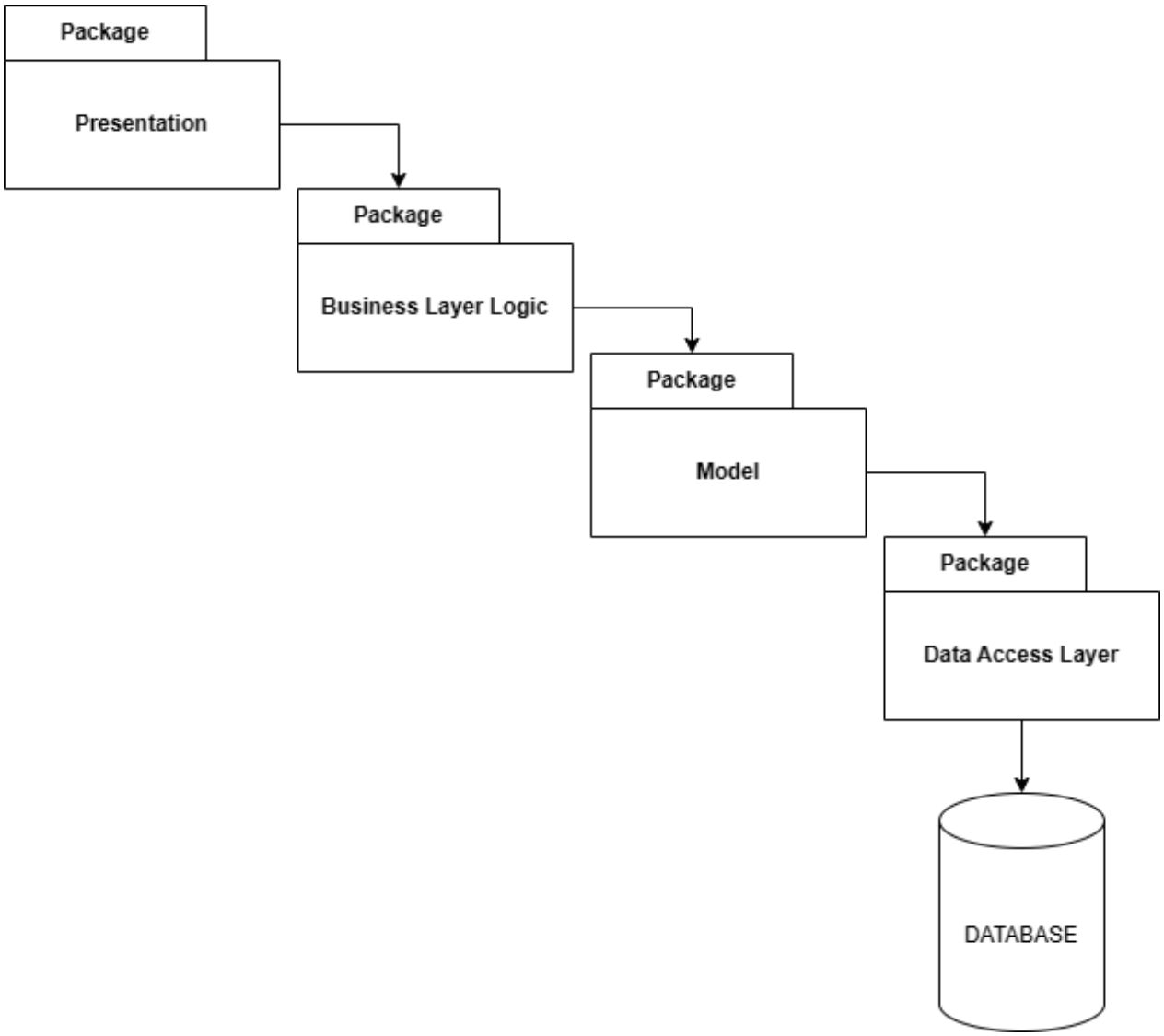

Figure 2. Package diagram.

4. Implementation

The implementation of the classes and of the main methods is discussed in the Javadoc files that are in a folder in the same directory as this file.

5. Results

I have tested the functionality of each operation multiple times. No errors have appeared.

6. Conclusions

This assignment proved to be a lot bigger than I had initially anticipated. The challenge was to break down the application itself into independent modules that can perform more general tasks or are (more or less) open to extension while keeping it simple and easy to understand. The independence of packages granted by the layered architecture was not as difficult to achieve as I have imagined, however, the final project may implement the sinkhole antipattern. I grew more proficient in using the Swing library, I gained more knowledge about how layouts, frames and panels work. I tried to standardize the classes by means of interfaces and I tried to limit the boilerplate code with records and abstract classes. The number of classes I was left with is still big, but there are techniques such as reflection that would let me merge some of them. In this way I can reduce their number with some extra logic.

The final application is not the best it can be for sure. I consider its main defect to be the graphical user interface implementation (the code is a bit chaotic). However, to me, the GUI is simple, intuitive, and practical. The backend structure could be optimized as well.

Other further developments:

- Improved class design for the GUI.
- Better design of the GUI interfaces.
- Better management of the action listeners.
- Improved and simplified inter-class connections.
- Improved, simplified, and more specialized intra-class connections.
- Cleaner, more explicit code style.

7. Bibliography

1. Programming Techniques lectures, laboratories, and support materials
2. Ștefan Tanasă, Ștefan Andrei, Cristian Olaru, Java: de la 0 la expert (2nd edition), Publisher: Polirom, Iași, Romania, ISBN: 978-973-46-2405-8, Published: 2011
3. Paul Deitel, Harvey Deitel, Java How to program (10th edition), Publisher: Pearson Education, Inc., Upper Saddle River, NJ, United States, ISBN: 978-0-13-380780-6, Published: 2015
4. Use Case Diagrams | Unified Modeling Language (UML) - GeeksforGeeks
5. Use Case Diagram Tutorial (Guide with Examples) | Creately
6. Unified Modeling Language - Wikipedia
7. Introduction to JDBC | Baeldung
8. Layers of a Standard Enterprise Application - DZone
9. Creating PDF Files in Java | Baeldung
10. Introduction to JavaDoc | Baeldung
11. Java 14 Record Keyword | Baeldung
12. PostgreSQL | IntelliJ IDEA Documentation (jetbrains.com)
13. Setting up the JDBC Driver | pgJDBC (postgresql.org)
14. Guide to Java Reflection | Baeldung
15. What are Java Records and How to Use them Alongside Constructors and Methods? - GeeksforGeeks