

Order Management

Student: Bețianu Miruna – Laura
Group: 30422

Table of Contents

1. TASK.....	3
2. PROJECT SPECIFICATIONS	3
3. COMPOSITION	4
1. PACKAGES	4
2. CLASSES	4
THE UML CLASS DIAGRAM:	5
4. IMPLEMENTATION.....	6
• <i>Client (package model).....</i>	<i>6</i>
• <i>Queue (package model)</i>	<i>6</i>
• <i>SimulationManager (package controller)</i>	<i>7</i>
• <i>Controller (package controller)</i>	<i>9</i>
• <i>SimulationFrame (package view)</i>	<i>9</i>
5. USER INTERFACE (HOW TO USE THE APPLICATION)	10
6. MULTITHREADING	ERROR! BOOKMARK NOT DEFINED.
7. FURTHER DEVELOPMENTS	14
8. CONCLUSION	14
9. BIBLIOGRAPHY	14

1. Task

Consider an application Order Management for processing customer orders for a warehouse.

2. Project specifications

This project has as goal the connection between a Java application and a database. Relational databases are used to store the products, the clients and the orders. Furthermore, the application uses (minimally) the following classes:

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

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

The interface should let the user modify the contents of each table of the warehouse database. It is required to have the operations of insert, delete, edit and view all. These operations require a specific implementation using reflection. Using reflection, the code will be more generic and compact.

Also, the application must have an order platform in which the users chooses the client, a product and the quantity of the product. If the quantity required by the client is bigger than the one from stock (stored in the database), the application will print an error message displaying 'Under stock'. Otherwise, if there are as many pieces of product as needed, the stock will be decreased, and the order will be processed. After a successful order, the data regarding the order will be inserted in a specific table and the bill will be stored in a pdf.

Regarding the classes, each layer has a special purpose and calls functions of the layers below it.

- ⇒ Presentation Layer - contains the classes defining the user interface
- ⇒ Business Layer – contains the classes that encapsulate the application logic
- ⇒ Data Access Layer – contains the classes containing the queries and the database connection
- ⇒ Model – contains classes mapped to the database table

3. Composition

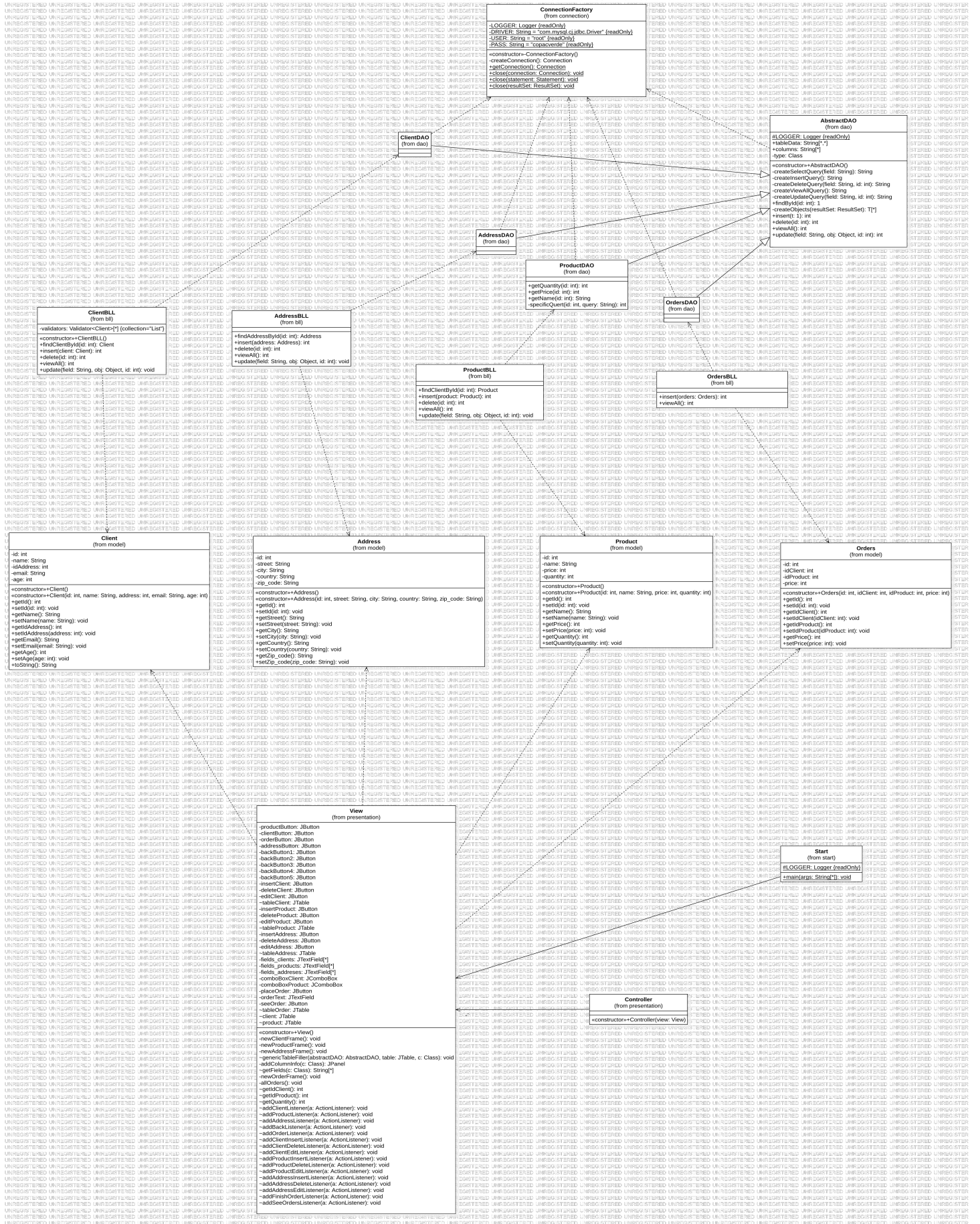
1. Packages

- ❏ bll
- ❏ bll.validators
- ❏ connection
- ❏ dao
- ❏ model
- ❏ presentation
- ❏ start

2. Classes

- ⦿ AddressBLL
- ⦿ ClientBLL
- ⦿ OrdersBLL
- ⦿ ProductBLL
- ⦿ ClientAgeValidator
- ⦿ EmailValidator
- ⦿ Validator
- ⦿ Connection Factory
- ⦿ AbstractDAO
- ⦿ AddressDAO
- ⦿ ClientDAO
- ⦿ OrdersDAO
- ⦿ ProductDAO
- ⦿ Address
- ⦿ Client
- ⦿ Orders
- ⦿ Product
- ⦿ Controller
- ⦿ View
- ⦿ Start

The UML Class Diagram:



4. Implementation

Client (package model)

This class describes the object 'Client'. Each client is identified by some particular attributes: id, name, id of its address, email and age.

The 'id' is the unique feature of the clients (there will be no more than one client with a specific id).

The address, email and age are required when we want to introduce a new client in the database. Each client has to be an adult (over 18) and to have a valid email. The address is auxiliary to the shipping method.

Methods:

- `setName(name)`: sets the name of the client;
- `getName()`: returns the name of the client;
- `setIdAddress(id)`: sets the address id of the client;
- `getIdAddress()`: returns the address id of the client;
- `getId()`: returns the id of the client;
- `setEmail(email)`: sets the email of the client;
- `getEmail()`: returns the email of the client;
- `setAge(age)`: sets the age of the client;
- `getAge()`: returns the age of the client;
- `Client(name, idAddress, email, age)`: creates the object of type Client having the properties received as parameters;

Address (package model)

This class describes the object 'Address'. Each address is identified by some particular attributes: id, street, city, country and zip code.

The 'id' is the unique feature of the addresses (there will be no more than one address with a specific id).

Each address is specific to one or many clients.

Methods:

- `Address (street, city, country, zip code)`: creates the object of type Address. Also, it sets its properties to the ones received as parameters.
- `setStreet(street)`: sets the street of the address;
- `getStreet()`: returns the street of the address;
- `setCity(city)`: sets the city of the address;
- `getCity()`: returns the city of the address;
- `setCountry(country)`: sets the country of the address;
- `getCountry()`: returns the country of the address;
- `setZip_code(zip_code)`: sets the zip code of the address;
- `getZip_code()`: returns the zip code of the address;
- `getId()`: returns the id of the address;

• Product (package model)

This class describes the object 'Product'. Each address is identified by some particular attributes: id, name, price and quantity.

The 'id' is the unique feature of the products (there will be no more than one product with a specific id).

Methods:

- **Product (name, price, quantity)**: creates the object of type Product. Also, it sets its properties to the ones received as parameters.
- **setName(name)**: sets the name of the product;
- **getName()**: returns the name of the product;
- **setPrice(price)**: sets the price of the product;
- **getPrice()**: returns the price of the product;
- **setQuantity(quantity)**: sets the quantity of the product;
- **getQuantity()**: returns the quantity of the product.

• Orders (package model)

This class describes the object 'Orders'. Each address is identified by some particular attributes: id, client id, product id and price.

The 'id' is the unique feature of the orders (there will be no more than one order with a specific id).

Methods:

- **Orders (idProduct, idClient, price)**: creates the object of type Orders. Also, it sets its properties to the ones received as parameters.
- **setIdProduct(idProduct)**: sets the product id of the order;
- **getIdProduct()**: returns the product id of the order;
- **setIdClient(idClient)**: sets the client id of the order;
- **getIdClient()**: returns the client id of the order;
- **setPrice(price)**: sets the price of the order;
- **getPrice()**: returns the price of the order.
- **getId()**: returns the id of the order.

• AbstractDAO (package dao)

This class controls the execution of the queries being the one behind the data access. In this class, the procedure of reflection has a main usage. Being abstract, the class can be extended afterwards to the other classes responsible with data access.

Methods (only generic methods):

- **AbstractDAO()**: constructor of the class; the type specific to the class is initialized.
- **createSelectQuery(field)**: this method generates a string corresponding to the SQL syntax of SELECT query;

- `createInsertQuery()`: this method generates a string corresponding to an INSERT query;
- `createDeleteQuery()`: this method generates a string corresponding to a DELETE query;
- `createViewAllQuery()`: this method generates a string corresponding to SELECT query without any conditions after it;
- `createUpdateQuery()`: this method generates a string corresponding to UPDATE query;
- `findById(id)`: returns the set of objects corresponding to that id, using reflection as technique.
- `createObjects()`: returns the set of objects related to the result set obtain after executing a query.
- `insert(T)`: executes the INSERT query;
- `delete(T)`: executes the DELETE query;
- `viewAll(T)`: executes the view all query, displaying all the data from the table given as parameter;
- `update(field, object, id)`: updates the field given as parameter with the value of the object at the specific id; this is done by executing the UPDATE query.

• ProductDAO (package dao)

This class extends the AbstractDAO class and implements the additional queries specific only to the products. Example of usage: when an order is placed, the bill needs to contain the name of the product and also the price and quantity in order to compute the total price and to see if there are enough products on the stock.

Methods:

- `getQuantity(id)`: returns the quantity of the product with id given as parameter;
- `getName(id)`: returns the name of the product with id given as parameter;
- `getPrice()`: returns the price of the product with id given as parameter;
- `specificQuery(id, query)`: executes a query; helping method for the ones presented above.

• ClientDAO, OrdersDAO, AddressDAO (package dao)

These classes extend the AbstractDAO class uses its methods for executing generic queries. Also, the implementation did not require specific information from these tables, so they contain only the methods from the method they extend.

• **ConnectionFactory (package connection)**

This class does the connection with the SQL database having methods responsible of realizing the actual connection and of closing the connection, the statement and the result set.

• **AddressBLL, ClientBLL, OrdersBLL, ProductBLL (package bll)**

The BLL classes represent the business logic level and encapsulate the application logic. They contain methods that call other methods from the data access layer.

• **ClientAgeValidator (package bll.validators)**

This class implements a validator for the age of the client.

• **ClientEmailValidator (package bll.validators)**

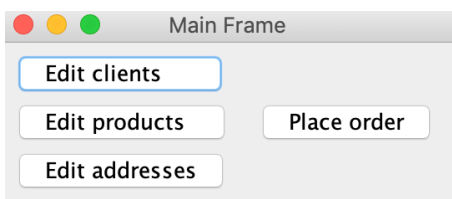
This class implements a validator for the email of the client.

• **Controller (package presentation)**

This class implements all the listeners responsible with how the application should work. It uses 'ActionListener' by implementing in its inner classes.

• **View (package presentation)**

This class creates the graphical user interface.



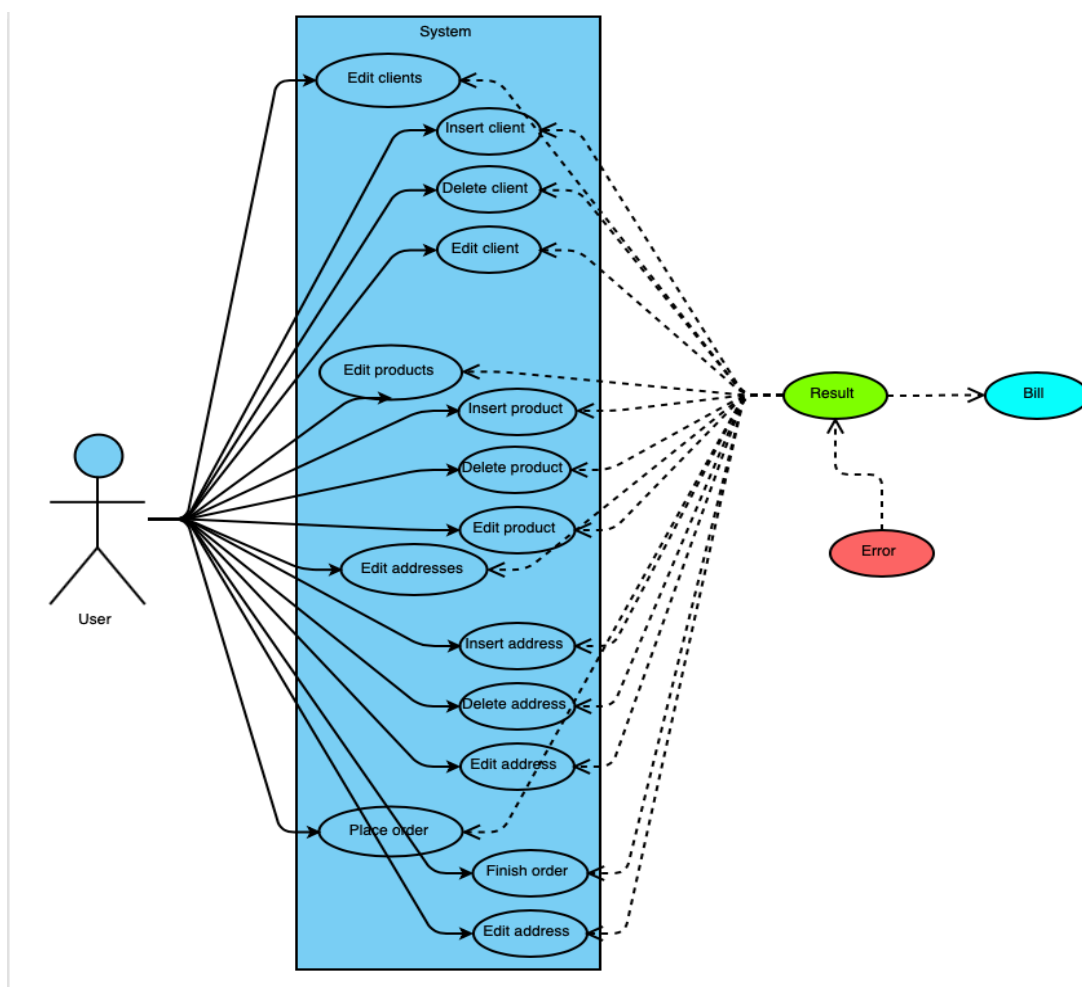
Each text field, label and the button needed in order to create this interface is created and instantiated in this class.

Methods:

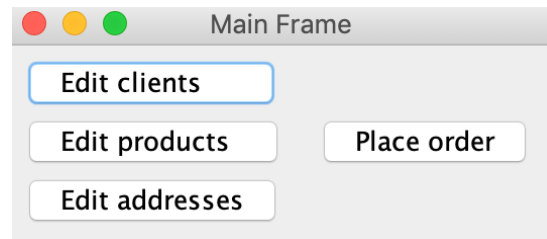
- **View()**: constructor of the class;
- **newGenericFrame()**: method that opens a new frame once a button from the main frame is pressed.
- **newClientFrame()**: opens the client frame once the 'Edit clients' button is pressed.
- **newProductFrame()**: opens the product frame once the 'Edit products' is pressed.
- **newAddressFrame()**: opens the address frame once the 'Edit addressed' is pressed.

- **newOrderFrame()**: opens the order frame once the button 'Place order is pressed'.
- **allOrders()**: frame that opens when the button 'See orders' from order frame is pressed.
- **genericTableFiller(parameters)**: fills a JTable with respect to a specific class.
- **addColumnInfo(Class)**: returns the panel that contains the column info needed when wanting to insert a row in a table;
- **getFields()**: returns the information introduced by the user;
- **getClientId()**: returns the client id from the order platform;
- **getProductId()**: returns the product id from the order platform;
- **getQuantity()**: returns the quantity of a product specified by the user in the interface;
- **other methods responsible with setting the listeners.**

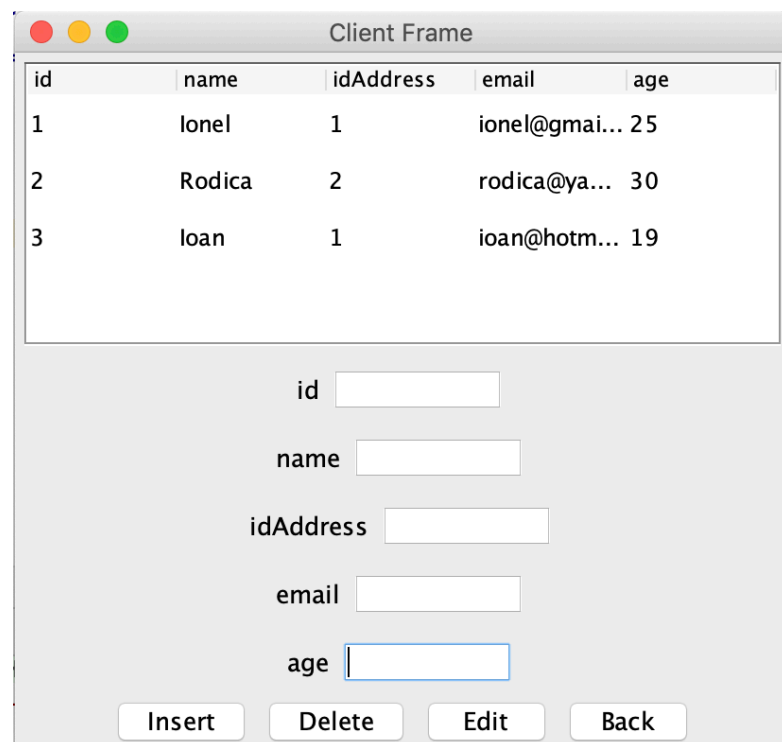
5. User Interface (how to use the application)



Use-Case Diagram



- ⇒ At a first glance, we observe that the graphical interface has buttons;
- ⇒ Once the 'Edit clients' button is pressed, the application will display the following frame:



The 'Client Frame' window displays a table with the following data:

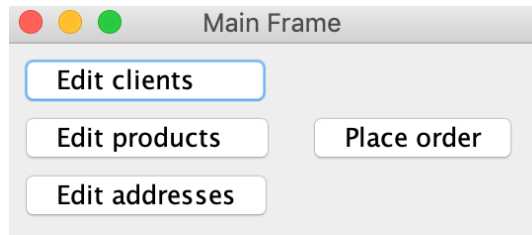
id	name	idAddress	email	age
1	Ionel	1	ionel@gmai...	25
2	Rodica	2	rodica@ya...	30
3	Ioan	1	ioan@hotm...	19

Below the table are input fields for 'id', 'name', 'idAddress', 'email', and 'age'. At the bottom are four buttons: 'Insert', 'Delete', 'Edit', and 'Back'.

- ⇒ This form is responsible with the client's management. The user can modify the table 'Client' from the database by pressing the buttons from the button.
- ⇒ Note that if the user does not introduce enough data or valid data, the application will display an error message. In case of invalid data, pressing the buttons will not change the data from the table.
- ⇒ Example: pressing the button 'Insert' without completing the text fields:



⇒ Pressing the button 'Back' brings the user to the main frame



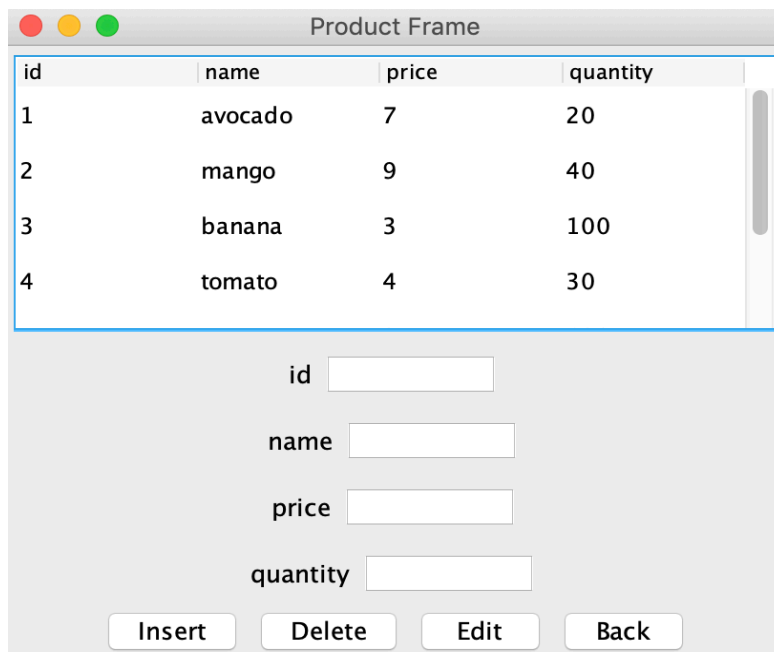
Main Frame

Edit clients

Edit products Place order

Edit addresses

⇒ The product frame and the address one are similar to the one presented above.



Product Frame

id	name	price	quantity
1	avocado	7	20
2	mango	9	40
3	banana	3	100
4	tomato	4	30

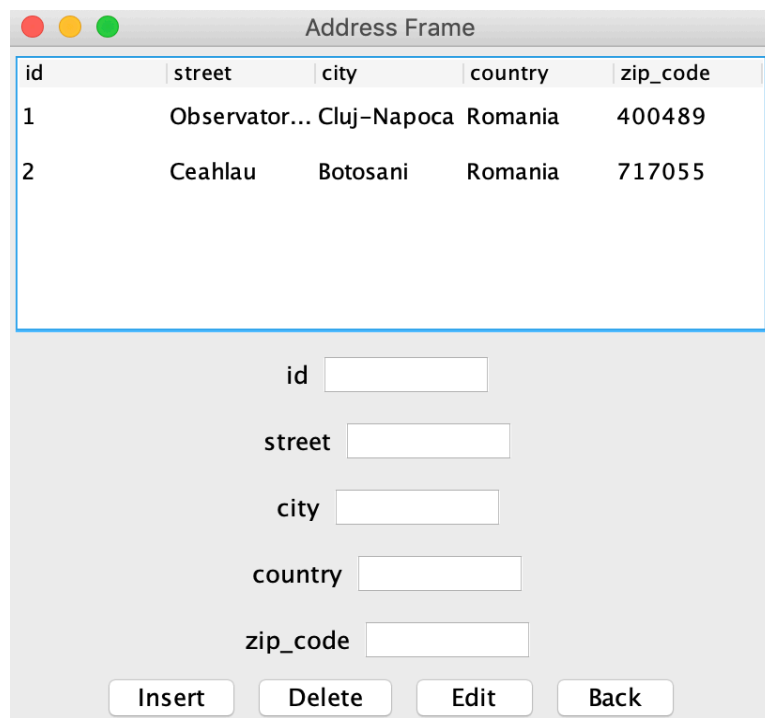
id

name

price

quantity

Insert Delete Edit Back



Address Frame

id	street	city	country	zip_code
1	Observator...	Cluj-Napoca	Romania	400489
2	Ceahlau	Botosani	Romania	717055

id

street

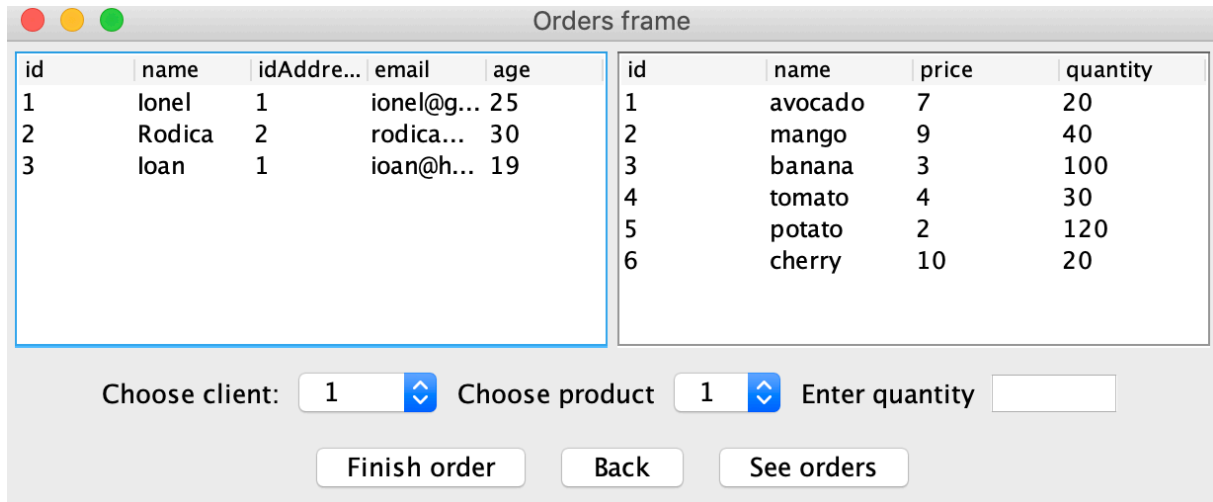
city

country

zip_code

Insert Delete Edit Back

⇒ Order frame:



id	name	idAddress	email	age
1	Ionel	1	ionel@g...	25
2	Rodica	2	rodica...	30
3	Ioan	1	ioan@h...	19

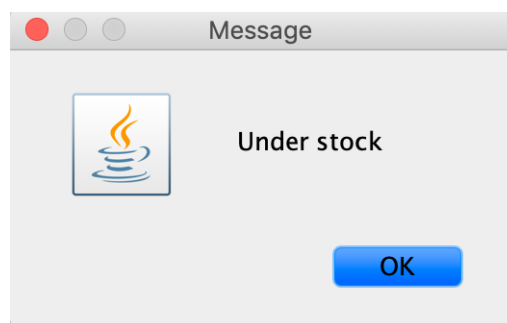
id	name	price	quantity
1	avocado	7	20
2	mango	9	40
3	banana	3	100
4	tomato	4	30
5	potato	2	120
6	cherry	10	20

Choose client: 1 Choose product: 1 Enter quantity:

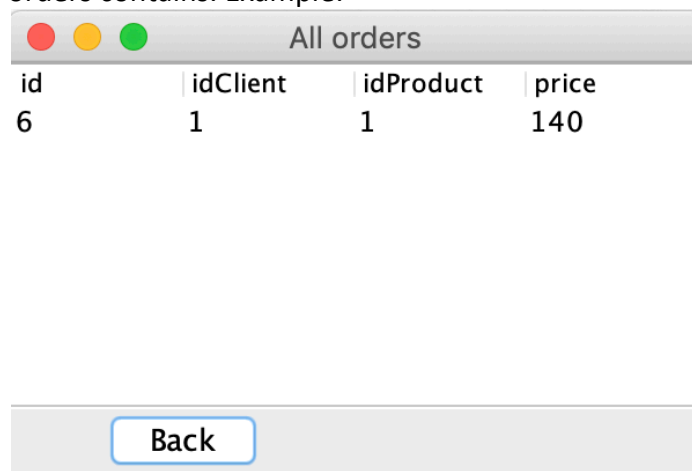
Finish order Back See orders

⇒ The user may compute an order. Using the combo boxes, the user can choose what client and what product are needed. Also, a quantity is required in order to place the command. After hitting the 'Finish order' button, if the desired quantity is in stock and no other invalid data occurs, the order is placed and registered in the orders table.

⇒ Example of error message:



⇒ When the button 'See orders' is pressed, a new frame opens having displayed what the table of orders contains. Example:



id	idClient	idProduct	price
6	1	1	140

Back

6. Further developments

Some of the functionalities of this application can be extended further in order to be more complex and to give the user a real – life store simulation. For instance, we can develop the application so that the user can make a bigger order containing more products. Also, it would be nice to develop shipping module where the address can be finally used.

7. Conclusion

To conclude, creating this application developed my knowledge about java programming language and how to work with the graphical interface (java.swing). Also, it helped me understanding how exceptions work by making me doing research about how to use them and how to mould them in order to be useful for my project. I consider that this project was a little bit challenging because I had never worked with reflection, but after doing research and starting to understand how it works, I realised that this application of order management was not that complicated. I learned about creating generic methods in order to use them for big data.

8. Bibliography

<http://stackoverflow.com> (learned about reflection and helped me to fix some bugs)
<https://www.geeksforgeeks.org> (learned about how to make queries)
<https://www.concretepage.com/itext/create-pdf-with-text-list-table-in-java-using-itext>
(learned about how to create a pdf and to write data into it)