

# PROGRAMMING TECHNIQUES

## ASSIGNMENT 3

### ORDER MANAGEMENT SYSTEM

Student: Chindea Miruna  
Group: 30424

# **SUMMARY**

## **1. Homework objective**

- 1.1. Main objective
- 1.2. Secondary objectives

## **2. Problem analysis, scenarios, use cases**

- 2.1. Functional requirements
- 2.2. Use cases

## **3. Design**

- 3.1. Pattern
  - 3.1.1. Presentation Layer
  - 3.1.2. Business Layer
  - 3.1.3. Data Access Layer
- 3.2. Classes and packages
- 3.3. Interfaces
- 3.4. Class Diagram
- 3.5. Relationships

## **4. Implementation**

- 4.1. Classes
  - 4.1.1. Client
  - 4.1.2. Product
  - 4.1.3. Order
  - 4.1.4. Model
  - 4.1.5. Validator classes
  - 4.1.6. ConnectionFactory
  - 4.1.7. ClientDAO
  - 4.1.8. ProductDAO
  - 4.1.9. OrderDAO
- 4.2. Graphical User Interface
  - 4.2.1. View
  - 4.2.2. Controller

## **5. Results**

## **6. Conclusions**

## **7. Bibliography**

# 1. Homework objective

## 1.1. Main objective

The main objective of the homework is to propose, design and implement an **order management application** for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders.

## 1.2. Secondary objectives

The secondary objectives of the homework, which will help in building the main objective, are:

1. Creating a **Graphical User Interface**, in order to help the user interact with it. The user will perform the following operations: add a new client, edit client, delete client, view all clients in a table, add new product, edit product, delete product, view all products in a table.
2. Creating a **product order for a client**: the application user will be able to select an existing product, select an existing client, and insert a desired quantity for the product to create a valid order. In case that there are not enough products, an under stock message will be displayed. After the order is finalized, the product stock is decremented.
3. Using **reflection techniques** to create a method 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 with the values of the elements from the list
4. Creating a **bill** for each order as a text file.
5. Using **relational databases** for storing the data for the application, minimum three tables: Client, Product and Order.
6. Implementing a **Layered Architecture** (at least four packages: dataAccessLayer, businessLayer, model and presentation).

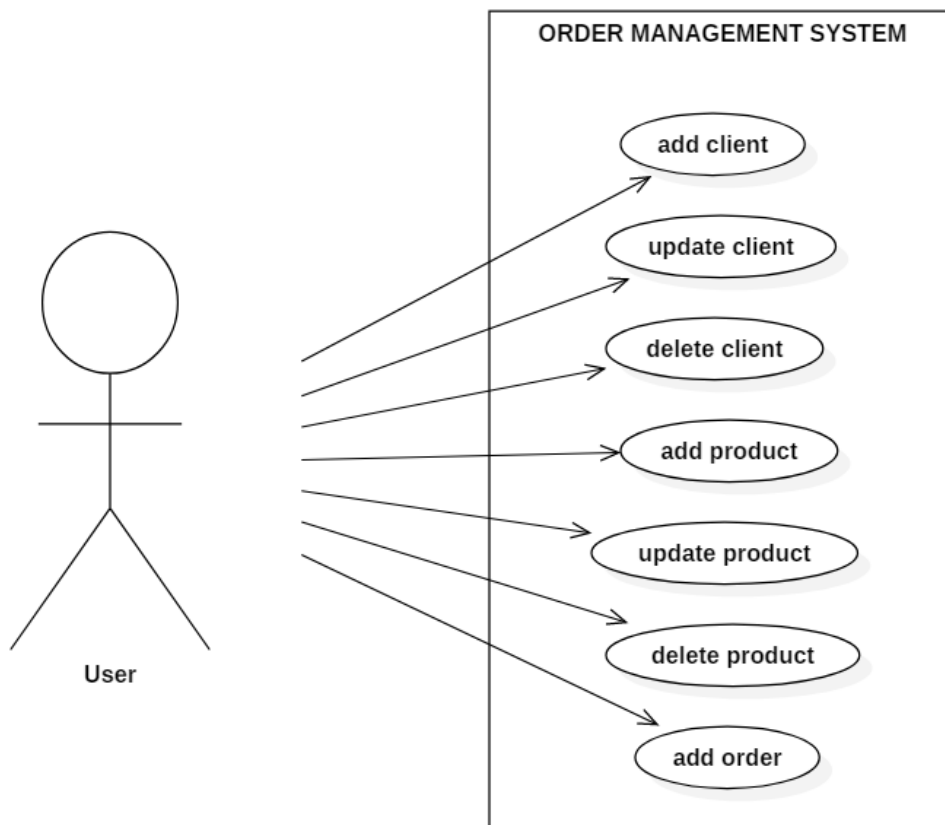
## 2. Problem analysis, scenarios, use cases

### 2.1. Functional requirements

There will be displayed a graphical interface for the user to interact with. The system should successfully implement the specific operations when buttons are pressed, if the user correctly introduces the input. In case the user incorrectly introduces the input, there will be displayed a panel which communicates to the user that the input he introduced is not respecting the pattern, for example if the client mail is invalid or the ordered quantity is smaller or equal to zero.

### 2.2. Use cases

The use cases will be presented and described.



**Fig. 1** Use case diagram

- **Use case title: add client**
  - **Summary:** This use case performs the insertion of a new client in database
  - **Actors:** User
  - **Preconditions:** The user introduces data in all fields
  - **Main success scenario:**
    1. The user introduces client data
    2. The user presses “Add” button
    3. The client is introduced in the database
  - **Alternative sequences:**
    - a) Client e-mail is invalid
      1. The system communicates the problem to the user
      2. The scenario returns to step 1.
    - b) The introduced e-mail is already taken
      1. The system communicates the problem to the user
      2. The scenario returns to step 1.
  
- **Use case title: update client**
  - **Summary:** This use case performs the update of a client
  - **Actors:** User
  - **Preconditions:** The user introduces data in all fields
  - **Main success scenario:**
    1. The user introduces client data
    2. The user presses “Edit” button
    3. The client’s information is updated
  - **Alternative sequences:**
    - a) There is no user with the specified id
      1. The system communicates the problem to the user
      2. The scenario returns to step 1.
    - b) Client e-mail is invalid
      1. The system communicates the problem to the user
      2. The scenario returns to step 1.

- c) The introduced e-mail is already taken
- 1. The system communicates the problem to the user
- 2. The scenario returns to step 1.

- **Use case title: delete client**

- **Summary:** This use case performs the deletion of a client

- **Actors:** User

- **Preconditions:** The user introduces client's id

- **Main success scenario:**

- 1. The user introduces client id
    - 2. The user presses "Delete" button
    - 3. The client is removed from the database

- **Alternative sequences:**

- a) There is no client with the specified id
    - 1. The system communicates the problem to the user
    - 2. The scenario returns to step 1.

- **Use case title: add product**

- **Summary:** This use case performs the insertion of a new product in database

- **Actors:** User

- **Preconditions:** The user introduces data in all fields

- **Main success scenario:**

- 1. The user introduces product data
    - 2. The user presses "Add" button
    - 3. The product is introduced in the database

- **Alternative sequences:**

- a) Product price is smaller or equal to zero
    - 1. The system communicates the problem to the user
    - 2. The scenario returns to step 1.
  - b) Product stock number is smaller or equal to zero
    - 1. The system communicates the problem to the user

2. The scenario returns to step 1.
- c) Product stock number is not an integer
  1. The system communicates the problem to the user
  2. The scenario returns to step 1.

- **Use case title: update product**

- **Summary:** This use case performs the update of a product

- **Actors:** User

- **Preconditions:** The user introduces data in all fields

- **Main success scenario:**

1. The user introduces product data
2. The user presses “Edit” button
3. The product information is updated

- **Alternative sequences:**

- a) There is no product with the introduced id
  1. The system communicates the problem to the user
  2. The scenario returns to step 1.
- b) Product price is smaller or equal to zero
  1. The system communicates the problem to the user
  2. The scenario returns to step 1.
- b) Product stock number is smaller or equal to zero
  1. The system communicates the problem to the user
  2. The scenario returns to step 1.
- d) Product stock number is not an integer
  1. The system communicates the problem to the user
  2. The scenario returns to step 1.

- **Use case title: delete product**
  - **Summary:** This use case performs the deletion of a product
  - **Actors:** User
  - **Preconditions:** The user introduces product's id
  - **Main success scenario:**
    1. The user introduces product id
    2. The user presses "Delete" button
    3. The product is removed from the database
  - **Alternative sequences:**
    - a) There is no product with the specified id
      1. The system communicates the problem to the user
      2. The scenario returns to step 1.
  
- **Use case title: add client**
  - **Summary:** This use case performs the insertion of a new order in database
  - **Actors:** User
  - **Preconditions:** The user introduces data in all fields
  - **Main success scenario:**
    1. The user selects the client
    2. The user selects the product
    3. The user introduces the desired product quantity
    4. The user presses "Add order" button
    5. The order is introduced in the database
    6. A text file with the order summary is created
  - **Alternative sequences:**
    - a) The quantity is not a positive number or an integer
      1. The system communicates the problem to the user
      2. The scenario returns to step 1.
    - b) The quantity exceeds the stock number of the product
      1. The system communicates the problem to the user
      2. The scenario returns to step 1.



## 3. Design

### 3.1. Pattern

The application was split in different layers. The **3-tier architecture** was used in order to build the project. A 3-tier architecture is a type of software architecture which is composed of three “tiers” or “layers” of logical computing: **Presentation Layer**, **Business Layer** and **Data Access Layer**. Each layer has a special purpose and calls functions of the layers below it. The **Model** contains the classes mapped to the database table.

#### 3.1.1. Presentation Layer

The Presentation Layer contains the classes defining the user interface. It is the front end layer in the 3-tier system and consists of the graphical user interface. This user interface is often a graphical one accessible through a web browser or web-based application and which displays content and information useful to an end user.

#### 3.1.2. Business Layer

The Business Layer contains the classes that encapsulate the application logic. This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

#### 3.1.3. Data Access Layer

The Data Access layer contains the classes containing the queries and the database connection. Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

## 3.2. Classes and packages

The classes are structured in packages according to the 3-tier architecture. Using these packages helps in organizing the classes and interfaces.

### 1. model

This package contains the data models of the application. The model classes are the following:

- a) **Client:** the client entity with the same attributes as in the database: id, name, e-mail, address and telephone number
- b) **Product:** the product entity with the same attributes as in the database: id, name, category, brand, price and stock number
- c) **Order:** the order entity with the same attributes as in the database: id, client id, product id, quantity, shipping address and total
- d) **Model:** the model which contains lists of all clients, products and orders

### 2. presentationLayer

This package contains the classes which define the graphical user interface with its functionality. The user interface classes are the following:

- a) **View:** contains the implementation of the GUI and the action listeners
- b) **Controller:** retrieves information from the view and calls the data access functions in order to implement the operations, and then displays the results in the view

### 3. businessLayer

This package contains validator classes for the inputs:

- a) **EmailValidator:** class for validating client's email
- b) **OrderValidator:** class for validating ordered quantity, so that is a positive number
- c) **QuantityValidator:** class for validating product information, so that the price and the stock number are positive numbers

## 4. dataAccessLayer

This package contains the classes which access the database:

- a) **ConnectionFactory:** this class creates a new connection to the database
- b) **ClientDAO:** this class contains queries for getting all the clients, inserting, updating and deleting a client from the database
- c) **ProductDAO:** this class contains queries for getting all the products, inserting, updating and deleting a product from the database
- d) **OrderDAO:** this class contains queries for getting all the orders from the database and inserting a new order in the database

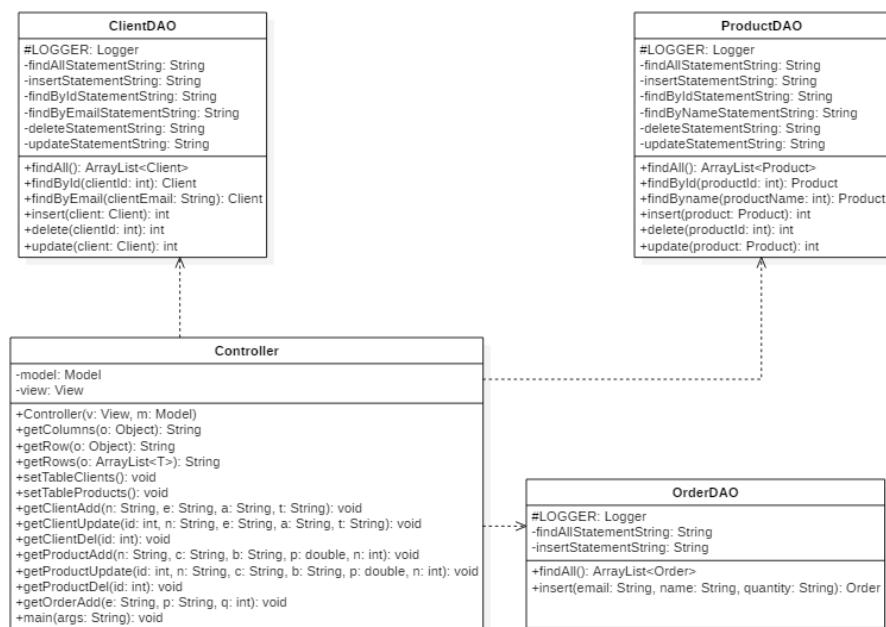
## 3.3. Interfaces

The interfaces that were implemented in this project are:

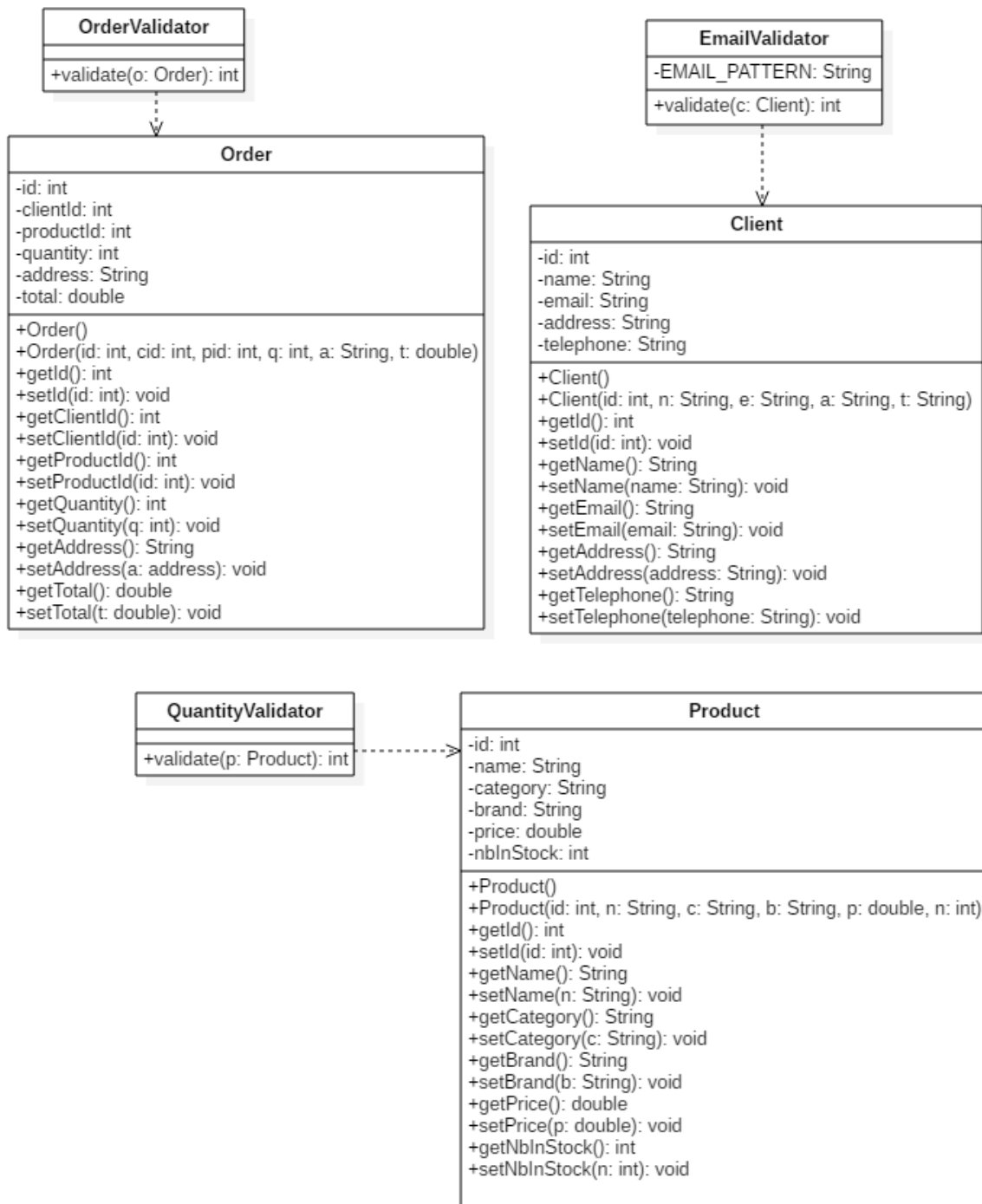
- *Validator:* contains only one method which takes as parameter an object; if the object doesn't match the requirements, the method will return 0 and otherwise, 1.

## 3.4. Class diagram

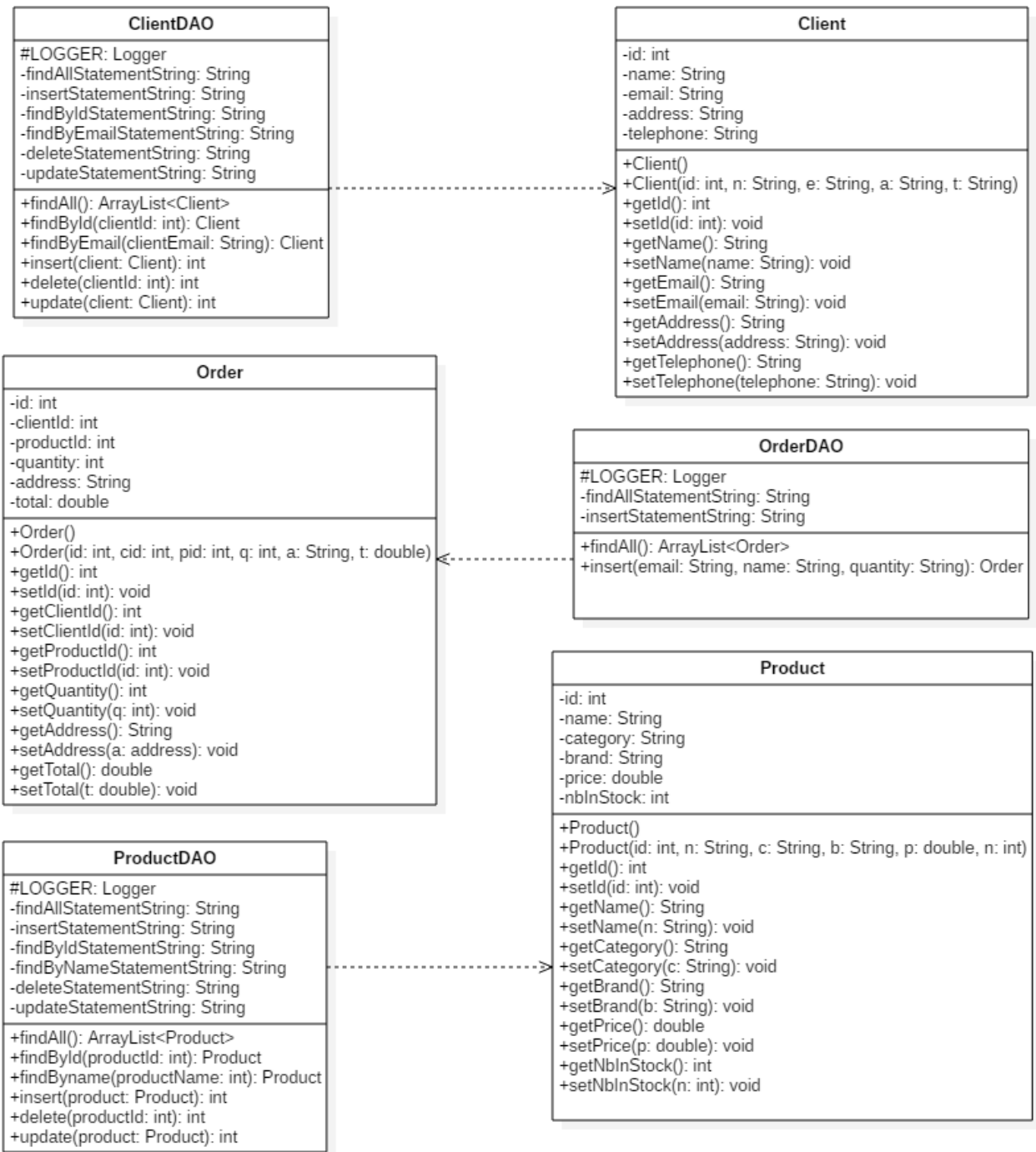
The class diagram was split in 4 pictures to illustrate better the relationship between classes.



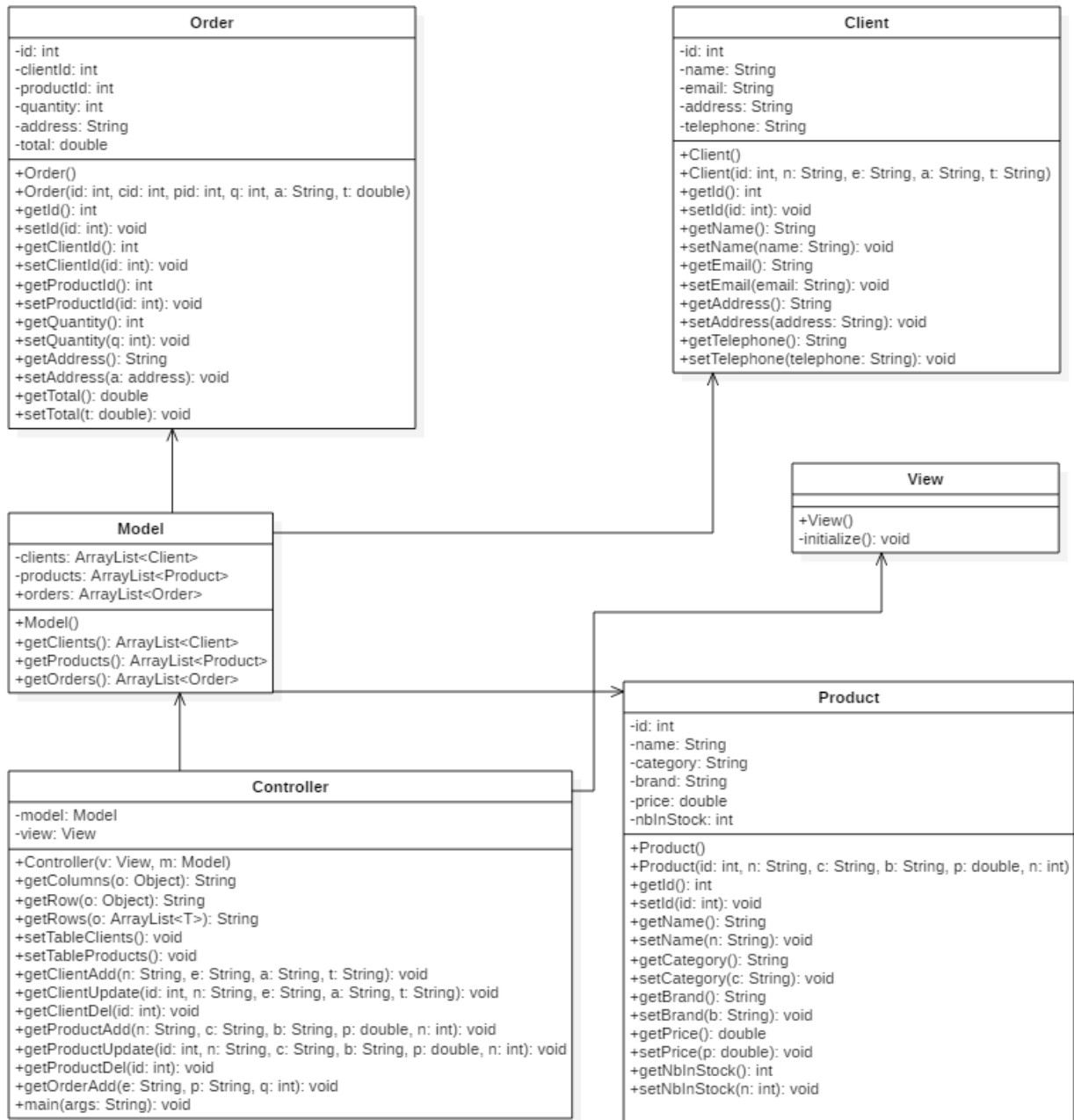
**Fig. 2** Class diagram - Controller and Data Access classes



**Fig. 3** Model classes and validators



**Fig. 4** Model classes and Data Access classes



**Fig. 5** Model classes and presentation classes

### 3.5. Relationships

- **Controller-ProductDAO:** there is a **dependency** relationship between these two classes. The controller class uses the static methods from the other class in order to implement the operations.
- **Controller-ClientDAO:** there is a **dependency** relationship between these two classes. The controller class uses the static methods from the other class in order to implement the operations.
- **Controller-ProductDAO:** there is a **dependency** relationship between these two classes. The controller class uses the static methods from the other class in order to implement the operations.
- **OrderValidator-Order:** there is a **dependency** relationship between these two classes. The OrderValidator class takes an Order object as parameter in it's validate() method.
- **ClientValidator-Client:** there is a **dependency** relationship between these two classes. The ClientValidator class takes a Client object as parameter in it's validate() method.
- **ProductValidator-Product:** there is a **dependency** relationship between these two classes. The ProductValidator class takes a Product object as parameter in it's validate() method.
- **OrderDAO-Order:** there is a **dependency** relationship between these two classes. The OrderDAO class creates new instances of Order class after running the queries.
- **ClientDAO-Client:** there is a **dependency** relationship between these two classes. The ClientDAO class creates new instances of Client class after running the queries.
- **ProductDAO-Product:** there is a **dependency** relationship between these two classes. The ProductDAO class creates new instances of Product class after running the queries.
- **Model-Order:** there is an **association** relationship between these two classes. The Model class has an attribute an array list of Order instances.
- **Model-Client:** there is an **association** relationship between these two classes. The Model class has an attribute an array list of Client instances.
- **Model-Product:** there is an **association** relationship between these two classes. The Model class has an attribute an array list of Product instances.
- **Model-Controller:** there is an **association** relationship between these two classes. The Controller class has as attribute an instance of the Model class.
- **View-Controller:** there is an **association** relationship between these two classes. The Controller class has as attribute an instance of the View class.

## 4. Implementation

In this chapter, there will be rigorously described:

- all the classes with their attributes and important methods;
- the Graphical User Interface.

### 4.1. Classes

#### 4.1.1. Client

This class represents the client entity. A Client instance has the following attributes:

- id: int – client's id
- name: String – client's name
- email: String – client's email
- address: String – client's address
- telephone: String – client's telephone number

#### 4.1.2. Product

This class represents the product entity. A Product instance has the following attributes:

- id: int – product id
- name: String – product name
- category: String – product category
- brand: String – product brand
- price: double – product price
- nbInStock: int – stock number of the product

#### 4.1.3. Order

This class represents the order entity. An Order instance has the following attributes:

- id: int - order id
- clientId: int - client id
- productid: int - product id
- quantity: int - ordered quantity
- address: String - shipping address
- total: double - order total



#### 4.1.4. Model

This class represents the Model entity. A Model entity has the following attributes:

- clients: ArrayList<Client> - list of all clients in the database
- products: ArrayList<Product> - list of all products in the database
- orders: ArrayList<Order> - list of all orders in the database

#### 4.1.5. Validator Classes

All the Validator classes implement the Validator interface, with only one method, which validates an object and returns 1 if the object follows the validation rules, 0 otherwise. The validator classes are the following:

- **EmailValidator** – validates the email of the client
- **OrderValidator** – validates the ordered quantity
- **QuantityValidator** – validates the information about a product

#### 4.1.6. ConnectionFactory

This class makes the connection to the database.

#### 4.1.7. ClientDAO

This class represents a Data Access Object from the 3-tier architecture. This class has as attributes Strings which represent queries for selection, insertion, update and deletion of a client from the database. All its methods are static and perform operations with database. The methods are the following:

- findAll(): ArrayList<Client>
- findById(int clientId): Client
- findByEmail(String clientEmail): Client
- insert(Client client): int
- delete(int clientId): int
- update(Client client): int

#### **4.1.8. ProductDAO**

This class represents a Data Access Object from the 3-tier architecture. This class has as attributes Strings which represent queries for selection, insertion, update and deletion of a product from the database. All its methods are static and perform operations with database. The methods are the following:

- findAll(): ArrayList<Product>
- findById(int productId): Product
- findByName(String productName): Product
- insert(Product product): int
- delete(int productId): int
- update(Product product): int

#### **4.1.9. OrderDAO**

This class represents a Data Access Object from the 3-tier architecture. This class has as attributes Strings which represent queries for selection and insertion of an order in the database. All its methods are static and perform operations with database. The methods are the following:

- findAll(): ArrayList<Order>
- insert(String clientEmail, String productName, int quantity): Order

### **4.2. Graphical User Interface**

The graphical user interface classes represent the Presentation Layer from the 3-tier architecture. Here is where the design of the interface with its functionality is implemented. The Graphical User Interface classes will be presented.

#### **4.2.1. View**

This is the class where we instantiate the frame, panels, tables, text fields, buttons, combo boxes and labels. Also, here are the action listeners defined for the buttons. We have buttons for insertion, updating and deleting a client or a product, and for adding a new order. The combo boxes were used to help the user select the client and the product in order to make the order.

#### 4.2.2. Controller

This is the class where the interface and the model classes are connected. In this class, the information introduced are retrieved from the interface, and methods from the Data Access Layer are called. The Controller class has as attributes a model and a view instance. The methods of this class are:

- `getColumns(Object object): String[]` – this method uses reflection in order to set a table's header
- `getRow(Object object): String[]` – this method uses reflection in order to set a table's row
- `getRows(ArrayList<T> object): String[][]` - this method uses reflection in order to populate a table

The screenshot displays a web application interface with three tabs: "Clients", "Products", and "Orders". The "Clients" tab is active, showing a table with the following data:

id	name	email	address	telephone
1	Chindea Miruna	mirunachindea@yah...	Eroilor 2/45	0747928152
2	Covaci Cristian	covacicristian@yaho...	Republicii 4/287	0776243819
3	Avram Georgiana	avramgeorgiana@g...	Dorobantilor 3/18	0725142829
4	Ionas Emanuela	emaionas@outlook.c...	Bulevardul Bucuresti	0717228371
5	Ana Maria Iordache	aiordache@yahoo.c...	Dorobantilor 4/19	0725373293

To the right of the table, there are three sections:

- Add a new client**: Includes input fields for Name, E-mail, Address, and Telephone, followed by an "Add" button.
- Edit client**: Includes an "Id" input field and input fields for Name, E-mail, Address, and Telephone, followed by an "Edit" button.
- Delete client**: Includes an "Id" input field and a "Dele..." button.

**Fig. 6** Graphical User Interface of the project

## 5. Results

Suppose we want to make an order. We will select client's email and the product we want to order and introduce the quantity.

id	name	category	brand	price	nblnStock
1	Refrigerator XC9...	refrigerators	Daewoo	500.0	30
2	Freezer 68162W...	freezers	Artix	300.99	20
3	Supreme Care F...	laundry machines	LG	600.0	4
4	Dry Cooker 1.0	Pans	Delimano	300.0	24
5	CoffeeMaker 093...	coffee makers	Delonghi	259.0	38
6	CoffeeMaker 093...	coffee makers	Delonghi	259.0	20
7	Dormeo Mattres...	mattresses	Dormeo	10000.0	3

### Make a new order

Client e-mail

covacicristian@yahoo.com

Product name

Freezer 68162WH873

Quantity

5

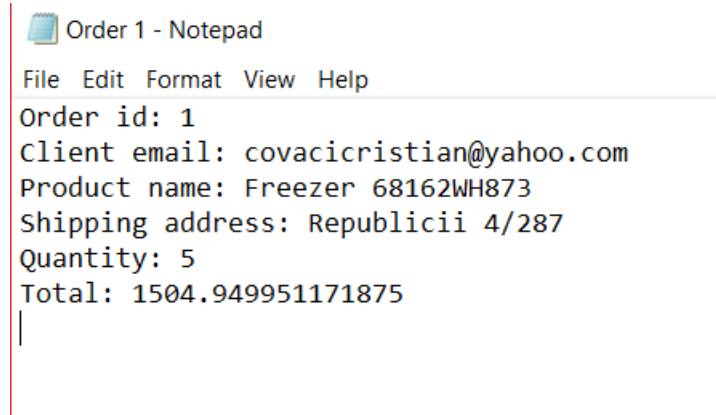
Add order

We observe that the number in stock of the Freezer is 20.

id	name	category	brand	price	nblnStock
1	Refrigerator XC9...	refrigerators	Daewoo	500.0	30
2	Freezer 68162W...	freezers	Artix	300.9899902343...	15
3	Supreme Care F...	laundry machines	LG	600.0	4
4	Dry Cooker 1.0	Pans	Delimano	300.0	24
5	CoffeeMaker 093...	coffee makers	Delonghi	259.0	38
6	CoffeeMaker 093...	coffee makers	Delonghi	259.0	20
7	Dormeo Mattres...	mattresses	Dormeo	10000.0	3

Because the client ordered 5 Freezers, the stock number is now 15.

We also have the order bill as a text file:



```
Order id: 1
Client email: covacicristian@yahoo.com
Product name: Freezer 68162WH873
Shipping address: Republicii 4/287
Quantity: 5
Total: 1504.949951171875
|
```

## 6. Conclusions

In conclusion, the order management system successfully computes the operations if the input is correctly introduced by the user. The user can visualize all the clients and products in tables, insert, update or delete a client or a product, or add a new order. If the inputs are not correctly introduced, the system will display different messages communicating the issues to the user.

I learnt from this project about the 3-tier architecture, about reflection and improved my code writing.

This application can be improved in many ways, for example:

- splitting the system into user and administrator, where user can only add orders and modify its data, and the administrator can modify the products;
- displaying the product with a picture;
- improving the GUI into a more complex and user-pleasing one.

## 7. Bibliography

- <http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>
- <http://theopentutorials.com/tutorials/java/jdbc/jdbc-mysql-create-database-example/>
- <https://dzone.com/articles/layers-standard-enterprise>
- Reflection: <http://tutorials.jenkov.com/java-reflection/index.html>