DOCUMENTATION

ORDERS MANAGEMENT SYSTEM FOR A WAREHOUSE

NAME: Mihai Anca

CONTENTS

1. Assignment Objective

Main objective: Create an application **Orders Management** for processing client orders for a warehouse.

Relational databases should be used to store the *products*, the *clients*, and the *orders*.

The application should be designed according to the layered architecture pattern and 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

Create a graphical user interface including:

• A window for *client operations*: add new client, edit client, delete client, view all clients in a table (JTable)

• A window for *product operations*: add new product, edit product, delete product, view all products in a table (JTable)

• A window for creating *product orders*:

- the 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 there are not enough products, an under-stock message will be displayed.

After the order is finalized, the product stock is decremented.

Use 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

- create a generic class that contains the methods for accessing the DB (all tables except *Log*): create object, edit object, delete object and find object

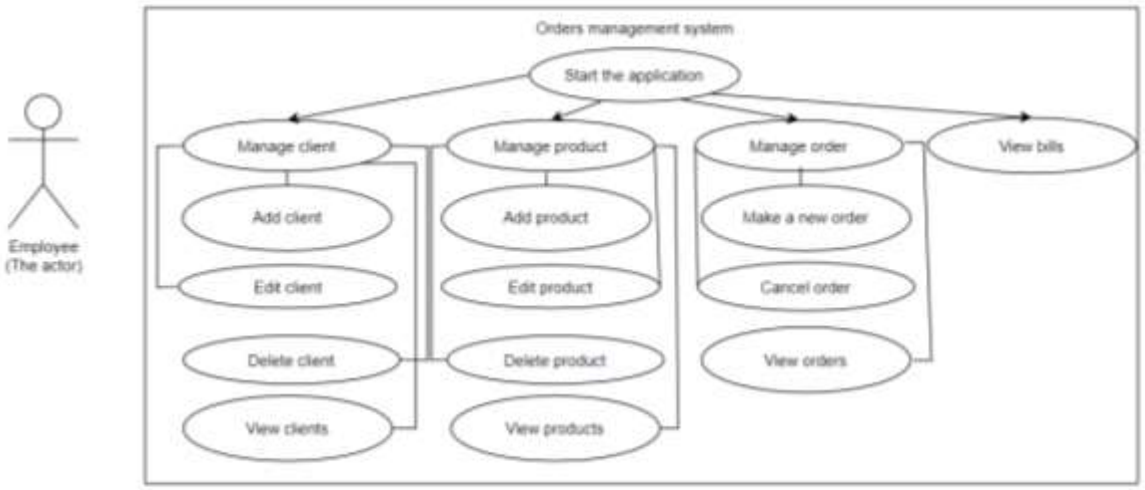
Sub-objectives:

1. understand the problem and its domain
2. analyze the problem and identify the requirements
3. find solutions to individual problems, from which the overall solution is synthesized, sub-objective which can be divided into another two:
3.1 design the orders management system
3.2 implement the orders management system
4. test the application

2. Problem Analysis, Modeling, Scenarios, Use Cases

.

Use case diagram:



Functional requirements:
1. The application should allow an employee to add a new client
2. The application should allow an employee to edit a client
3. The application should allow an employee to delete a client
4. The application should allow an employee to view all currently existing clients
5. The application should allow an employee to add a new product
6. The application should allow an employee to edit a product
7. The application should allow an employee to delete a product
8. The application should allow an employee to view all currently existing products
9. The application should allow an employee to insert a new order
10. The application should allow an employee to cancel an order
11. The application should allow an employee to view all orders that have been made until present moment
12. The application should allow an employee to view all bills that have been generated until present moment

Non-functional requirements:
1. The orders management system should be intuitive and easy to use by the user
2. The orders management system should perform the operations fast
3. The orders management system should be reliable and consistent

Use cases:

1. <u>Name</u>: *add client*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to add a new client
II. The application will display a form in which the client details should be inserted
III. The employee inserts the name of the client, their email and their address
IV. The employee clicks on the "Add client" button
V. The application stores the client data in the database and displays an acknowledge message

<u>Alternative Sequence</u>:
Invalid values for the client's data
- The employee inserts an invalid email
- The application displays an error message and requests the employee to insert a valid email
- The scenario returns to step **III.**

2. <u>Name</u>: *edit client*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to edit a client
II. The application will display a form in which the client's new details should be inserted
III. The employee inserts the name of the client, their email and their address
IV. The employee clicks on the "Edit client" button
V. The application edits the client data in the database and displays an acknowledge message

<u>Alternative Sequence</u>:
Invalid values for the client's data
- The employee inserts an invalid email/id
- The application displays an error message and requests the employee to insert a valid email/id
- The scenario returns to step **III.**

3. <u>Name</u>: *delete client*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to delete a client
II. The application will display a form in which the client to be deleted is selected
III. The employee clicks on the "Delete client" button
V. The application deletes the client from the database and displays an acknowledge message

4. <u>Name</u>: *view clients*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to view clients

II. The application will display in a JTable all the clients existing in the database

5. **<u>Name</u>:** *add product*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to add a new product
II. The application will display a form in which the product details should be inserted
III. The employee inserts the name of the product, its stock and its price
IV. The employee clicks on the "Add product" button
V. The application stores the product data in the database and displays an acknowledge message

<u>Alternative Sequence</u>:
Invalid values for the product's data
- The employee inserts an invalid stock/price
- The application displays an error message and requests the employee to insert a valid stock/price
- The scenario returns to step **III.**

6. **<u>Name</u>:** *edit product*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to edit a product
II. The application will display a form in which the product's new details should be inserted
III. The employee inserts the name of the product, its stock and its price
IV. The employee clicks on the "Edit product" button
V. The application edits the product data in the database and displays an acknowledge message

<u>Alternative Sequence</u>:
Invalid values for the product's data
- The employee inserts an invalid stock/price/id
- The application displays an error message and requests the employee to insert a valid stock/price/id
- The scenario returns to step **III.**

7. **<u>Name</u>:** *delete product*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to delete a product
II. The application will display a form in which the product to be deleted is selected
III. The employee clicks on the "Delete product" button
V. The application deletes the product from the database and displays an acknowledge message

8. <u>Name</u>: *view products*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to view products
II. The application will display in a JTable all the products existing in the database

9. <u>Name</u>: *make a new order*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to make a new order
II. The application will display a form in which the client/product of the order is selected
III. The employee inserts the quantity of the product to be ordered
IV. The employee clicks on the "Confirm order" button
V. The application stores the order data in the database and displays the content of the bill generated by confirming the order

<u>Alternative Sequence</u>:
Invalid values for the order's data
- The employee inserts an invalid quantity (ex: > available stock)
- The application displays an error message (ex: under-stock) and requests the employee to insert a valid quantity
- The scenario returns to step **II.**

10. <u>Name</u>: *cancel order*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to cancel an order
II. The application will display a form in which the order to be deleted's id should be inserted
III. The employee clicks on the "Delete order" button
IV. The application edits the order's Available flag data in the database and displays an acknowledge message

<u>Alternative Sequence</u>:
Invalid values for the order's data
- The employee inserts an invalid id
- The application displays an error message and requests the employee to insert a valid id
- The scenario returns to step **II.**

<u>Alternative Sequence</u>:
Already cancelled ordered
- The employee inserts an id for an already cancelled order
- The application displays an error message and requests the employee to insert a valid id for an order which wasn't cancelled yet
- The scenario returns to step **II.**

11. <u>Name</u>: *view orders*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to view all orders
II. The application will display in a JTable all the orders existing in the database

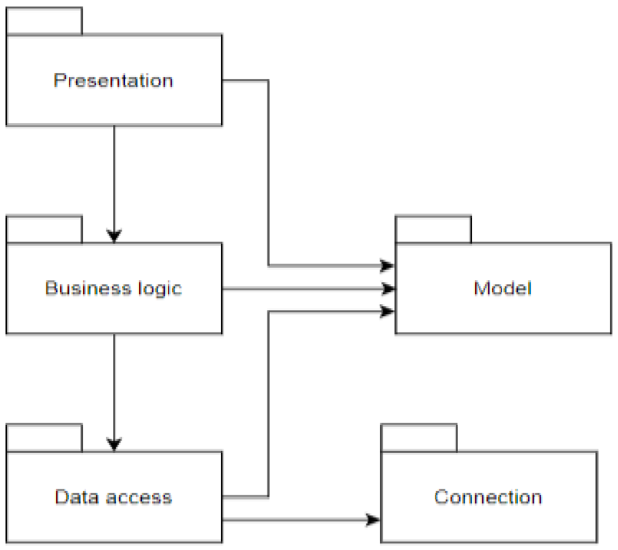
12. <u>Name</u>: *view bills*
<u>Primary Actor</u>: *employee*
<u>Main Success Scenario</u>:
I. The employee selects the option to view all bills
II. The application will display in a JTable all the bills existing in the database

3. Design

OOP design of the application:

The project is based on a layered architecture, which has the following advantages:
1. **Code Maintenance is easy**: we can easily determine any kind of change in the code
2. **Security:** the data-providing package isn't affected by the other packages
3. **Ease of development:** building time taken by the application will be small as all the layers can work together at the same time

UML package diagram:



The project has 5 packages + the *App* class (its purpose is to run the application):

1. *Model* – contains the classes modeling the application data

- in this app, it contains 4 classes: the *Bill*, *Client*, *Order* and *Product* classes

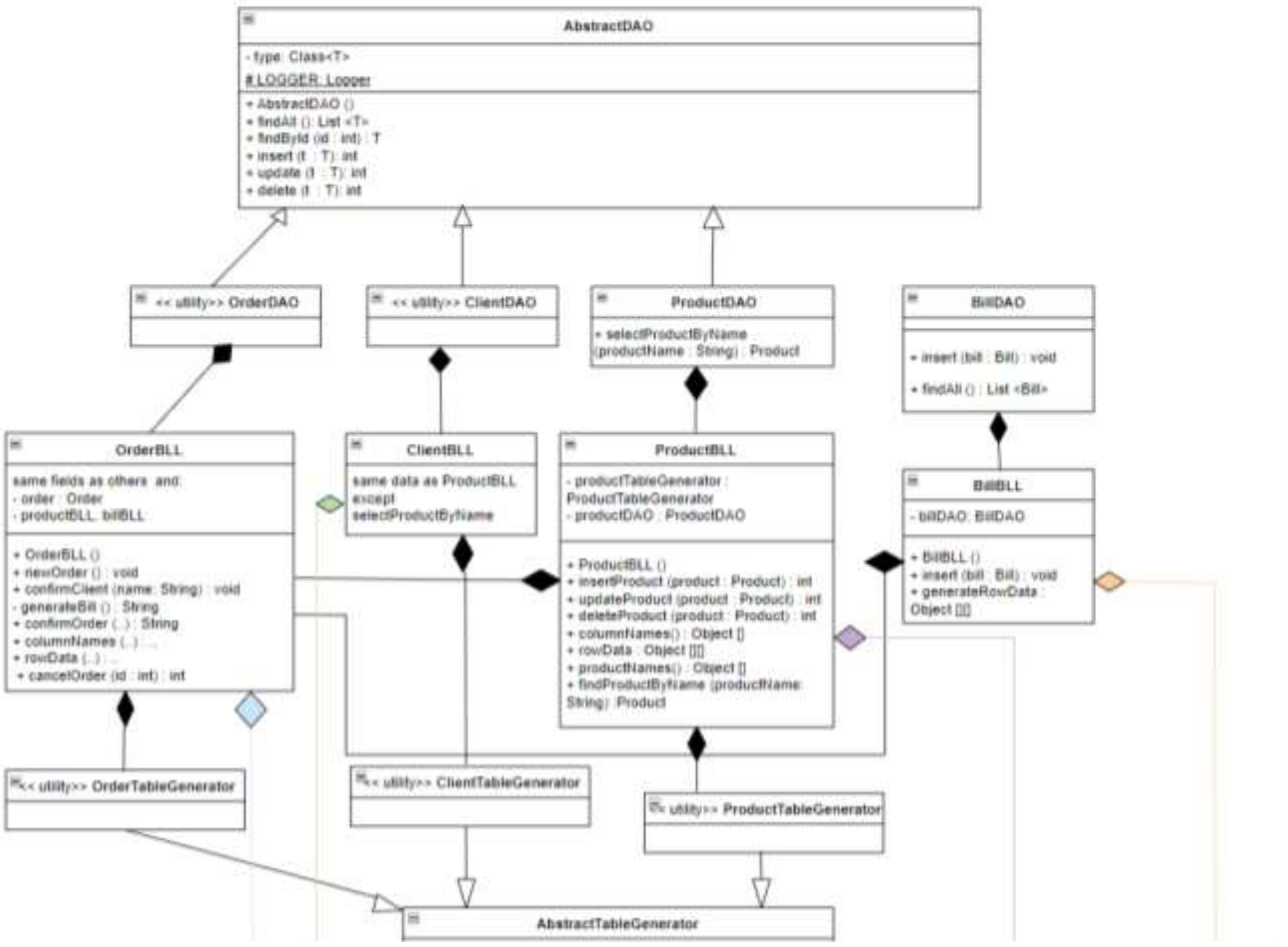
2. *Presentation* – contains the classes implementing the graphical user interface

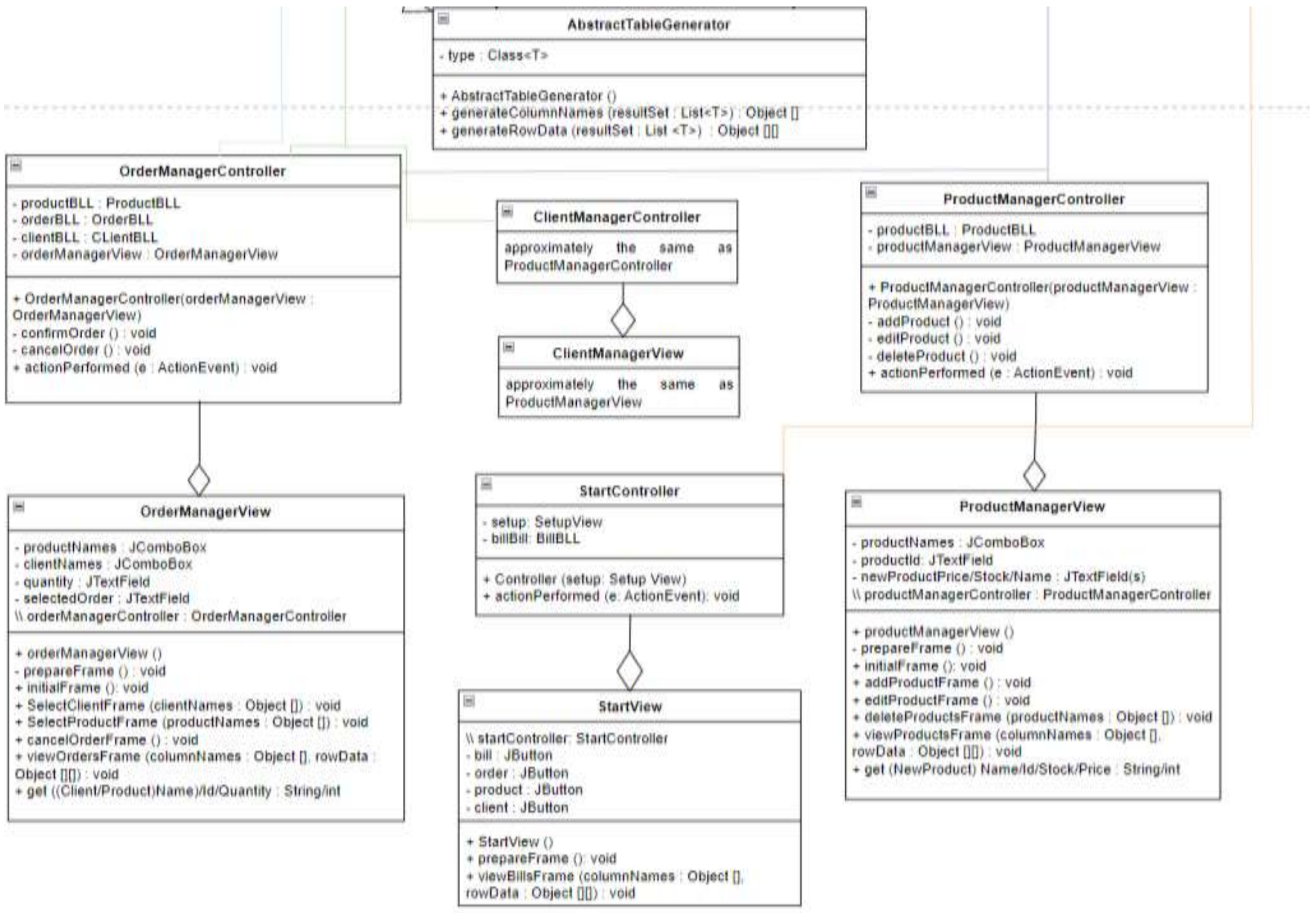
3. *Business Logic* – contains the classes that get the inputs of the graphical user interface and process them accordingly (ex: alter the data of the tables of the database based on what was introduced)

4. *Data Access* - contains the classes containing the queries for the database

5. *Connection* – contains the database connection

Class Diagram:


AbstractDAO
- type: Class<T>
# LOGGER: Logger
+ AbstractDAO ()
+ findAll (): List <T>
+ findById (id : int) : T
+ insert (t : T): int
+ update (t : T): int
+ delete (t : T): int
<< utility>> OrderDAO
<< utility>> ClientDAO
ProductDAO
+ selectProductByName (productName : String) : Product
BillDAO
+ insert (bill : Bill) : void
+ findAll () : List <Bill>
OrderBLL
same fields as others and:
- order : Order
- productBLL, billBLL
+ OrderBLL ()
+ newOrder () : void
+ confirmClient (name: String) : void
- generateBill () : String
+ confirmOrder (...) : String
+ columnNames (...) : ...
+ rowData (...) : ...
+ cancelOrder (id : int) : int
ClientBLL
same data as ProductBLL except selectProductByName
ProductBLL
- productTableGenerator : ProductTableGenerator
- productDAO : ProductDAO
+ ProductBLL ()
+ insertProduct (product : Product) : int
+ updateProduct (product : Product) : int
+ deleteProduct (product : Product) : int
+ columnNames() : Object []
+ rowData : Object [][]
+ productNames() : Object []
+ findProductByName (productName : String) : Product
BillBLL
- billDAO: BillDAO
+ BillBLL ()
+ insert (bill : Bill) : void
+ generateRowData : Object [][]
<< utility>> OrderTableGenerator
<< utility>> ClientTableGenerator
<< utility>> ProductTableGenerator
AbstractTableGenerator


AbstractTableGenerator
- type : Class<T>
+ AbstractTableGenerator ()
+ generateColumnNames (resultSet : List<T>) : Object []
+ generateRowData (resultSet : List <T>) : Object [][]
OrderManagerController
- productBLL : ProductBLL
- orderBLL : OrderBLL
- clientBLL : CLientBLL
- orderManagerView : OrderManagerView
+ OrderManagerController(orderManagerView : OrderManagerView)
- confirmOrder () : void
- cancelOrder () : void
+ actionPerformed (e : ActionEvent) : void
ClientManagerController
approximately the same as ProductManagerController
ClientManagerView
approximately the same as ProductManagerView
ProductManagerController
- productBLL : ProductBLL
- productManagerView : ProductManagerView
+ ProductManagerController(productManagerView : ProductManagerView)
- addProduct () : void
- editProduct () : void
- deleteProduct () : void
+ actionPerformed (e : ActionEvent) : void
OrderManagerView
- productNames : JComboBox
- clientNames : JComboBox
- quantity : JTextField
- selectedOrder : JTextField
\\ orderManagerController : OrderManagerController
+ orderManagerView ()
- prepareFrame () : void
+ initialFrame (): void
+ SelectClientFrame (clientNames : Object []) : void
+ SelectProductFrame (productNames : Object []) : void
+ cancelOrderFrame () : void
+ viewOrdersFrame (columnNames : Object [], rowData : Object [][]) : void
+ get ((Client/Product)Name)/Id/Quantity : String/int
StartController
- setup: SetupView
- billBill: BillBLL
+ Controller (setup: Setup View)
+ actionPerformed (e: ActionEvent): void
StartView
\\ startController: StartController
- bill : JButton
- order : JButton
- product : JButton
- client : JButton
+ StartView ()
+ prepareFrame (): void
+ viewBillsFrame (columnNames : Object [], rowData : Object [][]) : void
ProductManagerView
- productNames : JComboBox
- productId: JTextField
- newProductPrice/Stock/Name : JTextField(s)
\\ productManagerController : ProductManagerController
+ productManagerView ()
- prepareFrame () : void
+ initialFrame (): void
+ addProductFrame () : void
+ editProductFrame () : void
+ deleteProductsFrame (productNames : Object []) : void
+ viewProductsFrame (columnNames : Object [], rowData : Object [][]) : void
+ get (NewProduct) Name/Id/Stock/Price : String/int

Used data structures:

- *primitive data types* (ex: int)
- *List* (ArrayList): I used *Java Collections Framework* because it reduces programming effort and increases program speed, quality respectively; I used the *List* interface to store the elements of a table of the database
- *immutable objects* (Bill class)
- *arrays and matrices* (Object[], Object[][]): used to store elements that populate the JTables used
- *newly-created objects* (ex: Client, Product)

4. Implementation

Implementation of classes:

I. model Package**:**

Bill Class

- <u>important methods</u>:
  - ➢ *public String toString()*: puts into a string all the content of the bill generated when an order is finished

Client Class

- <u>important fields</u>:
  - ➢ *private* int ID : the ID of the client
  - ➢ *private* String name : the name of the client
  - ➢ *private* String email : the email of the client
  - ➢ *private* String address: the address of the client

Order Class

- <u>important fields</u>:
  - ➢ *private* int ID : the ID of the order
  - ➢ *private* String clientName : the name of the client
  - ➢ *private* String productName : the name of the product
  - ➢ *private* int quantity: the quantity ordered for the selected product
  - ➢ *private* Date date : the date of the order
  - ➢ *private* int totalPrice : total cost of the order
  - ➢ *private* String available : whether the order has been cancelled or not

Product Class

- <u>important fields</u>:
  - ➢ *private* int ID : the ID of the product
  - ➢ *private* String name : the name of the product
  - ➢ *private* int stock : the current stock of the product
  - ➢ *private* int price : the price of the product

II. business_logic Package:

- *bll* Package

BillBLL Class

- <u>important fields</u>:
  - ➢ *private* BillDAO billDAO : access to the Bill (Log) Table of the database

- <u>important methods</u>:
  - ➢ *public void insert(Bill bill)* : this method inserts a newly-generated bill into the Log Table
  - ➢ *public Object[][] generateRowData()*: generates the data for the JTable displayed when "View bills" button is pressed

ClientBLL Class

- <u>important fields</u>:

➢ *private* ClientDAO clientDAO : access to the Client Table of the database

➢ *private* ClientTableGenerator clientTableGenerator : used to generate the column names and the data to populate the JTable displayed when "View clients" button is pressed

- <u>important methods</u>:

➢ *public int insertClient(Client client)* : this method checks if the newly-inserted client has an appropriate email and inserts it in the Client table of the database

➢ *public int updateClient(Client client)* : this method checks if the client whose data was updated has an appropriate email and updates it in the Client table of the database

➢ *public int deleteClient(Client client)* : this method deletes a client from the Client table of the database

➢ *public Object[] columnNames ()* : this method generates the header of the JTable generated when "View clients" button is pressed

➢ *public Object[][] rowData()* : this method generates the data to populate the JTable generated when "View clients" button is pressed

➢ *public Object[] clientNames ()* : this method returns the names of the currently existing clients in the Client table of the database

OrderBLL Class

- <u>important fields</u>:

➢ *private* OrderDAO orderDAO : access to the Order Table of the database

➢ *private* OrderTableGenerator orderTableGenerator : used to generate the column names and the data to populate the JTable displayed when "View orders" button is pressed

- <u>important methods</u>:

➢ *public void confirmClient(String name)* : this method sets the client of the order

➢ *private String generateBill()* : this method generates the bill of the newly-confirmed order

➢ *public String confirmOrder(String productName, int quantity, ProductBLL productBLL)* throws *Exception*: this method sets the product to be ordered and checks if the quantity required by the client is available; if not, the client will be informed of the under-stock

➢ *public Object[] columnNames ()* : this method generates the header of the JTable generated when "View orders" button is pressed

➢ *public Object[][] rowData()* : this method generates the data to populate the JTable generated when "View orders" button is pressed

➢ *public int cancelOrder(int id)*: this method cancels an available order by setting its available attribute from "Yes" to "No" and by putting back in the stock the products which were ordered

ProductBLL Class

- <u>important fields</u>:

➢ *private* ProductDAO productDAO : access to the Product Table of the database

- *private* ProductTableGenerator productTableGenerator : used to generate the column names and the data to populate the JTable displayed when "View products" button is pressed

- important methods:
- *public int insertProduct(Product product)* : this method checks if the newly-inserted product has an appropriate stock and price and inserts it in the Client table of the database
- *public int updateProduct(Product product)* : this method checks if the product whose data was updated has an appropriate stock and price, and updates it in the Product table of the database
- *public int deleteProduct(Product product)* : this method deletes a product from the Product table of the database
- *public Object[] columnNames ()* : this method generates the header of the JTable generated when "View products" button is pressed
- *public Object[][] rowData()* : this method generates the data to populate the JTable generated when "View products" button is pressed
- *public Object[] productNames ()* : this method returns the names of the currently existing products in the Product table of the database
- *public Product findProductByName(String productName)* : this method finds a product based on the name selected

- *table_generators* Package

AbstractTableGenerator Class

- important methods:
- *public Object[] generateColumnNames(List<T> resultSet)*: this method generates the header of the JTable
- *public Object[][] generateRowData(List<T> resultSet)* : this method the data which will populate the JTable

- class extended by the *ClientTableGenerator, OrderTableGenerator, ProductTableGenerator classes*

– *validators* Package

EmailValidator Class - checks if the string introduced as the email for a client respects the standard format of an email
PriceValidator Class - checks if the integer introduced as the price for a product is greater than 0
StockValidator Class - checks if the integer introduced as the stock for a product is equal to or greater than 0

III. connection Package

ConnectionFactory Class – class responsible for ensuring the database connection

IV. data_access Package

AbstractDAO Class

- <u>important methods</u>:
  - *public List<T> findAll()* : this method returns all the data from a table of the database
  - *public T findById(int id)* : this method searches in a table of the database the element with the given id
  - *public int insert(T t)* : this method inserts in a table of the database an element
  - *public int update(T t)* : this method updates an element in a table of the database
  - *public int delete(T t)* : this method deletes an element from a table of the database

- class extended by the *ClientDAO, OrderDAO, ProductDAO classes*

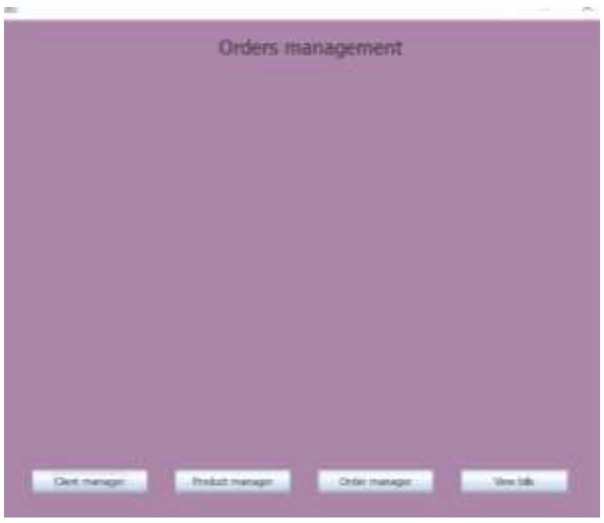
ProductDAO Class

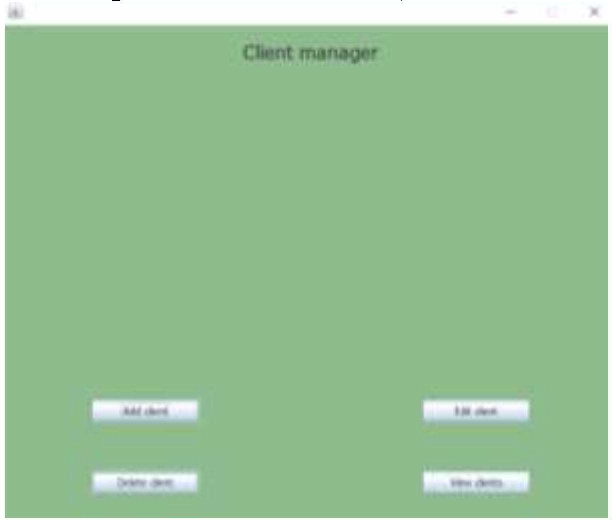
- <u>other important method</u>: *public Product selectProductByName(String productName)* : searches in the Product table of the database a product with a given name

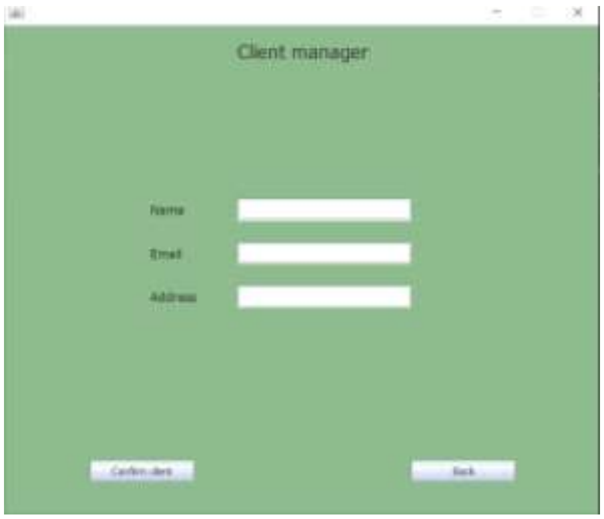
BillDAO Class

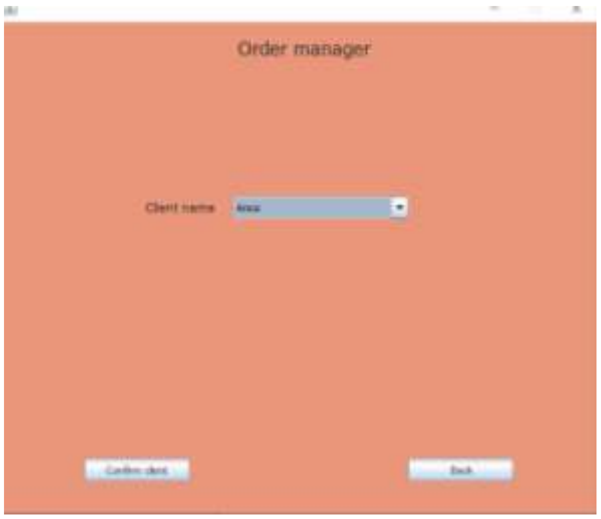
- <u>important methods</u>:
  - *public void insert(Bill bill)* : this method inserts a newly-created bill in the Bill (Log) table of the database
  - *public List<Bill> findAll()* : this method returns all the bills created from the Bill table of the database

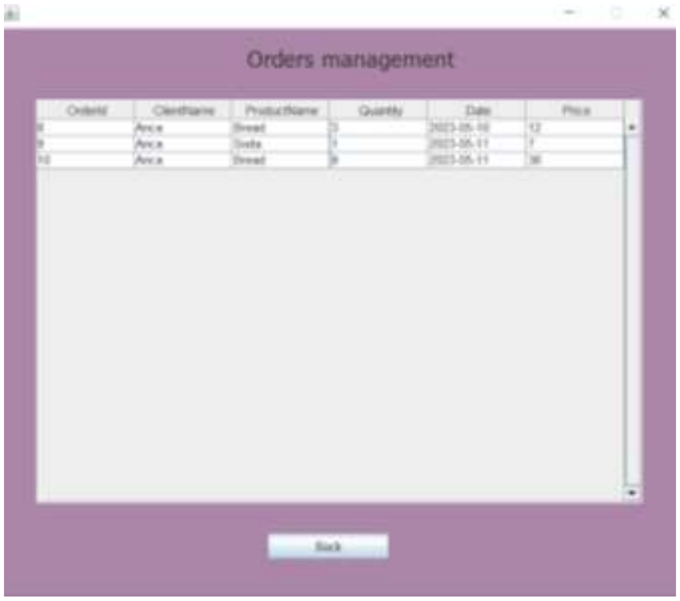
V. presentation Package (Implementation of the Graphical User Interface)











OrderId	ClientName	ProductName	Quantity	Date	Price
8	Anca	Bread	3	2023-05-10	12
9	Anca	[illegible]	1	2023-05-11	7
10	Anca	Bread	9	2023-05-11	36

– *views* Package

StartView class – this class presents the start window of the application

- extends JFrame because, in this way, I can freely choose its size, visibility and layout

- I chose the AbsoluteLayout because I can put my elements (labels, TextFields, buttons) however I want, by specifying the bounds of each element

- <u>important fields:</u>
  - ➢ *private* JButton client: button which, when clicked, opens the Client manager
  - ➢ *private* JButton product: button which, when clicked, opens the Product manager

- *private* JButton order: button which, when clicked, opens the Order manager
- *private* JButton bill: button which, when clicked, displays the bills generated until the moment the button was pressed
- StartController startController

- <u>important methods:</u>
- *public void prepareFrame ()* : sets the initial view of the window
- *public void viewBillsFrame(Object[] columnNames, Object[][] rowData)* : displays in a JTable all the generated bills, found in the Bill table of the database
- getters: used by the controller to take the information introduced by the user in the gui

ClientManagerView class – this class presents the Client Manager window of the application

- extends JFrame because, in this way, I can freely choose its size, visibility and layout

- I chose the AbsoluteLayout because I can put my elements (labels, TextFields, buttons) however I want, by specifying the bounds of each element

- <u>important fields:</u>
- ClientManagerController clientManagerController
- private JComboBox clientNames – displays the names of the current existing clients in the Client table of the database

- <u>important methods:</u>
- *public void initialFrame ()* : sets the initial view of the window
- *public void addClientFrame ()* : sets the view of the window when "Add client" button is pressed
- *public void editClientFrame ()* : sets the view of the window when "Edit client" button is pressed
- *public void deleteClientFrame ()* : sets the view of the window when "Delete client" button is pressed
- *public void viewClientsFrame(Object[] columnNames, Object[][] rowData)* : displays in a JTable all the existing clients, found in the Client table of the database
- getters: used by the controller to take the information introduced by the user in the gui

ProductManagerView Class has a similar structure and functionality to the *ClientManagerView* Class

OrderManagerView Class also has a similar structure and functionality to the *ClientManagerView* Class but:

- has no updateOrder Frame
- has in addition:

  - <u>field</u> : private JComboBox productNames – displays the names of the current existing products in the Product table of the database

  - <u>methods</u> :

    ➢ *public void SelectClientFrame(Object[] clientNames)* : sets the view of the window when "New order" button is pressed, where a client must be selected for the order

    ➢ *public void SelectProductFrame(Object[] productNames)* : sets the view of the window when "Confirm client" button is pressed, where a product and a quantity for it must be selected for the order

- *controllers* Package

StartController class

- <u>important methods</u>

  ➢ *public void actionPerformed (ActionEvent e)* (for detecting whenever a button in the *StartView* window is pressed)

ClientManagerController class

- <u>important methods</u>

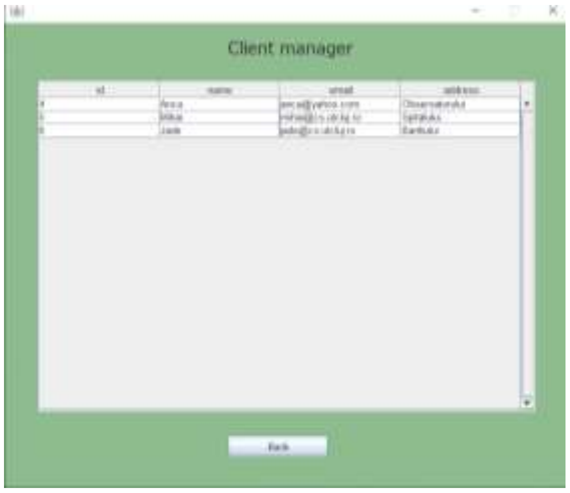
  ➢ *public void actionPerformed (ActionEvent e)* (for detecting whenever a button in the *ClientManagerView* window is pressed)

ProductManagerController and *OrderManagerController* Classes have a similar structure and functionality to the *ClientManagerController* Class

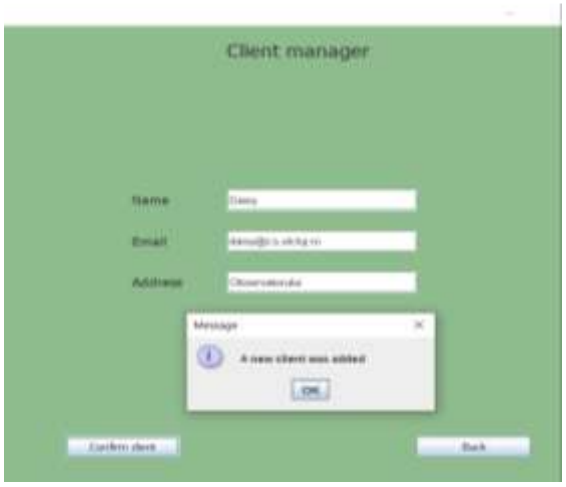
5. Results

Displayed by the Graphical User Interface

1. Client Manager/ Product Manager Window (inserting a new client)



id	name	email	address
[illegible]	Anca	anca@yahoo.com	Observatorului
[illegible]	Mihai	mihai@cs.utcluj.ro	Spitalului
[illegible]	Jade	jade@cs.utcluj.ro	Baritiului



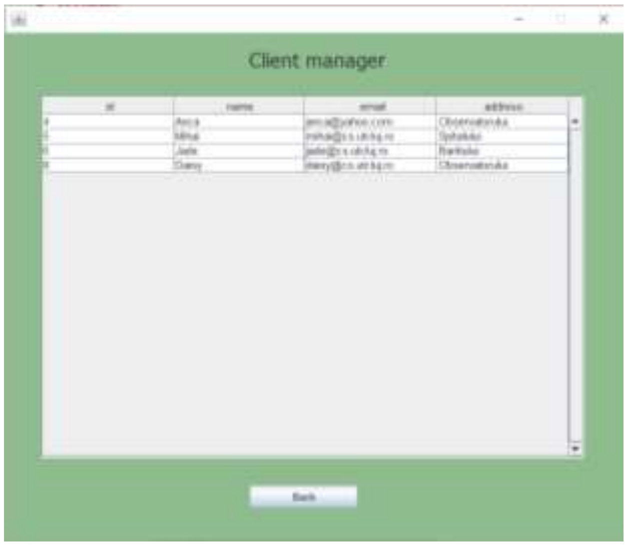
Name: Daisy

Email: daisy@cs.utcluj.ro

Address: Observatorului

Message

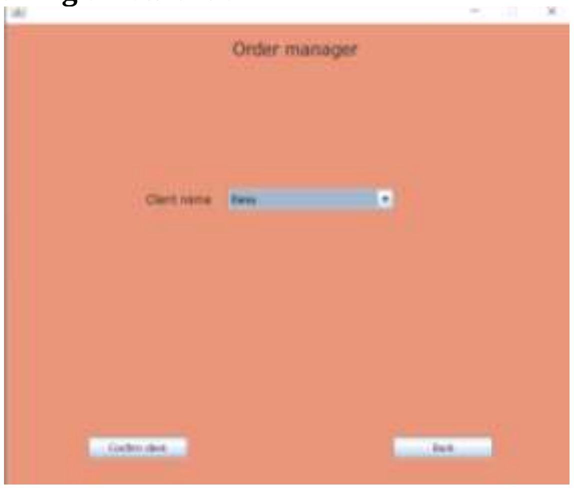
A new client was added



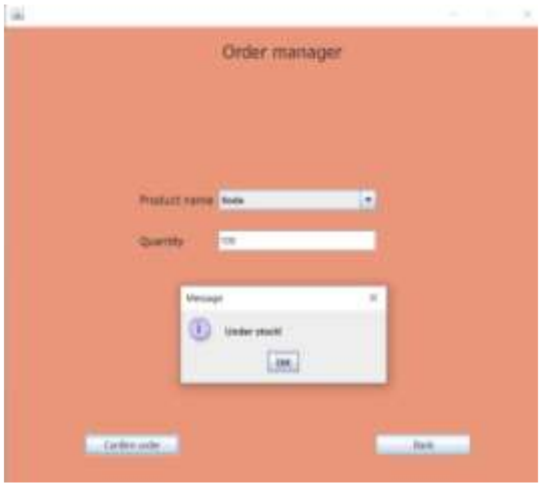
id	name	email	address
[illegible]	Anca	anca@yahoo.com	Observatorului
[illegible]	Mihai	mihai@cs.utcluj.ro	Spitalului
[illegible]	Jade	jade@cs.utcluj.ro	Baritiului
[illegible]	Daisy	daisy@cs.utcluj.ro	Observatorului

2. Order Manager Window

I. making a new order



Client name: Daisy

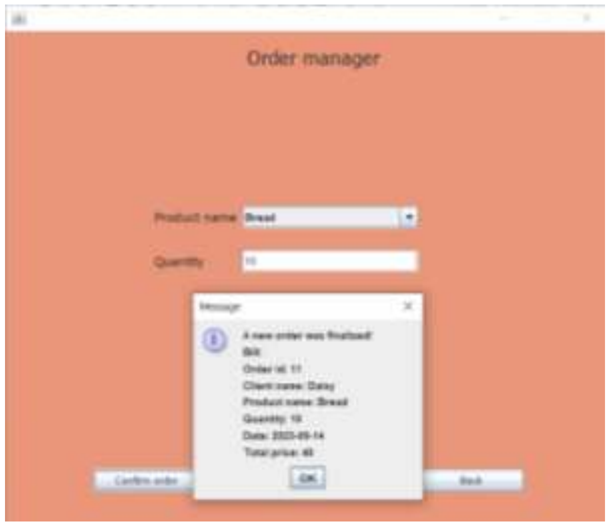


Product name: Soda

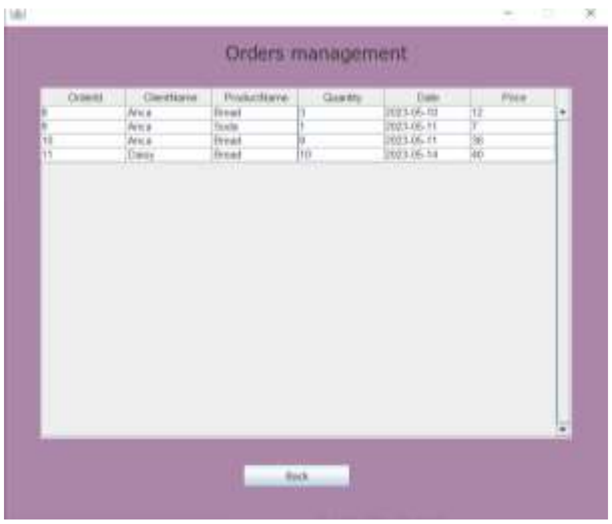
Quantity: 100

Message

Under stock!

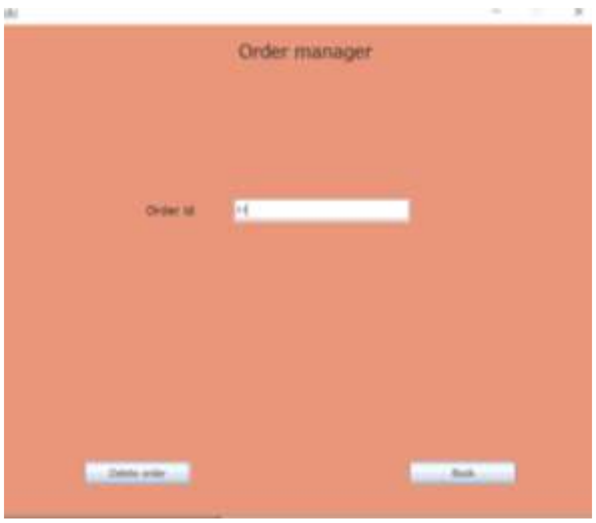


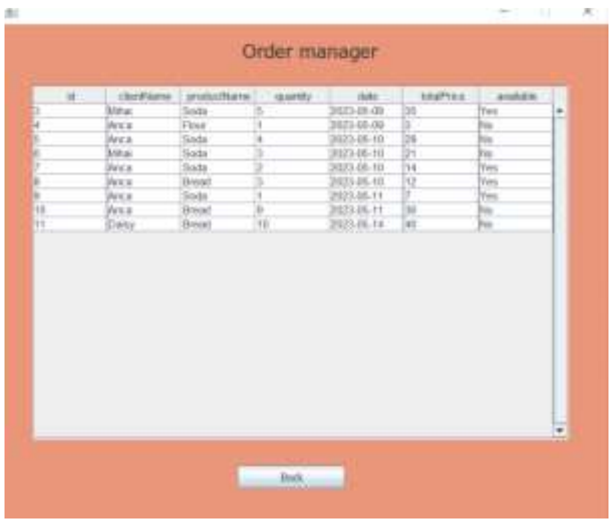
A new order was finalized!
Bill:
Order id: 11
Client name: Daisy
Product name: Bread
Quantity: 10
Date: 2023-05-14
Total price: 40



OrderId	ClientName	ProductName	Quantity	Date	Price
8	Anca	Bread	1	2023-05-10	12
9	Anca	Soda	1	2023-05-11	7
10	Anca	Bread	9	2023-05-11	36
11	Daisy	Bread	10	2023-05-14	40

II. cancelling an order





id	clientName	productName	quantity	date	totalPrice	available
3	Mihai	Soda	5	2023-05-09	35	Yes
4	Anca	Flour	1	2023-05-09	3	No
5	Anca	Soda	4	2023-05-10	28	No
6	Mihai	Soda	3	2023-05-10	21	No
7	Anca	Soda	2	2023-05-10	14	Yes
8	Anca	Bread	3	2023-05-10	12	Yes
9	Anca	Soda	1	2023-05-11	7	Yes
10	Anca	Bread	9	2023-05-11	36	No
11	Daisy	Bread	10	2023-05-14	40	No

6. Conclusions

Conclusions

Firstly, with the help of this assignment, I have remembered the OOP notions learned last semester and also improved them. In addition, I have "relearned" how to model an application connected to a database.

Secondly, I had to manage my time properly in order to be able to complete until deadline this assignment which represents quite a complex project.

Lastly, I faced and learned from multiple scenarios when the code I wrote wasn't doing exactly what I was expecting it to do and, therefore, I had to debug it, tool which can be seen as crucial when programming.

What I have learned from this assignment: working with reflection in order to generalize:
- the main SQL queries specific to a database (find, insert, update, delete, view all)
- the construction of the corresponding JTable of a table in a database

Future developments:
- an order can contain multiple types of products
- a client has a limit budget and, when making an order, checks if the total price is less or equal to the client's budget and then subtracts the value of it from their budget

7. Bibliography

1. Lectures of Fundamental Programming Techniques - S.L. dr. ing. Cristina POP
2. https://www.geeksforgeeks.org/
3. https://stackoverflow.com/questions
4. https://www.baeldung.com/javadoc
5. https://gitlab.com/utcn_dsrl/pt-layered-architecture
6. https://gitlab.com/utcn_dsrl/pt-reflection-example
7. https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html
8. https://docs.oracle.com/javase/tutorial/uiswing/components/table.html