Order Manager

1. Objectives

• Main Objective

The main objective of this homework is the implementation of an order management application. The application should be able to establish a connection with a database, support all CRUD (Create Read Update Delete) operations on the database and establish connections between different tables in the database. The goal of the application is for the user to be able to provide customers, provide products and create orders. Additionally, the application can provide a bill for each client with detailed billing information.

• Secondary Objectives

The main objective can be further divided into smaller objectives. Together they form the application as a whole.

I. Input checking

When inserting, updating, deleting or placing orders, the user has to provide certain input data to the application. To assure a good functioning, the application needs to protect itself against malicious or bad input. Therefore, a through input checking is enforced on the user defined data.

II. Database connection

The application is strongly reliant on a database. The connection to the database needs to be established by means of the JDBC driver.

III. CRUD operations on customers and products

The user needs to be able to insert, update and delete customers and products from the database by only interacting with the applications GUI. Moreover, other operations on the database also need to be provided.

IV. Order placing mechanism

The user needs to be able to place orders directly from the applications GUI. There orders are stored in a specialized database table.

V. Bill generation

The user can generate detailed billing information directly from the applications GUI. The bills are stored in a specialized folder. A bill is created for each customer, containing all products that the customer ordered.

VI. Implementing a Graphical User Interface (GUI)

In order to facilitate the use and interaction with the application, a GUI was implemented. The work of the applications user is greatly simplified by means of an intuitive Graphical User Interface.

2. Problem Analysis

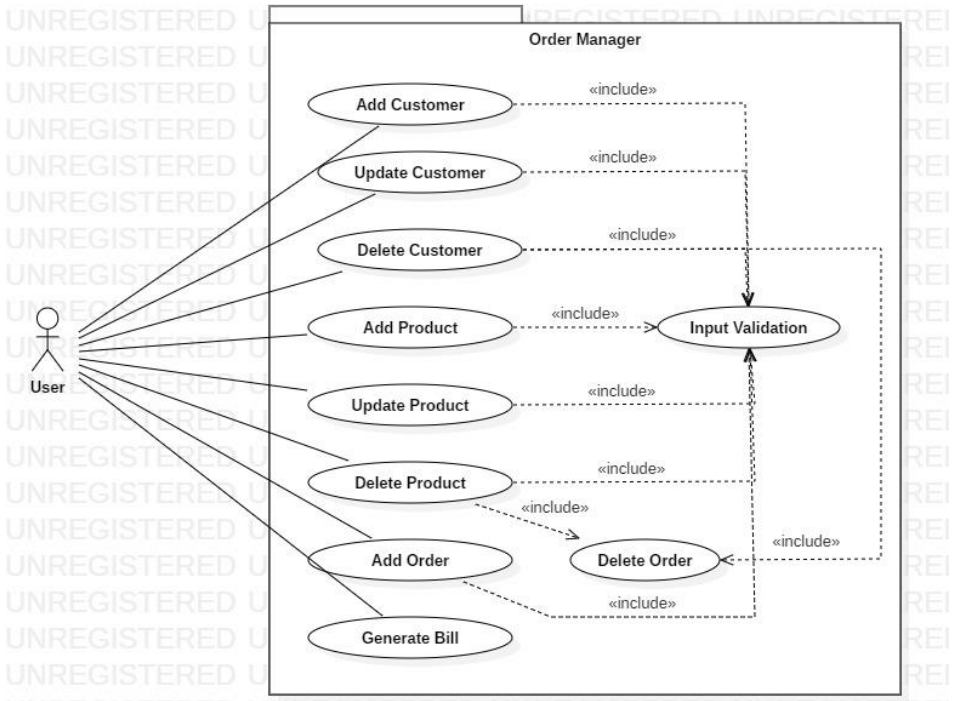
The application should be able to fulfill all the requirements in order to keep track of orders. The products, customers and orders are to be kept in a database. Inserting, updating, deleting and other operations will be executed directly on the database. Therefore, a database connection was developed by means of the MYSQL JDBC driver.

When running the application, all previous records from the database are deleted. The user has to insert both customers and products before orders can be placed. However, inserting data in the database requires the user to provide data to the application. The data needs to be validated before the application can continue. All the necessary conditions for valid data are presented below:

- The fields cannot be empty
- The customer email needs to be a valid email format
- The customer email and address need to be unique
- The product name and stock id need to be unique
- The price and stock of a product need to be positive

After providing all the necessary customer and product information, the user can proceed to place orders. When placing an order, the user will need to provide customer id, product id and quantity. All the necessary conditions for a valid order are presented below:

- The fields cannot be empty
- The customer ID needs to be an integer and be present in the database
- The product ID needs to be an integer and be present in the database
- The quantity needs to be an integer and be less than the stock



The main success scenario is presented below:

Actors	The user
Summary	Allows the user to manage orders
Preconditions	The user provides correct input data
Main success scenario	1) The user provides all the needed input data 2) The application converts text into integers as needed 3) The application checks the data for correctness

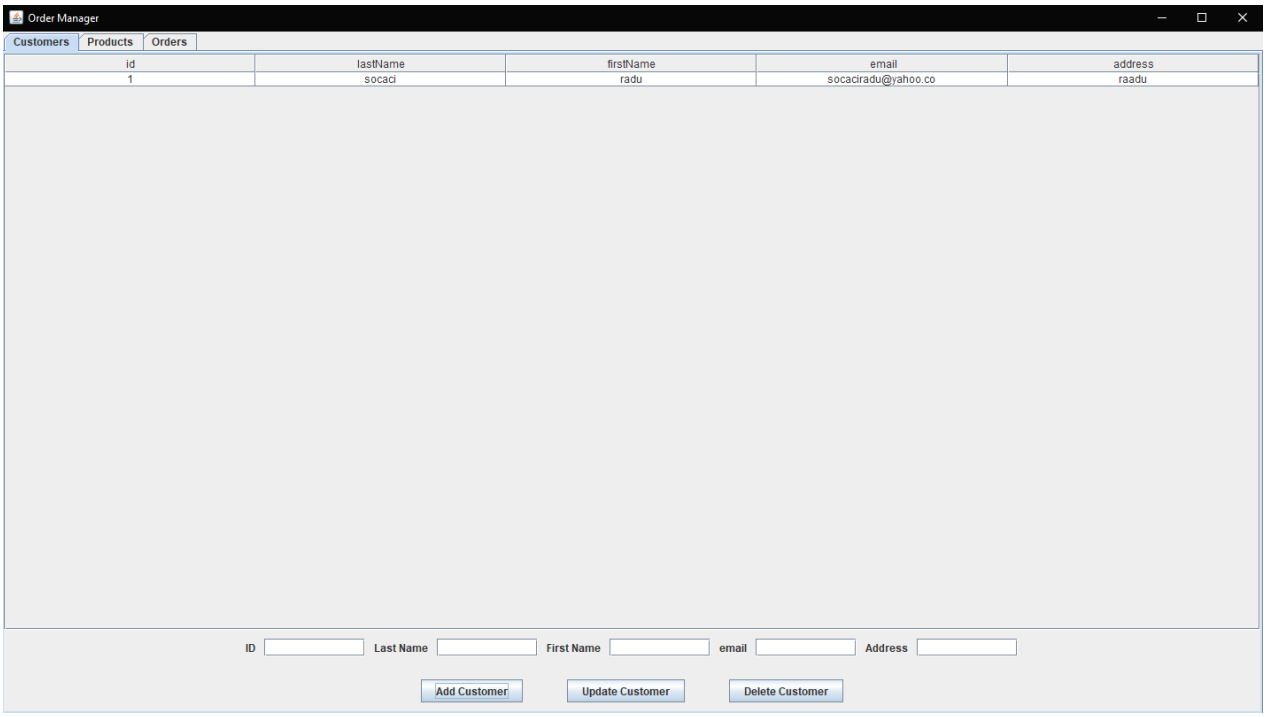
	4) The application inserts customers and products in the database 5) The user switches to the order panel and provides all needed input data 6) The application converts text into integers as needed 7) The application checks the data for correctness 8) The application registers all the orders 9) The user generates bills 10) The application will generate all bills in a special folder

In case the input data in not correctly provided, an alternating sequence occurs:

Actors	The user
Summary	Allows the user to manage orders
Preconditions	The user provides incorrect input data
Alternating sequences	a. Non-integer input: 1) The user provides all the needed input data 2) The application fails to convert text into integers. Return execution to step 1. b. Incorrect data (see above description): 1) The user provides all the needed input data 2) The application converts text into integers if needed 3) Input checking fails. Return execution to 1.

3. Design

From an interface design standpoint, the application combines 3 different panels (customer, product and order) into one, and allows the user to choose at any time which panel he wants to interact with, by means of a tabbed pane.



The application is structured into 4 different layers. Each logical layer has its own package. The layers are: data access, model, presentation and business logic.

From a design standpoint, the application uses a layered architecture. The underlying logic of the application is completely isolated from the GUI itself. The model layer provides classes that strongly relate to the database tables. The data access layer provides a database connection and all the necessary operations to be executed on the data from the database. The business logic layer implements all the necessary logic for the good functioning of the application. Lastly, the presentation layer provides the Graphical User Interface. Utilizing this pattern helps maintaining the application and allows for certain modules to be modified without affecting the other modules.

The application is started by running the main method, situated in the **Main** class. This class is straightforward, and its only purpose is to declare and initialize all the GUI components.

Other straightforward classes are found in the **View** package. This package provides all the view related components, such as the customer panel, product panel and order panel. Each panel contains a JTable which allows for a neat display of the databases contents on the GUI.

These components provide methods for resetting the view, adding action listeners, showing errors, etc.

The link between the logic of the application and the user interface is realized by the **Controller**. This package contains all the GUI controller related classes:

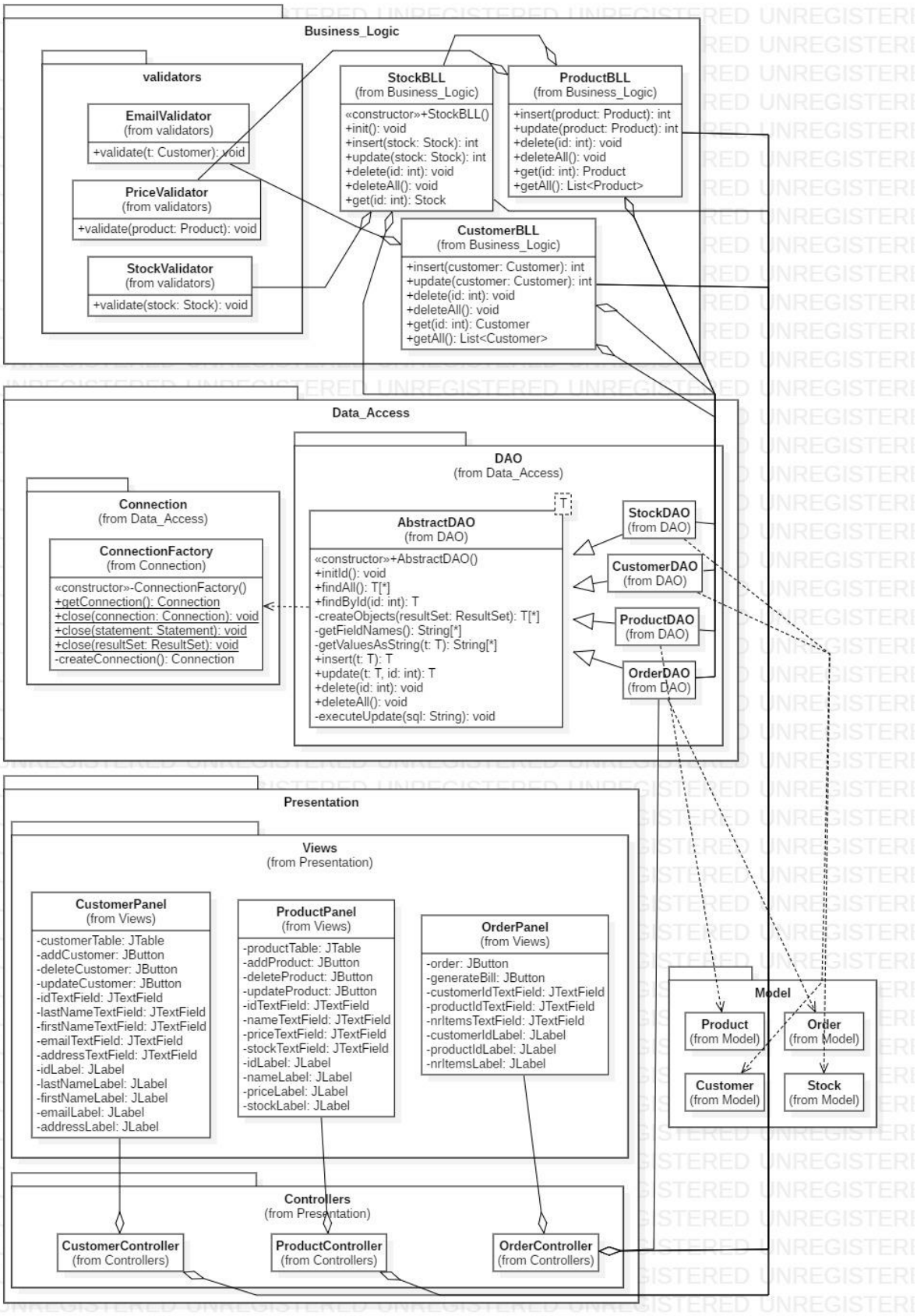
- **Customer Controller**: Provides the functionality for the 3 main operations performed on clients (insert, update, delete). Moreover, this class is tasked with generating an error message in case a failure occurred in the application (bad input, database error, etc.)
- **Product Controller**: Just like the customer controller, this class provides the functionality for the 3 main operations performed on products and handles exceptional cases.
- **Order Controller**: Provides the functionality for placing orders and generating bills. This class can also generate error messages according to the situation at hand. When generating bills, the folder containing the bills is cleared at first and each bill generated corresponds with a customer entry in the database.

The **Model** package contains classes that strongly relate to the actual tables in the database. These classes have similar fields with the database tables themselves and provide getters and setters for each field. Some may even provide a toString method, used to display the billing information.

The **Data Access** package contains the class that implements the database connection and other classes that provide basic CRUD operations for each table in the database. This package defines all queries and provides all the necessary interaction with the database needed for the application.

The **Business Logic** package contains all the validators, that are used for input data validation or database side validation. This package also contains the classes that implement the logic behind the application by combining all the previously defined layers, except for the presentation layer.

A detailed class diagram is presented below. Packages are also described in a pictorial manner.


Business_Logic
validators
EmailValidator
(from validators)
+validate(t: Customer): void
PriceValidator
(from validators)
+validate(product: Product): void
StockValidator
(from validators)
+validate(stock: Stock): void
StockBLL
(from Business_Logic)
«constructor»+StockBLL()
+init(): void
+insert(stock: Stock): int
+update(stock: Stock): int
+delete(id: int): void
+deleteAll(): void
+get(id: int): Stock
ProductBLL
(from Business_Logic)
+insert(product: Product): int
+update(product: Product): int
+delete(id: int): void
+deleteAll(): void
+get(id: int): Product
+getAll(): List<Product>
CustomerBLL
(from Business_Logic)
+insert(customer: Customer): int
+update(customer: Customer): int
+delete(id: int): void
+deleteAll(): void
+get(id: int): Customer
+getAll(): List<Customer>
Data_Access
Connection
(from Data_Access)
ConnectionFactory
(from Connection)
«constructor»-ConnectionFactory()
+getConnection(): Connection
+close(connection: Connection): void
+close(statement: Statement): void
+close(resultSet: ResultSet): void
-createConnection(): Connection
DAO
(from Data_Access)
T
AbstractDAO
(from DAO)
«constructor»+AbstractDAO()
+initId(): void
+findAll(): T[*]
+findById(id: int): T
-createObjects(resultSet: ResultSet): T[*]
-getFieldNames(): String[*]
-getValuesAsString(t: T): String[*]
+insert(t: T): T
+update(t: T, id: int): T
+delete(id: int): void
+deleteAll(): void
-executeUpdate(sql: String): void
StockDAO
(from DAO)
CustomerDAO
(from DAO)
ProductDAO
(from DAO)
OrderDAO
(from DAO)
Presentation
Views
(from Presentation)
CustomerPanel
(from Views)
-customerTable: JTable
-addCustomer: JButton
-deleteCustomer: JButton
-updateCustomer: JButton
-idTextField: JTextField
-lastNameTextField: JTextField
-firstNameTextField: JTextField
-emailTextField: JTextField
-addressTextField: JTextField
-idLabel: JLabel
-lastNameLabel: JLabel
-firstNameLabel: JLabel
-emailLabel: JLabel
-addressLabel: JLabel
ProductPanel
(from Views)
-productTable: JTable
-addProduct: JButton
-deleteProduct: JButton
-updateProduct: JButton
-idTextField: JTextField
-nameTextField: JTextField
-priceTextField: JTextField
-stockTextField: JTextField
-idLabel: JLabel
-nameLabel: JLabel
-priceLabel: JLabel
-stockLabel: JLabel
OrderPanel
(from Views)
-order: JButton
-generateBill: JButton
-customerIdTextField: JTextField
-productIdTextField: JTextField
-nrItemsTextField: JTextField
-customerIdLabel: JLabel
-productIdLabel: JLabel
-nrItemsLabel: JLabel
Model
Product
(from Model)
Order
(from Model)
Customer
(from Model)
Stock
(from Model)
Controllers
(from Presentation)
CustomerController
(from Controllers)
ProductController
(from Controllers)
OrderController
(from Controllers)

4. Implementation

This section provides implementation details for some very important methods/classes.

The application is based on a database. Therefore, the database connection is very important. This connection is realized by a **Connection Factory** class, which uses the singleton design pattern, protecting against multiple definition. The class also provides overloaded methods for closing the connection and getting a new connection.

Another very important aspect is the applications ability to interact with the database directly from the user interface context. The class tasked with providing the database interaction is the **AbstractDAO** class. The class has a generic type parameter, which specifies a class contained in the **Model** layer of the application. The class provides basic CRUD operations on the table referenced by the generic parameter by means of reflection. The generic parameters class is retrieved, and all its fields are computed. This provides a neat way for query construction and allows the same class to implement CRUD operations on any table from the database.

Since the application relies on the user to input data, the data needs to be validated and errors need to be thrown in case of bad data. This task is attributed to the **Business Logic** layer. There are 3 validators, an email validator, which validates the email by means of pattern matching (regex), a price validator and a stock validator (they protect against negative values). In case of an error, this layer is also tasked with throwing the exception to the GUI and returning the control to the user, without performing the operation.

The **Business Logic** layer also provides data integrity and protects against database constraints violation. For example, when removing a client all the orders that the client has in the order table need to be deleted as well, because the client id field in the order table is a foreign key corresponding to the client id from the client table. This layer assures a good functioning and deals with all possible error cases that may occur, both user related and database related.

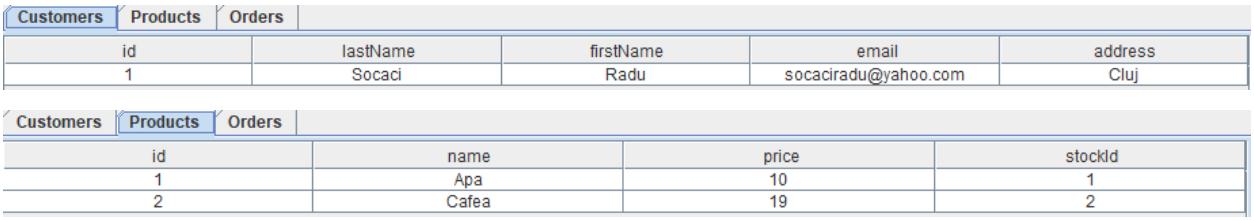
The **Model** layer defines classes that strongly relate to the database tables. These classes have the fields in the same order as the ones in the database, because reflection is used to retrieve the field names and values each time a query needs to be created. Moreover, these classes act like a java representation of a database record. For example, when getting the customer with a

certain id from the database, the returned record is stored in a customer instance and can be used by any portion of the application, directly from a java context.

The design of the GUI was described in section 3. However, the **View** class defines a static method, *createTable*, which given a list of objects, returns a user-friendly representation of the given list as a JTable, ready to be displayed on the GUI. To accomplish this, this method uses reflection techniques for getting the fields and values for each object in the list and inserting each of this data in the JTable. Moreover, this method provides some additional look and feel optimizations for the JTable, such as centered word positioning.

5. Results

This section shows a possible run of the application and its results. The application is very intuitive and easy to use. The only concern for the user is to correctly feed the input data to the application. What "correct" means was defined in section 2.




```
Customer: Socaci Radu
E-Mail: socaciradu@yahoo.com
Address: Cluj

Product Name: Apa       Quantity: 20        Price: 10
Product Name: Cafea     Quantity: 2     Price: 19

Total Payment: 238
```

The first 2 pictures show the initial data that was provided by the user (1 customer and 2 products). After pressing the "generate bills" button, the 3rd picture shows the format of the bill. It is composed of 3 main sections:

- The first part of the bill displays all customer related information
- The second part of the bill displays each product that the customer ordered
- The last section displays the total amount that the customer must pay

6. Conclusions

Nowadays, the e-commerce is everywhere. This means that the need for applications/websites that can interact with a database and allow for customers to easily place orders is continuously increasing. This application provides the building grounds for such a system and can be used as a baseline.

This assignment helped me improve my JDBC related knowledge and practice java reflection.

The application can be further improved. For instance, the billing information can be displayed in a more user-friendly manner, more tables can be added in the database and the user interface can be improved.

7. Bibliography

- https://stackoverflow.com/
- https://www.tutorialspoint.com/jdbc/