

STORE DATABASE MANAGEMENT APPLICATION

(ASSIGNMENT 3)

Student Name: Cristea Tudor

Group: 30423

CONTENT

1. ASSIGNMENT OBJECTIVE ………………………………………………………... 3
2. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES …………...….. 3
3. DESIGN AND IMPLEMENTATION ……………………………………………..… 5
4. RESULTS …………………………………………………………………………... 10
5. CONCLUSIONS ………………………………………………………………….... 10
6. BIBLIOGRAPHY ………………………………………………………………..… 10
7. ASSIGNMENT OBJECTIVE

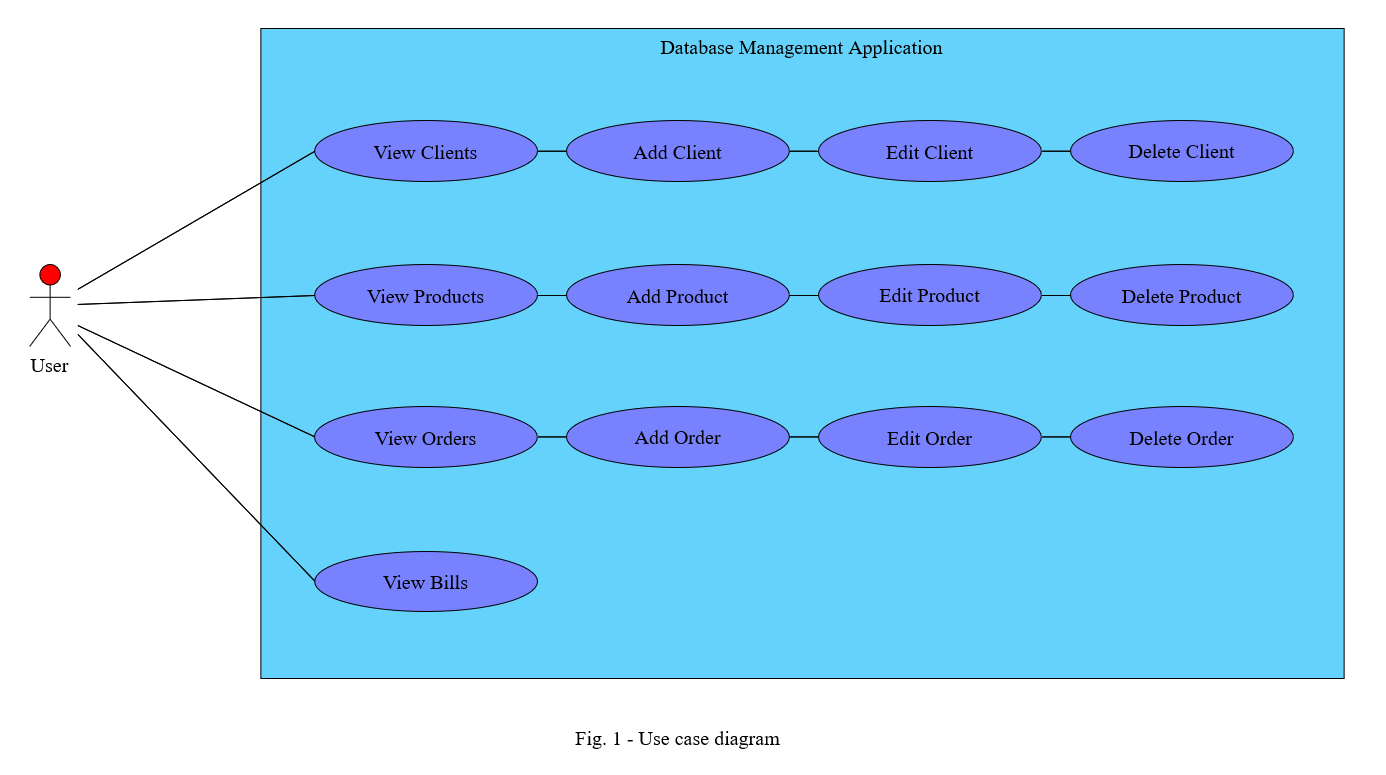
The objective of this assignment is to design and implement a store database management application which gives the user the possibility to manipulate, partially or completely, the tables found in the database. In this sense, the user can insert, update, read or delete entries from certain tables.

The sub-objectives are the following:

* + - * Design and implement several model classes, the objects of which will resemble precisely the structure of their counterpart entities from the tables that exist in the database
* Design and implement dao classes which will ensure the connection and manipulation of the database; these classes will use reflection in order to accomplish their task
* Design and implement the Graphical User Interface (GUI)
* Design and implement the controllers that link the presentation layer and the business layer
* Testing and Debugging possible errors and/or edge-cases

1. PROBLEM ANALYSIS, MODELING, SCENARIOS, USE CASES

The use case diagram is depicted in Figure 1.



**Use case #1: View Clients/Products/Orders/Bills**

**Primary actor: User**

**Success Scenario Steps:**

1. The user presses the button corresponding to the table of objects he wants to see/analyze.
2. The corresponding view (window) opens and the table is shown to the user.

**Alternative Scenario:**

1. The corresponding window does not open
   * + The application informs the user through a pop-up about a potential error (bad database connection)
     + The scenario returns to step 1

**Use case #2: Add Client/Product/Order**

**Primary actor: User**

**Success Scenario Steps:**

1. The user presses the add client/product/order button and a second view (window) opens up
2. The user enters the values of the fields of a new client/product/order in the designated text fields (for an order, the client and product need to be selected using combo boxes)
3. The user presses the “Add” button
4. The data is converted internally from Strings to their actual type and inserted as a new entry into the Client/Product/Order table from the database
5. For an Order, a Bill entity is also created and inserted into the database in the Bill table

**Alternative Scenarios:**

1. The user enters anything other than a positive integer (for product/order quantity), a positive real number (for product price) or an appropriate String (for client phone number, client/product name)
   * + The application informs the user that the text he entered is invalid
     + The client/product/order cannot be added to the database
     + The scenario returns to step 2
2. The user leaves one of the text fields empty

* The application informs the user that there is at least one empty text field
  + - The client/product/order cannot be added to the database
* The scenario returns to step 2

**Use case #3: Edit Client/Product/Order**

**Primary actor: User**

**Success Scenario Steps:**

1. The user presses the edit client/product/order button and a second view (window) opens up
2. The user enters the new values for some or all fields of an existing client/product/order in the designated text fields (for an order, the client and product need to be selected using combo boxes)
3. The user presses the “Edit” button
4. The data is converted internally from Strings to their actual type and the existing entity from the Client/Product/Order table from the database is updated
5. For an Order, a Bill entity is also created and inserted into the database in the Bill table

**Alternative Scenarios:**

1. The user enters anything other than a positive integer (for product/order quantity), a positive real number (for product price) or an appropriate String (for client phone number, client/product name)
   * + The application informs the user that the text he entered is invalid
     + The client/product/order cannot be modified in the database
     + The scenario returns to step 2
2. The user leaves one of the text fields empty

* The application informs the user that there is at least one empty text field
  + - The client/product/order cannot be modified in the database
* The scenario returns to step 2

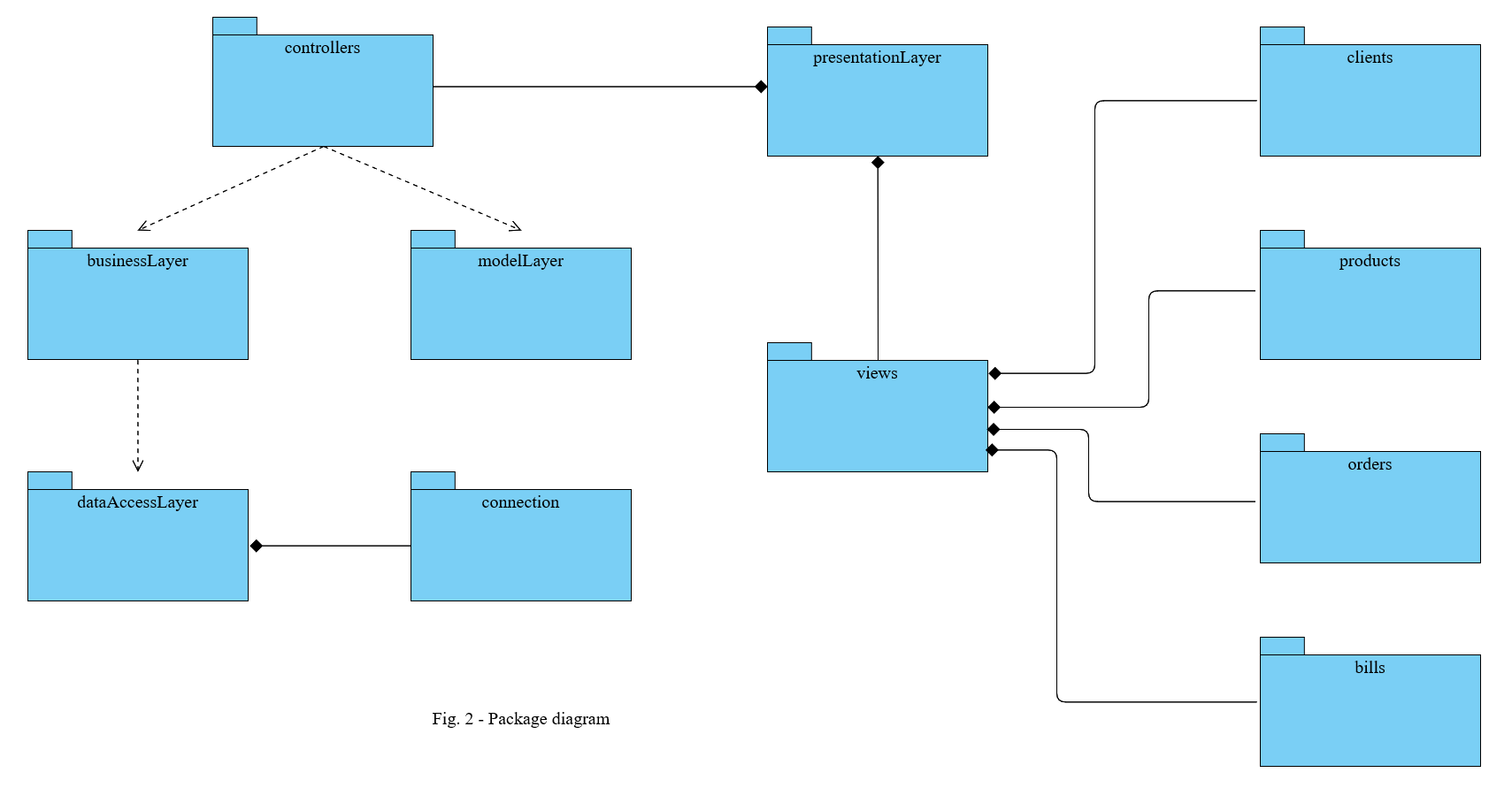
**Use case #4: Delete Client/Product/Order**

**Primary actor: User**

**Success Scenario Steps:**

1. The user presses the delete client/product/order button and a second view (window) opens up
2. The user is shown the current values of the fields of the potentially deleted entity of a Client/Product/Order
3. The user presses the “Delete” button
4. The corresponding entity is deleted from the database
5. DESIGN AND IMPLEMENTATION

The package diagram is depicted in Figure 2. I decided to group my classes based on their functionality, and thus, adopted the Layered Architecture (Business Logic Layer, Model Layer, Presentation Layer and Data Access Layer) pattern for developing this application. This way, the application has a high coherence.

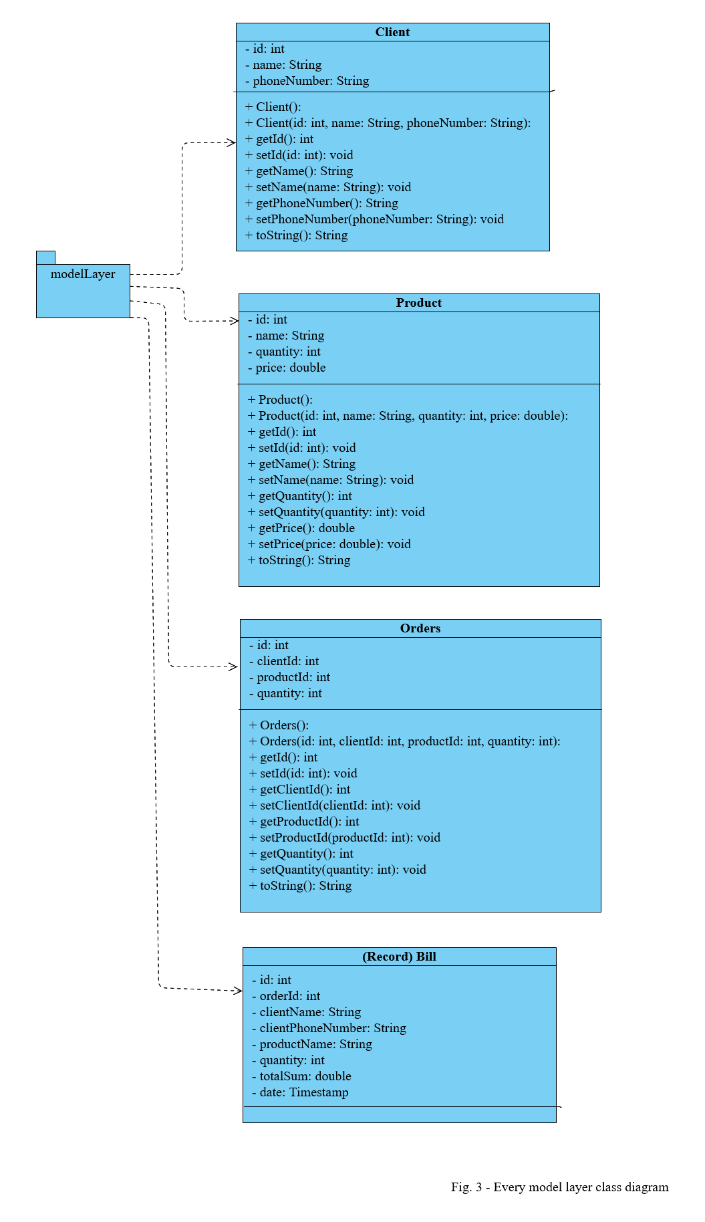


The *Client* class diagram is depicted in Figure 3. Each client has a unique id, a unique name and a unique phone number.

The *Product* class diagram is depicted in Figure 3. Each product has a unique id, a unique name, a quantity and a price, which is per unit of quantity.

The *Orders* class diagram is depicted in Figure 3. I used the plural form of the name for the class because in a database, “order” is a predefined keyword, hence, it cannot be used to define the name of a table, and since use reflection to work with the database, the name of the class and the name of the table have to coincide. Each order has a unique id, the client’s id, the product’s id and a quantity (which should be smaller than the total quantity of the product it references).

The *Bill* class diagram is depicted in Figure 3. It is a record and thus, an immutable object (once it is created, it cannot be modified, and in my application, it will not be deleted either). Additionally, a default constructor containing all fields, mutator methods as well as overridden toString(), hashCode() and equals() methods will be automatically generated. Each bill has a unique id, an order id, the client’s name and phone number, the product’s name, the order’s quantity, a total sum (which is computed by multiplying the unit price with the order’s quantity) and a date (which contains the year, month, day, hour, minute and second corresponding to its creation time).



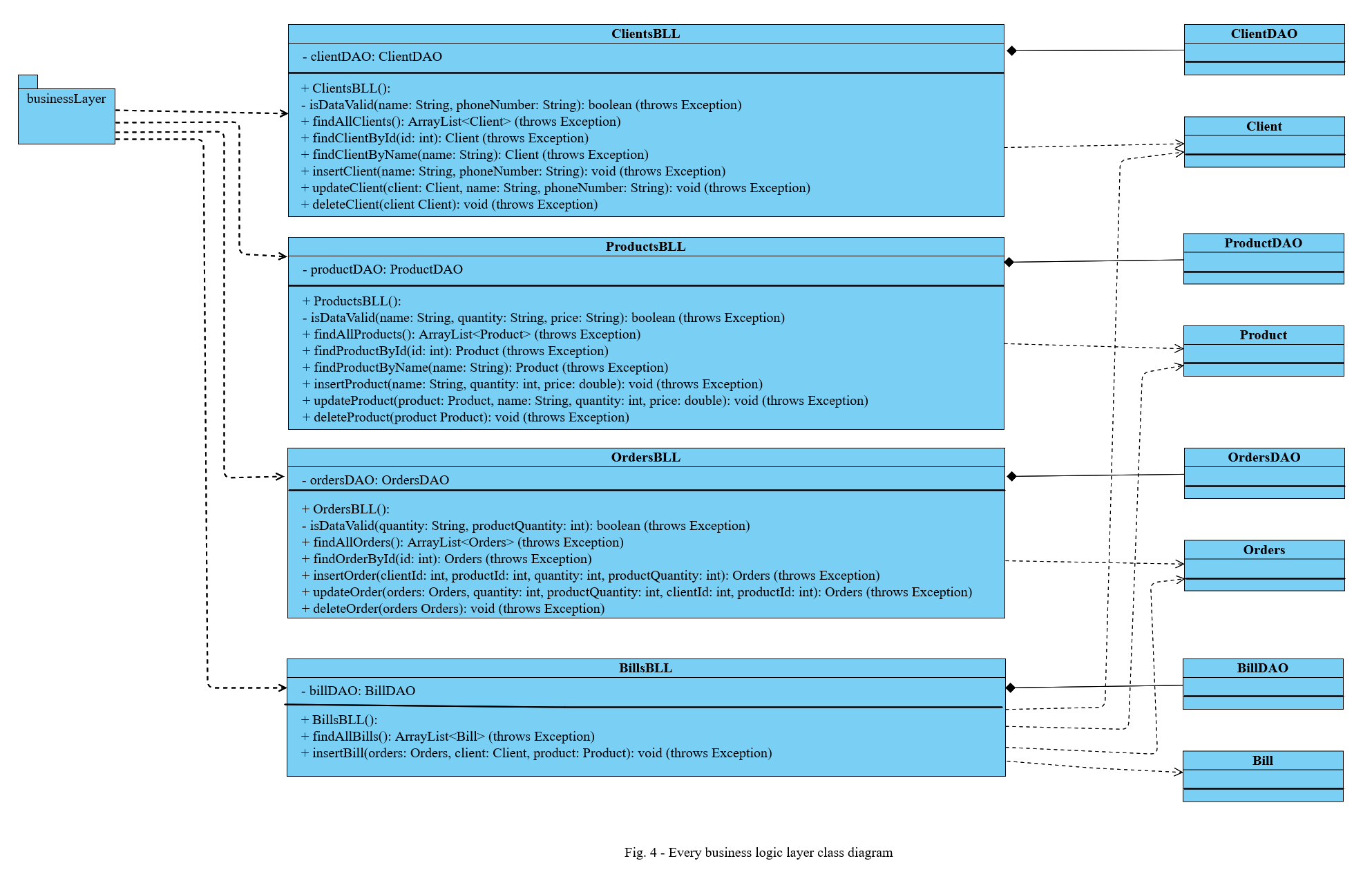
The business logic layer classes are depicted in Figure 4. These classes are responsible with accomplishing the functional requirements. They contain methods that use data access objects to communicate with the database and perform the CRUD operations. The validation of the data, when an object is either created or edited, is performed in these classes.

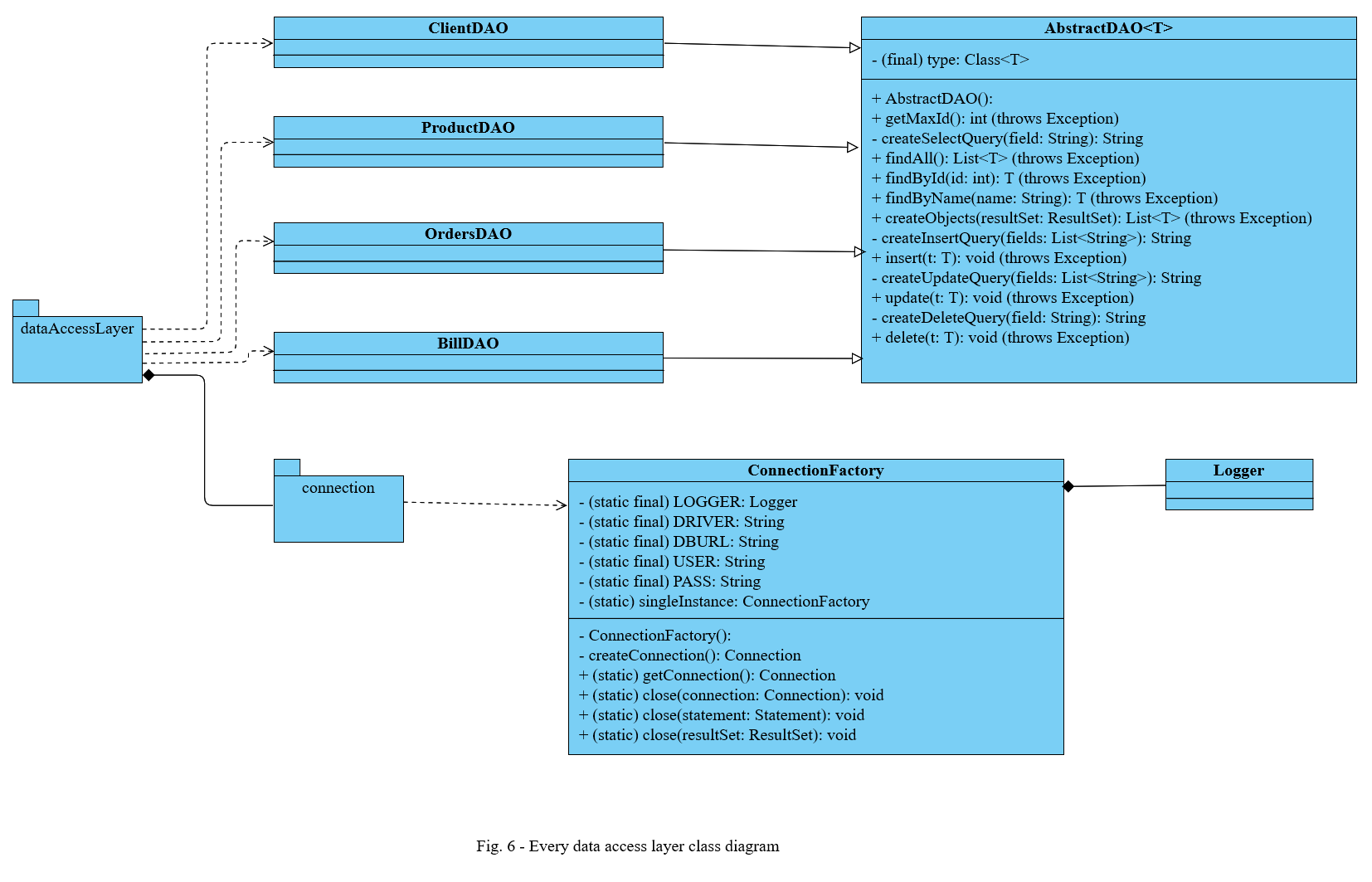
The *MainView* is depicted in Figure 5. It is the view from which the user accesses all the other views. It has four buttons: Clients, Products, Orders and Bills. Pressing any of these will bring up the corresponding view which contains a table where all entries from the database are displayed. From there, the user chooses between adding a new Client/Product/Order, editing an existing one (selected by the user) or deleting an existing one (selected by the user). However, Bills cannot be added, edited or deleted using the application. A new Bill is created whenever an Order is created or updated. Each view has a *showErrorMessage()* method which displays a prompt containing the error message, thus informing the user of the problem that occurred. These errors are either related to the connection with the database or to the fact that the user entered invalid data when creating or editing an object. Additionally, each second view has a *showMessage()* method which contains a positive message, thus informing the user that the operation performed by them was successful. All CRUD (Create, Read, Update and Delete) operations are implemented using reflection. Every second view has *resetButton*s which reset a text field or a combo box to the initial value (for adding, it resets a text field to an empty string and a combo box to the first item; for editing, it resets each of them to their corresponding initial values). Every view’s buttons are a custom version of the JButton, which allows them to have different colors when they are idle, when the mouse hovers over them and when they are pressed. Moreover, every view, besides the main one, have a custom *windowListener* that disables the previous window until the current one (the active one) is closed.

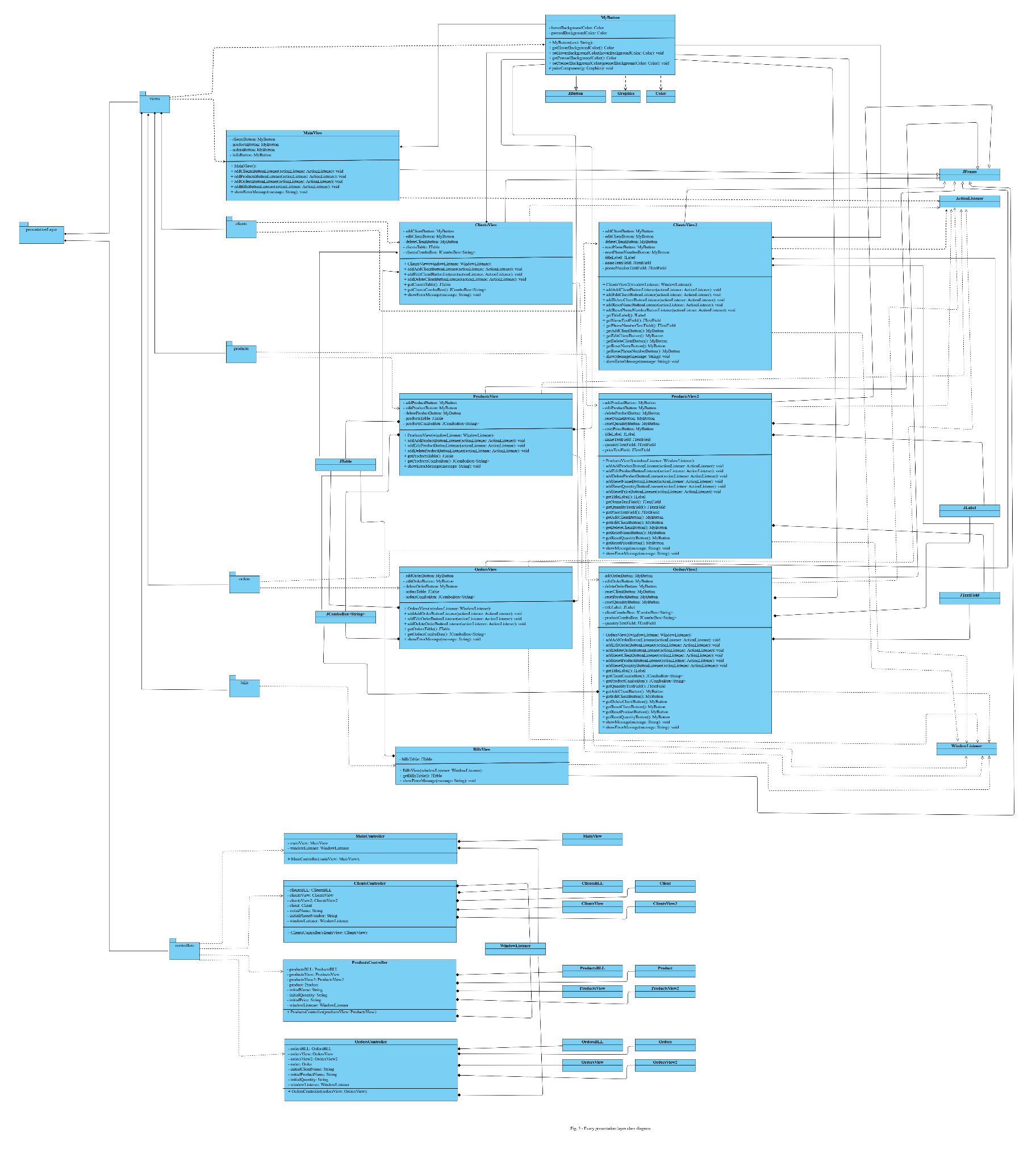
The *MainController* is also depicted in Figure 5. It is responsible with creating all the other controllers which, in turn, realize the connection between the presentation layer (the GUI/views) and the business layer. In addition, every controller contains inner classes that implement the *ActionListener* interface and override the *actionPerfomed()* method so that every button in the application will have their own utility. Since Figure 4 is not very visible, a png file was included in the GitHub repo so that it can be analyzed in more detail (but the inner classes were not added in the diagram for simplicity reasons).

The data access layer classes are depicted in Figure 6. The *AbstractDAO<T>* class is the main class of this layer since it contains all the logic necessary to manipulate the database. In this class, SELECT, INSERT, UPDATE and DELETE queries are created and used to perform the CRUD operations. A database connection is established whenever one of these operations is performed. Since it uses a generic type and all methods are implemented using reflection, all the other dao classes can inherit these methods and be used as such. The constructor establishes the type (class) of the newly created dao by using reflection. For a new client, product or order, the insert method takes the constructor that has no parameters and then uses mutator methods to set the fields afterwards. For a new bill, since it is a record, and therefore, an immutable object (the fields cannot be modified after they are set), the insert method will take the default constructor instead. The *getMaxId()* method is used to get the current maximum id of the entries of a table and after it is incremented, it is used to create a new object. The reasoning behind this decision is the fact that even though, ids can be auto-generated and auto-incremented, in case the object with the biggest id is deleted, the next one inserted will have the value of the id equal to the incremented value of the deleted object’s id. This way, a certain id value is never used anymore. In addition to the *findById()* method, I have also implemented a *findByName()* method, because all names of products and clients in the database are unique.

The *ConnectionFactory* class is also depicted in Figure 6. It creates a singleton object meaning that there will always exist only one *ConnectionFactory* object in the entire application. Moreover, it uses a *Logger* object for debugging purposes.







1. RESULTS

The application was tested by manipulating the database in several different ways. Every operation performed on each table was tested individually as well as together with other operations which might influence the obtained result. No issues were detected during the testing. However, unless the application is properly and fully used in a real manner, by a company for example, further maintenance cannot be performed.

1. CONCLUSIONS

In conclusion, a database management application for a store or a warehouse is really useful because it becomes marginally easier to manage the accountability part of the business. The data is stored in a safer and controlled (restricted) manner if we assume that the user interacts with the database using just this application. In addition, it is easier to manipulate the data.

Working on this assignment, personally I have learned how to integrate a MySQL database into a Java application, how to implement a Java application based on the business layer architecture (how classes should be grouped in order to respect the architecture and thus, have a high cohesion) and how to use generics and reflection for the classes and the methods that work with the database.

Obviously, the application can be developed further with additional features, such as:

* adding two types of accounts: regular user and administrator (each of which having a username and password), the latter having the ability to add fields to tables, add tables and delete bills (or mark them as cancelled)
* add filters, so that only a subset of entries is displayed (alphabetically, by price, by quantity or by date)

1. BIBLIOGRAPHY

* [online.visual-paradigm.com](https://online.visual-paradigm.com) (for diagrams)
* *Fundamental Programming Techniques* course lecture slides and laboratory slides
* [stackoverflow.com](https://stackoverflow.com)
* [chat.openai.com/chat](https://chat.openai.com/chat)
* [geeksforgeeks.org/](http://www.geeksforgeeks.org/)
* [youtube.com](https://www.youtube.com) (for tutorials)