

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH



**TECHNICAL UNIVERSITY**  
OF CLUJ-NAPOCA

**FACULTY OF AUTOMATION AND COMPUTER SCIENCE**  
**COMPUTER SCIENCE DEPARTMENT**

Fundamental Programming Techniques

Assignment 3

## **ORDER MANAGEMENT**

Teacher: prof. Ioan Salomie  
Teacher Assistant: Cristina Bianca Pop  
Student: Bucur Alexandra  
Group: 30424

2020-2021

## Table of Contents

1. Assignment objective.....	3
2. Problem analysis, modeling, scenarios, use cases.....	4
Analyzing the problem.....	4
Modelling.....	4
Scenarios and Use cases.....	5
3. Design .....	8
Design decisions .....	8
Database.....	9
Relationships, packages .....	9
UML diagrams .....	10
Data Structures.....	11
Class Design.....	11
User Interfaces .....	13
3. Implementation .....	15
Client Class .....	15
Product Class .....	16
Order Class.....	16
Dao.....	17
ClientDao OrderDao ProductDao .....	17
ProductBLL.....	17
OrderBLL.....	17
ClientBLL .....	18
ControllerStart.....	18
ControllerClient .....	19
ControllerProduct.....	19
ControllerOrder.....	20
App Class .....	20
5. Results.....	21
6. Conclusionsa .....	22
Further implementation.....	22
7. Bibliography .....	23

# 1. Assignment objective

## **Main objective**

The main objective is to design an application for processing clients and orders from a warehouse. It should use databases and layered architecture.

Designing an application (OrderManagement) for processing client orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application should be structured in packages using a layered architecture presented in the support presentation. Should use (minimally) the following classes:

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

## **Sub-objectives:**

- Analyze the problem and identify requirements
- Design the queue simulator
- Implement the queue simulator
- Test the queue simulator

### 1. Analyze the problem and identify requirements

This part will analyze the requirements and decide the modelling, scenarios and use cases. As functional requirements the user, which is also the primary actor of the project, should be able to insert all the necessary data for performing the order management and then see the resulted. It will be presented in detail in part 2 of the documentation.

### 2. Design the queue simulator

This part will analyze the design of the order management system. It will decide on a certain design pattern, decide on package configurations and the design of the classes.

It will also present the OOP design of the application, the UML class and package diagrams, the data structures used, the defined interfaces and the algorithms used (if applicable) will be presented.

This part will be presented in detail in part 3.

### 3. Implement the queue simulator

This part will present in detail the implementation of the project. Each class will be described with important fields and methods. The implementation of the user interface will be described. This will be explained later, in part 4.

### 4. Test the order management system

This part will present the result of the testing of the application. It will be presented later, in part 5.

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

### Analyzing the problem

#### General overview

This application should simulate the database of a warehouse or any supermarket, shop with clients and products. Clients should be able to create orders based on the products that are available in the stock, just like in real life. The idea is to discover how working with a database works, by having some clients and orders in tables and performing certain operations in an appropriate interface, designed especially for this scope.

This application should be able to fulfil all the requirements in order to display, modify, and keep track of orders, clients and products. These are stored in a relational MySQL database, along with the information about the users which have access to the system. This way, all the data is easier to retrieve and access from different computers.

### Modelling

The user will be able to select between 3 tables and perform certain operations related to databases, such as insert/delete/update or view. The client can also create orders, based on the existing clients and products that are available.

Based on the input introduced by the user, the result of the chosen operation will be displayed in the interface, in the designated place.

Data that can be seen:

- A table with all the clients/products
- The newly inserted client/product
- The client/product that will have its information changed.

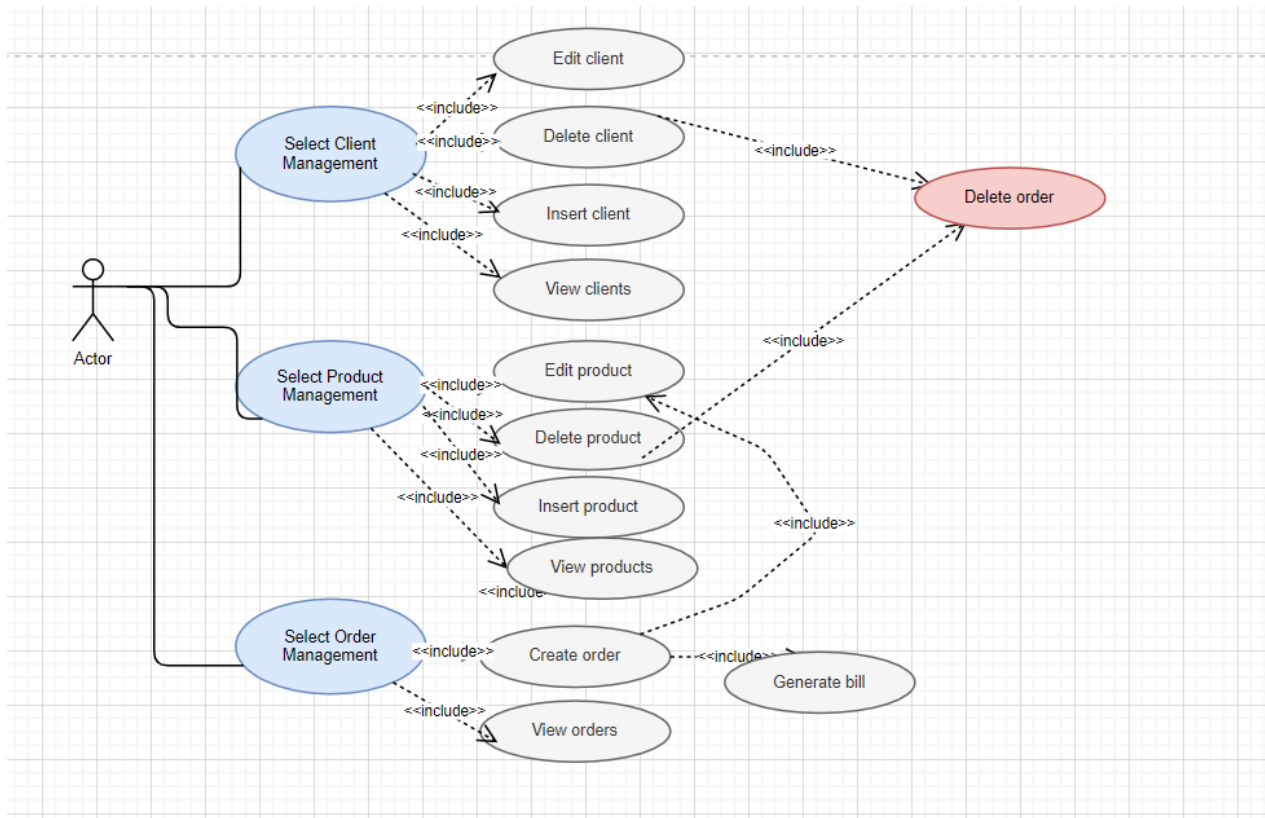
If the data that is introduced is not correct, it will display a proper error message.

#### Input and Output

When talking about the input in the application, the user can choose to manage 3 tables, Client, Product, and Order details. All tables have the options defined as the CRUD operations, Create (insert entry) , Read ( show all entries ), Update (modify entry) and Delete, and the user can introduce the values in the specific fields for any of the four different tables.

For example, for the Client table the user can introduce a new client with all the information, such as the name of the client, the Email address and the phone number. All fields have to obey the standard rules such as phone number has to have 10 digits (only digits ), the name of the client can contain only letters and spaces, the email address has to contain the “@” and the “.” symbols. The rest of the tables contain some constraints also, one would be that the price of a product should only contain numbers.

## Scenarios and Use cases



### Client

#### Insert Client

**Primary Actor:** user

**Main Success Scenario:**

1. The user selects the “Client Management” button
2. . The user selects the “Add client” button
3. The user input all information about a client
4. The user presses the “insert” button

**Alternative Sequence:** Incorrect data

- The user inserts incorrect data (e.g. negative numbers/ letters instead of numbers/wrong phone number)
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be introduced(in step 1)

#### Edit Client

**Primary Actor:** user

**Main Success Scenario:**

1. The user selects the “Client Management” button
2. The user selects the “Edit client” button
3. The user inputs a number, representing the id of the client to be edited
4. The user input all information about a client that wants to be changed
5. The user presses the “update” button

**Alternative Sequence:** Incorrect data

- The user inserts incorrect data (e.g. negative numbers/ letters instead of numbers/wrong phone number)

- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be introduced (in step 1)

## Delete Client

**Primary Actor:** user

**Main Success Scenario:**

1. The user selects the “Client Management” button
2. The user selects the “Delete client” button
3. The user inputs a number, representing the id of the client to be edited
4. The user sees a message telling that the client was deleted

**Alternative Sequence:** Incorrect data

- The user inserts incorrect data
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be introduced (in step 1)

## View all clients

**Primary Actor:** user

**Main Success Scenario:**

1. The user selects the “Client Management” button
2. The user selects the “View all clients” button
3. The user sees all the entries from the database in a table

**Alternative Sequence:** Incorrect data

- There are no clients so an attention message will be displayed

## Product

### Insert Product

**Primary Actor:** user

**Main Success Scenario:**

5. The user selects the “Product Management” button
6. The user selects the “Add product” button
7. The user input all information about a product
8. The user presses the “insert” button

**Alternative Sequence:** Incorrect data

- The user inserts incorrect data (e.g. negative numbers/ letters instead of numbers/wrong phone number)
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be introduced(in step 1)

### Edit Product

**Primary Actor:** user

**Main Success Scenario:**

6. The user selects the “Product Management” button
7. The user selects the “Edit product” button
8. The user inputs a number, representing the id of the product to be edited
9. The user input all information about a product that wants to be changed
10. The user presses the “update” button

**Alternative Sequence:** Incorrect data

- The user inserts incorrect data (e.g. negative numbers/ letters instead of numbers/wrong phone number)
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be introduced (in step 1)

## Delete Product

**Primary Actor:** user

**Main Success Scenario:**

5. The user selects the “Product Management” button
6. The user selects the “Delete Product” button
7. The user inputs a number, representing the id of the product to be edited
8. The user sees a message telling that the product was deleted

**Alternative Sequence:** Incorrect data

- The user inserts incorrect data
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be introduced (in step 1)

## View all products

**Primary Actor:** user

**Main Success Scenario:**

4. The user selects the “Product Management” button
5. The user selects the “View all products” button
6. The user sees all the entries from the database in a table

**Alternative Sequence:** Incorrect data

- There are no products so an attention message will be displayed

## Orders

### View all orders

**Primary Actor:** user

**Main Success Scenario:**

1. The user selects the “Order Management” button
2. The user selects the “View all orders” button
3. The user sees all the entries from the database in a table

**Alternative Sequence:** Incorrect data

- There are no orders so an attention message will be displayed

### Create order

**Primary Actor:** user

**Main Success Scenario:**

1. The user selects the “Order Management” button
2. The user selects a client from the choiceBox
3. The user selects a product from the choiceBox
4. The user inserts the number of products to be ordered
5. The user presses the “create order” button
6. The user sees details about the order
7. The user will see a generated bill

**Alternative Sequence:** Incorrect data

- The user inserts incorrect data (e.g. negative numbers/ letters instead of numbers/wrong phone number)
- An error message will be displayed for the data that was incorrect
- The scenario returns to the current field that has to be introduced (in step 1)

### 3. Design

#### Design decisions

This project uses the layered architecture design.

A Layered Architecture, as I understand it, is the organization of the project structure into four main categories: presentation, application, domain, and infrastructure. Each of the layers contains objects related to the particular concern it represents.

**Presentation layer:** contains all of the classes responsible for presenting the UI to the end-user or sending the response back to the client (in case we're operating deep in the back-end).

**Application layer:** contains all the logic that is required by the application to meet its functional requirements and, at the same time, is not a part of the domain rules.  
In most systems the application layer consisted of services orchestrating the domain objects to fulfill a use case scenario.

**The domain layer:** represents the underlying domain, mostly consisting of domain entities and, in some cases, services. Business rules, like invariants and algorithms, should all stay in this layer.

**The infrastructure layer** (also known as the persistence layer): contains all the classes responsible for doing the technical stuff, like persisting the data in the database, like DAOs, repositories.

There are two important rules for a classical Layered Architecture to be correctly implemented:

1. All the dependencies go in one direction, from presentation to infrastructure. (Well, handling persistence and domain are a bit tricky because the infrastructure layer often saves domain objects directly, so it actually knows about the classes in the domain)
2. No logic related to one layer's concern should be placed in another layer. For instance, no domain logic or database queries should be done in the UI.

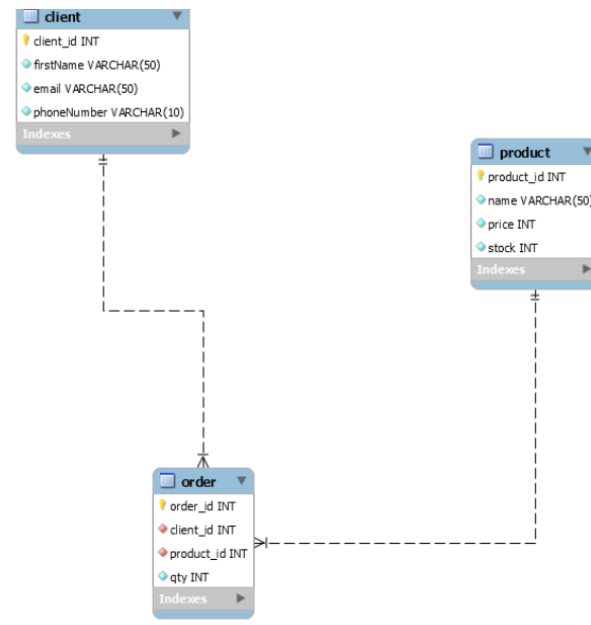
Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database ([Figure 1-1](#)). In some cases, the business layer and persistence layer are combined into a single business layer, particularly when the persistence logic (e.g., SQL or HSQL) is embedded within the business layer components. Thus, smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more layers.

Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a presentation layer would be responsible for handling all user interface and browser communication logic, whereas a business layer would be responsible for executing specific business rules associated with the request. Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the presentation layer doesn't need to know or worry about *how* to get customer data; it only needs to display that information on a screen in particular format. Similarly, the business layer doesn't need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from; it only needs to get the data from the persistence layer, perform business logic against the data (e.g., calculate values or aggregate data), and pass that information up to the presentation layer.



## Database

The database from this project is presented below.



## Relationships, packages

Java packages help in organizing multiple modules and group together related classes and interfaces.

A Layered Architecture is the organization of the project structure into four main categories: presentation, application, domain, and infrastructure. Each of the layers contains objects related to the particular concern it represents.

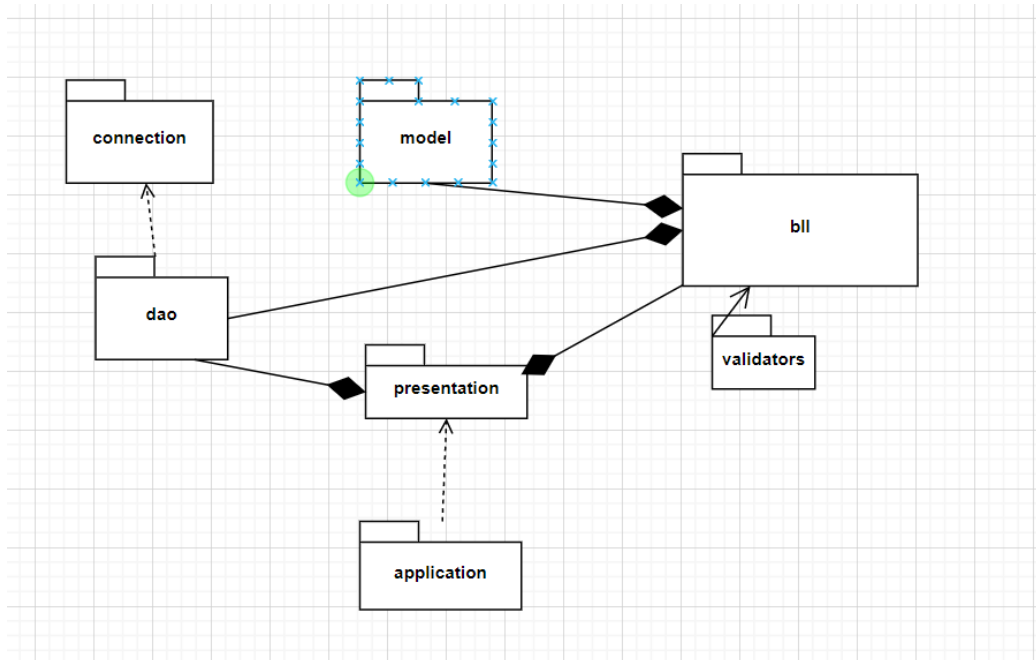
**Presentation layer:** contains all of the classes responsible for presenting the UI to the end-user or sending the response back to the client (in case we're operating deep in the back-end).

**Application layer:** contains all the logic that is required by the application to meet its functional requirements and, at the same time, is not a part of the domain rules.

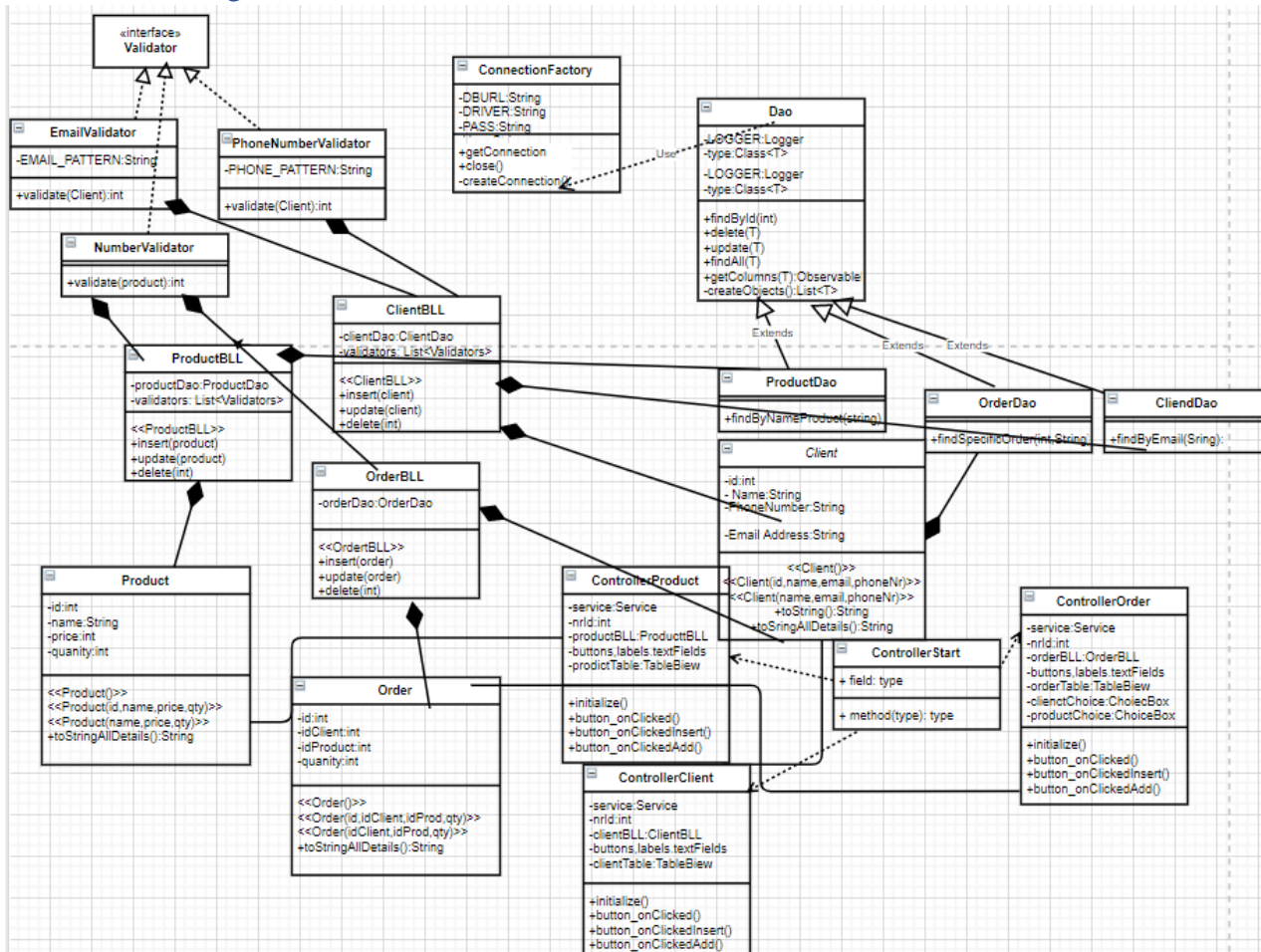
In most systems the application layer consisted of services orchestrating the domain objects to fulfill a use case scenario.

**The domain layer:** represents the underlying domain, mostly consisting of domain entities and, in some cases, services. Business rules, like invariants and algorithms, should all stay in this layer.

**The infrastructure layer** (also known as the persistence layer): contains all the classes responsible for doing the technical stuff, like persisting the data in the database, like DAOs, repositories.



## UML diagrams



## Data Structures

The data structures that have been used in this application are either primitive data types, such as integers or doubles. I used lists and ArrayLists in order to make the code easy to be read and to facilitate the use of memory. The rest of the data used are simple. I use strings in order to create the queries for the database and I mainly use reflection techniques in order to write generic methods for multiple types.

## Class Design

The whole idea of splitting your program into classes is based on a general rule named divide and conquer. The application divides a problem into smaller problems and then those small classes solve the simple and well-known problems.

- 1) The Model package – contains the logic of the application

### Client Class

Fields:

ID: int

Name:string

Email:string

phoneNr:string

Constructors:

– public Client (int ID,String name, String email, String phoneNr) : the constructor that initializes the client with the transmitted fields

– public Client (String name, String email, String phoneNr) : the constructor that initializes the client with the transmitted fields, no id

– public Client () basic constructor

Methods:

-setters and getters

-public void toString() : displays one client on the screen

### Product Class

Fields:

ID: int

Name:string

Price:int

quantityInStock: int

Constructors:

– public Product (int ID, String name, int price, int quantity) : the constructor that initializes the product with the transmitted fields

– public Product (String name, int price,int quantity) : the constructor that initializes the product with the transmitted fields, no id

– public Product () basic constructor

Methods:

-setters and getters

-public void toString() : displays one product on the screen

### Order Class

Fields:

ID: int

idClient:int

idProduct:int

Constructors:

- public Order (int ID,int idClient, int idProduct) : the constructor that initializes the order with the transmitted fields
- public Order (,int idClient, int idProduct) : the constructor that initializes the order with the transmitted fields, no id
- public Order () basic construcor

Methods:

- setters and getters
- public void toString() : displays one order on the screen

## 2) The presentation package

### **ControllerStart Class**

- deals with events for buttons and labels
- has a subclass **ProcessService** that helps displaying labels for a certain amount of time
- based on the clicked button it opens more pages

### **ControllerClient Class**

- deals with events for buttons and labels, but also validating data.
- creates new client objects that will be introduced into the table
- displays the table with all the entries in the screen
- has a subclass **ProcessService** that helps displaying labels for a certain amount of time
- based on the clicked button it opens more pages

### **ControllerProduct Class**

- deals with events for buttons and labels, but also validating data.
- creates new products objects that will be introduced into the table
- displays the table with all the entries in the screen
- has a subclass **ProcessService** that helps displaying labels for a certain amount of time
- based on the clicked button it opens more pages

### **ControllerOrder Class**

- deals with events for buttons and labels, but also validating data.
- creates new order objects that will be introduced into the table
- displays the table with all the entries in the screen
- has a subclass **ProcessService** that helps displaying labels for a certain amount of time
- based on the clicked button it opens more pages

## 3) The Dao(Data access package)

### **Dao class**

- deals with crud operations related to all the tables from the database
- most used/most important method => findById -> it searches in a table by the id
- other methods

Update(T t)

Insert(T t)

Delete(T t)

findAll(T t)

- All these methods perform operations on a specific table, given by the T class

### **ClientDao Class,OrderDao Class,ProductDao Class**

- extend the Dao class
- have more specific queries, such as deleting an entry by the name given or by a phone number

4) The bll(business layer)

(1) Validator package

**Validator Interface**

-interface with only one method : validate(T t)

**PhoneNumber Class**

-implements Validator

-checks if a client have a valid phone number or not

**EmailValidator Class**

-implements Validator

-checks if a client have a valid email or not

**NumberValidator Class**

-implements Validator

-checks if a product have a valid quantity field or not

**ClientBLL class**

```
private List<Validator<Client>> validators;
```

```
private ClientDao clientDao;
```

-class that deals with validating the client objects

-it uses the clientDao class in order to perform the CRUD operations for the Client table

**OrderBLL class**

```
private OrderDao orderDao;
```

-it uses the orderDao class in order to perform the CRUD operations for the Order table

**ProductBLL class**

```
private List<Validator<Product>> validators;
```

```
private ProductDao productDao;
```

-class that deals with validating the product objects

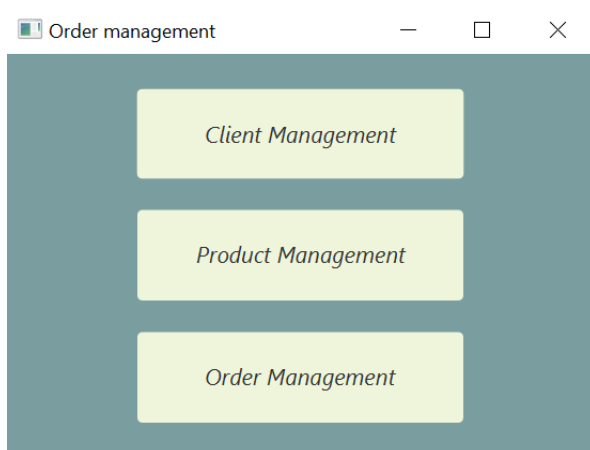
-it uses the productDao class in order to perform the CRUD operations for the Product table

5) The Main package

**App class**

-App class which runs the start() method in order to “turn on” the application.

## User Interfaces



This is the first page that can be seen when running the application. As we can see, the user will be able to choose between those 3 tables related to clients, products or orders.

Here is the client page. The user has already introduced which client will be updated. The rest of the operations are performed similarly.

As for the clients table, here we can see all the products that are available in the database, with all the information.

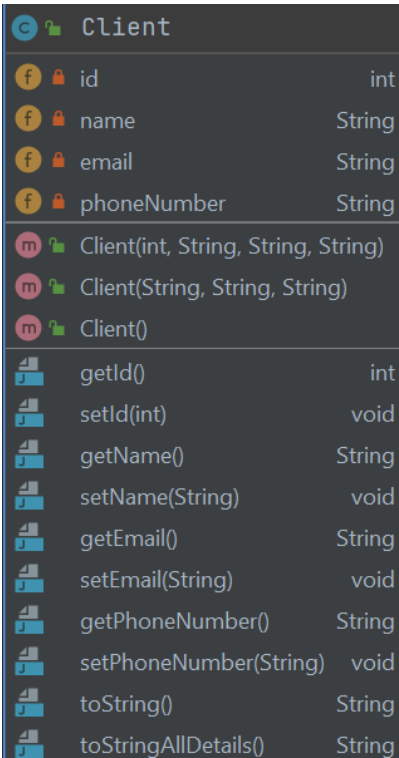
id	name	price	nrProducts...	
1	Phone	1499	222	
2	TV	299	1000	
3	Fridge	12300	22	
4	Tablet	599	60	
5	Laptop	1999	200	
6	Calculator	1699	3	

The table with all the products.

Here, for the database table, a client can create a new order based on the clients and products available.

### 3. Implementation

#### Client Class



	<b>Client</b>	
	id	int
	name	String
	email	String
	phoneNumber	String
	Client(int, String, String, String)	
	Client(String, String, String)	
	Client()	
	getId()	int
	setId(int)	void
	getName()	String
	setName(String)	void
	getEmail()	String
	setEmail(String)	void
	getPhoneNumber()	String
	setPhoneNumber(String)	void
	toString()	String
	toStringAllDetails()	String

The client class will be a simple class with getters and setter

It will have 3 constructors, used in different places.

It will also have a toString method that will display on the screen one client

The method toStringAllDetails is used for the bill.txt that will show all information about a client

## Product Class

Product		
f	id	int
f	name	String
f	price	int
f	nrProductsInStock	int
Constructors		
m	Product(int, String, int, int)	
m	Product()	
m	Product(String, int, int)	
Getters and Setters		
m	getId()	int
m	setId(int)	void
m	getName()	String
m	setName(String)	void
m	getPrice()	int
m	setPrice(int)	void
m	getNrProductsInStock()	int
m	setNrProductsInStock(int)	void
m	decreaseProductsInStock(int)	void
m	toString()	String
m	toStringAllDetails()	String

The product class will be a simple class with getters and setter

It will have 3 constructors, used in different places, such as inserting a new product into the database.

It will also have a toString method that will display on the screen one product

The method toStringAllDetails is used for the bill.txt that will show all information about a product

The decreaseProductsInStock() will decrease the number of products available in the table once an order is being placed.

## Order Class

Order		
f	id	int
f	idClient	int
f	idProduct	int
f	quantityOfProduct	int
Constructors		
m	Order(int, int, int, int)	
m	Order(int, int, int)	
m	Order()	
Getters and Setters		
m	getId()	int
m	setId(int)	void
m	getIdClient()	int
m	setIdClient(int)	void
m	getIdProduct()	int
m	setIdProduct(int)	void
m	getQuantityOfProduct()	int
m	setQuantityOfProduct(int)	void

The order class will be a simple class with getters and setter

It will have 3 constructors, used in different places, such as when inserting a new order



## Dao

Dao	
LOGGER	Logger
type	Class<T>
Dao()	
createSelectQuery(String)	String
findById(int)	T
createObjects(ResultSet)	List<T>
insert(T)	void
findAll()	List<T>
update(T)	void
getColumns(T)	ObservableList<TableColumn<T, ?>>
delete(T)	void

The Dao class deals with CRUD operations in a database. It uses the reflection techniques in order to allow multiple types of objects to be inserted into different table s=> the corresponding table.

The getColumns method will return a list with all the columns that are needed for the table that will be seen on the UI.

The rest of the methods will perform operations on the designated table.

For example, the findById() method will create a sql query for the table in order to show the searched object with the corresponding id

## ClientDao OrderDao ProductDao

This classes extend the Dao class and make it more specific. They also contain some specific queries that only exist for the corresponding table. The classes can be developed with more sql queries depending on the project.

ProductDao	OrderDao	ClientDao
deleteSpecificOrderProduct(String) void	deleteSpecificOrder(int, String) void	deleteSpecificOrderClient(String) void















## ProductBLL

ProductBLL	
validators	List<Validator<Product>>
productDao	ProductDao
ProductBLL()	
findProductById(int)	Product
update(Product)	boolean
insert(Product)	boolean
showAll()	List<Product>
getCols()	ObservableList<TableColumn<Product, ?>>
delete(int)	boolean

This class contains all the validators needed for a product object in order to be considered correct. The constructor will initialize the productDao and add all the validators.

In the update and insert methods some additional verification is performed in order to see if the data that is introduced is correct. The rest of the methods will only perform operations with the help of the dao classes.

## OrderBLL





















		OrderBLL	
		orderDao	OrderDao
		OrderBLL()	
		insert(Order)	void
		showAll()	List<Order>
		getCols()	ObservableList<TableColumn<Order, ?>>
		delete(int)	boolean

This class contains all the validators needed for a client object in order to be considered correct. The constructor will initialize the clientDao and add all the validators.

In the update and insert methods some additional verification is performed in order to see if the data that is introduced is correct. The rest of the methods will only perform operations with the

help of the dao classes.











## ClientBLL

		ClientBLL	
		validators	List<Validator<Client>>
		clientDao	ClientDao
		ClientBLL()	
		findClientById(int)	Client
		update(Client)	boolean
		insert(Client)	boolean
		showAll()	List<Client>
		getCols()	ObservableList<TableColumn<Client, ?>>
		delete(int)	boolean

This class contains all the validators needed for a client object in order to be considered correct. The constructor will initialize the clientDao and add all the validators.

In the update and insert methods some additional verification is performed in order to see if the data that is introduced is correct. The rest of the methods will only perform operations with the help of the dao classes.

## ControllerStart

		ControllerStart	
		clientButton	Button
		orderButton	Button
		productButton	Button
		button_OnClicked(ActionEvent)	void

The main controller that will deal with operations of the first page that is shown at runtime. It only has 1 method that deals with opening the “selected” new page: Client page, Order page, Product page.

## ControllerClient

ControllerClient		
m	initialize()	void
m	button_onClicked(ActionEvent)	void
m	button_onClicked_editClient(ActionEvent)	void
m	handleKeyReleased()	void
m	errors(String)	void
m	checkInput(TextField)	boolean
m	button_onClickedUpdate(ActionEvent)	void
m	button_onClickedAdd(ActionEvent)	void
m	button_onClickedAddClient(ActionEvent)	void
m	button_onClickedView(ActionEvent)	void
m	button_onClickedDelete()	void
m	handleKeyReleasedDelete()	void

This controller deals with the client page. It will perform operations in the back as the user clicks on some buttons or inserts some data. Here, the data will also be checked, for example in order to delete a client the id of the client has to be a number. The rest of the methods will deal with buttons designated for inserting, updating or showing all the clients.

## ControllerProduct

ControllerProduct		
m	initialize()	void
m	button_onClicked(ActionEvent)	void
m	button_onClicked_editProduct(ActionEvent)	void
m	handleKeyReleased()	void
m	errors(String)	void
m	checkInput(TextField)	boolean
m	button_onClickedUpdate(ActionEvent)	void
m	button_onClickedAdd(ActionEvent)	void
m	button_onClickedAddProduct(ActionEvent)	void
m	button_onClickedView(ActionEvent)	void
m	button_onClickedDelete()	void
m	handleKeyReleasedDelete()	void

This controller deals with the product page. It will perform operations in the back as the user clicks on some buttons or inserts some data. Here, the data will also be checked, for example in order to delete a product the id of the product has to be a number. The rest of the methods will deal with buttons designated for inserting, updating or showing all the products. It will also take all the products from the corresponding table at the beginning, when the page is created, in the initialize() method.

## ControllerOrder

ControllerOrder		
m	initialize()	void
m	button_onClicked()	void
m	errors(String)	void
m	checkInput(TextField)	boolean
m	handleKeyReleased()	void
m	button_onClickedCreate()	void
m	button_onClickedShow(ActionEvent)	void

This controller deals with the order page. It will perform operations in the back as the user clicks on some buttons or inserts some data. Here, the data will also be checked. The rest of the methods will deal with buttons designated for inserting, updating or showing all the orders. It will also take all the orders from the corresponding table at the beginning, when the page is created, in the `initialize()` method.

During the `initialize()` method there will be initialized the choice boxes in order to let the user select the client and product desired in order to create a new order. In this way the user will only be able to create orders with the available resources. If

the number of products is smaller than the desired products for an order an error message will be displayed.

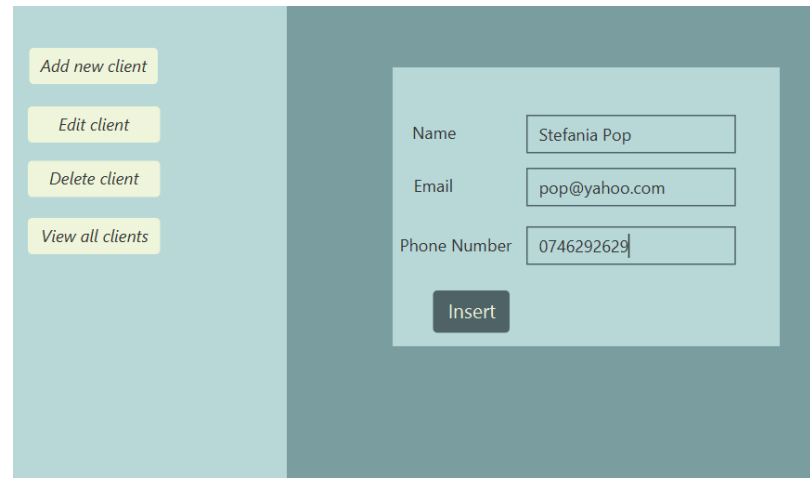
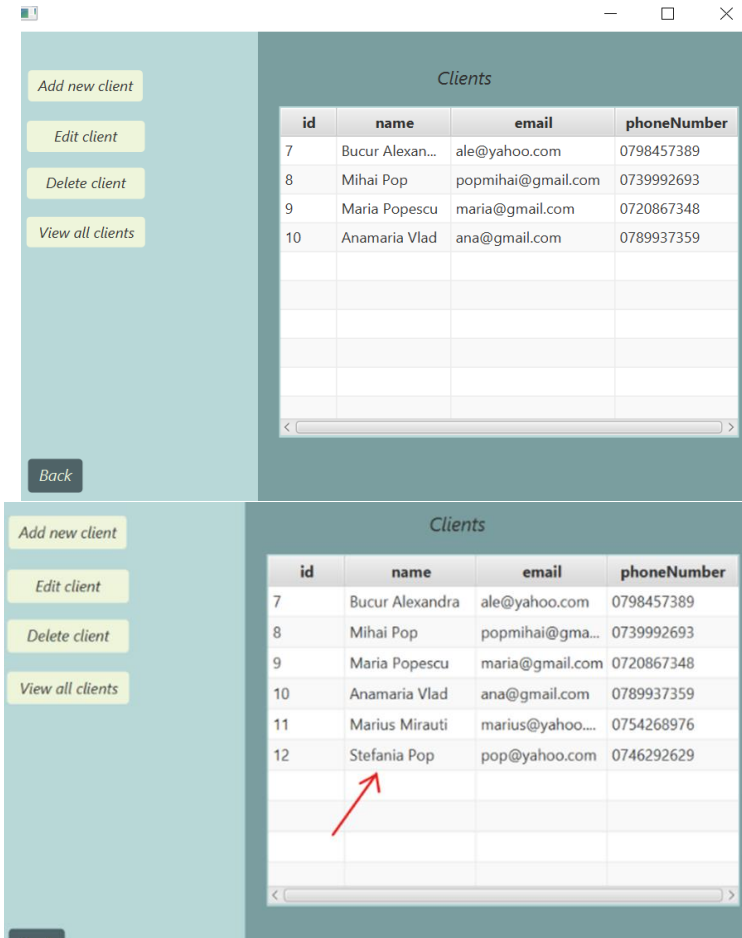
## App Class

App		
m	start(Stage)	void
m	main(String[])	void

This class contains the `main()` of the application. At runtime the application will start from this point.

## 5. Results

The application works as is should, here are some inputs from the application that demonstrate this:



In the first picture is presented the table before any insert. After that, the new client was inserted. There will be presented the updated table, after the client with the name Stefania Pop was inserted.

## 6. Conclusions

This project was a good exercise in remembering the OOP concepts learned in the first semester, but also learning some new concepts regarding design patterns, and the most important one, working with databases. Even though I have not properly worked with databases before, now, due to this project I now understand how they work and how to be used.

I also came to the conclusion that working by yourself and trying to solve the problems(bugs) by yourself is really helpful because the information will remain learned for further applications.

### Further implementation

- Make the interface more appealing
- Develop the application into an online shopping site
- Create more sql specific queries, such as finding a client by its email or phone number
- Develop the application in order to have a login page that will make the users able to access or create their own accounts.

## 7. Bibliography

1. Connect to MySql from a Java application :
  - a. <https://www.baeldung.com/java-jdbc>
  - b. <http://www.mkyong.com/jdbc/how-to-connect-to-mysql-with-jdbc-driver-java/>
2. Layered architectures
  - a. <https://dzone.com/articles/layers-standard-enterprise>
3. Reflection in Java
  - a. <http://tutorials.jenkov.com/java-reflection/index.html>
4. Creating PDF files in Java
  - a. <https://www.baeldung.com/java-pdf-creation>
5. JAVADOC
  - a. <https://www.baeldung.com/javadoc>
6. SQL dump file generation
  - a. <https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html>
7. JavaFX
  - a. [https://docs.oracle.com/javafx/2/ui\\_controls/choice-box.htm](https://docs.oracle.com/javafx/2/ui_controls/choice-box.htm)
  - b. [https://docs.oracle.com/javafx/2/ui\\_controls/table-view.htm](https://docs.oracle.com/javafx/2/ui_controls/table-view.htm)
  - c. [www.stackoverflow.com](http://www.stackoverflow.com)