DOCUMENTATATION

ASSIGNMENT 3: ORDERS MANAGEMENT

STUDENT NAME: PELLE ANDREI
GRUPA: 30424

TABLE OF CONTENTS

1. Objectives

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

Breaking down the problem, I considered the following tasks:

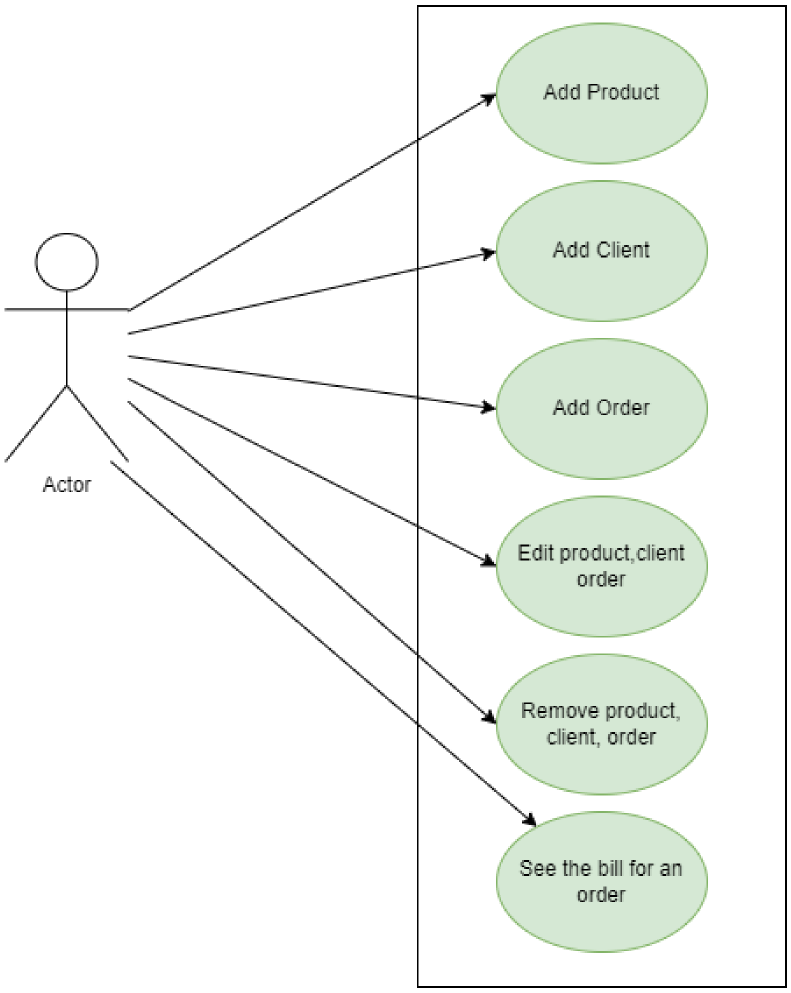
- **Analyze and design an elegant solution** : One has to decide before starting work who is going to be using this application, what the use cases are and how all the components work together at a high level. (Chapter 2 and 3)
- **Building a user-friendly graphical user interface** that allows the user to select the product window, order window, client window and the required operation(Chapter 4)
- **Data modelling**: A database management system with orders is a rather complex data structure; it must be modelled in a way that allows the operations to be performed easily and logically. (Chapter 3)
- **Input validation**: we must make sure the input is correct and display a message to the user if the input is somehow invalid. (Chapter 4)
- **Generics and reflection**: The application must be built using the principles of reflection and generics. These are not simple technologies, and their use requires know-how. (Chapter 4)
- **Test the application**: To ensure that the results are the ones we expect, the application needs to be tested. The results of a create order operation has been appended to thus documentation. (Chapter 5)

2. Problem analysis, modelling, use cases

The following **functional requirements** can be defined:

- The orders management system should allow an employee to add a new client
- The orders management system should allow an employee to add a new client
- The orders management system should allow an employee to add an order
- The orders management system should allow an employee to edit products, clients and orders already present in the system
- The orders management system should allow an employee to remove a product, client or order already present in the system
- The orders management system should allow an employee to see the generated bill for a client's order.

The following **non-functional requirements** can be defined:


Actor
Add Product
Add Client
Add Order
Edit product,client order
Remove product, client, order
See the bill for an order

- The orders management system should be intuitive and easy to use by the user
- The orders management system should be easy to maintain
- The orders management system should be stable
- The orders management system should be modifiable and allow for future development.

Use cases

Use Case 1: add product

Primary Actor: employee
Main Success Scenario:
1. The employee selects the products table
2. The employee selects the option to add a new product
3. The application will display a form in which the product details should be inserted
4. The employee inserts the name of the product, its price and current stock
5. The employee clicks on the "Accept" button
6. The application stores the product data in the database

Alternative Sequence: Invalid values for the product's data
- The user inserts a negative value for the stock of the product
- The application displays an error message and requests the user to insert a valid stock
- The scenario returns to step 4

Use case 2: add order

Primary actor: employee
Main Success Scenario:
1. The employee selects the orders table
2. The employee selects the option to add a new order
3. The application will display a form in which the order details should be inserted
4. The employee inserts the client id, product id and requested quantity
5. The employee clicks on the "Accept" button
6. The application stores the order data in the database and the product stock is correspondingly decremented

Alternative Sequence: Invalid values for the orders data
- The employee inserts a negative value for the stock of the product
- The application displays an error message and requests the employee to insert a valid stock
- The scenario returns to step 4

Alternative Sequence: Not enough stock
- The employee requests too much stock in the quantity field
- The application displays an error message and requests the employee to insert a stock amount that is smaller than the available product
- The scenario returns to step 4

Use case 3: edit order

Primary actor: employee
Main Success Scenario:
1. The employee selects the orders table
2. The employee selects the option to edit an order
3. The application will display a form in which the order details should be edited
4. The employee inserts the client id, product id and requested quantity
5. The employee clicks on the "Accept" button
6. The application updates the order data in the database and the product stock is correspondingly decremented or incremented depending on what the employee requested

Alternative Sequence: Invalid values for the orders data
- The employee inserts a negative value for the stock of the product
- The application displays an error message and requests the employee to insert a valid stock
- The scenario returns to step 4

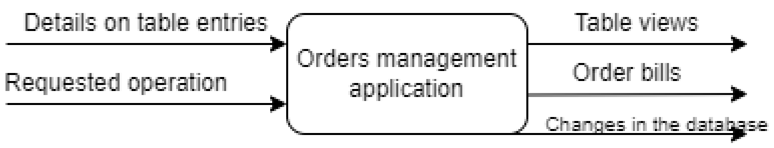
Alternative Sequence: Not enough stock on update
- The employee requests too much stock in the quantity field for the update
- The application displays an error message and requests the employee to insert a stock amount that is smaller than the available product
- The scenario returns to step 4
(Use cases for product and client are similar but no other searching is done)

Use case 4: delete product

Primary actor: employee
Main Success Scenario:
1. The employee selects the product table
2. The employee selects the option to delete a product
3. The application will display a prompt if the employee is sure they want to delete
4. The employee clicks on the “OK” button
5. The application updates the database and removes the product (in the case of an order, it will also increase the stock of the corresponding amount by the order size)

Alternative Sequence: Product still has orders attached to it
- The employee requests to delete a product that has orders attached to it
- The application displays an error message and tells the employee that it is not possible to delete the product in this way
- The scenario returns to step 2

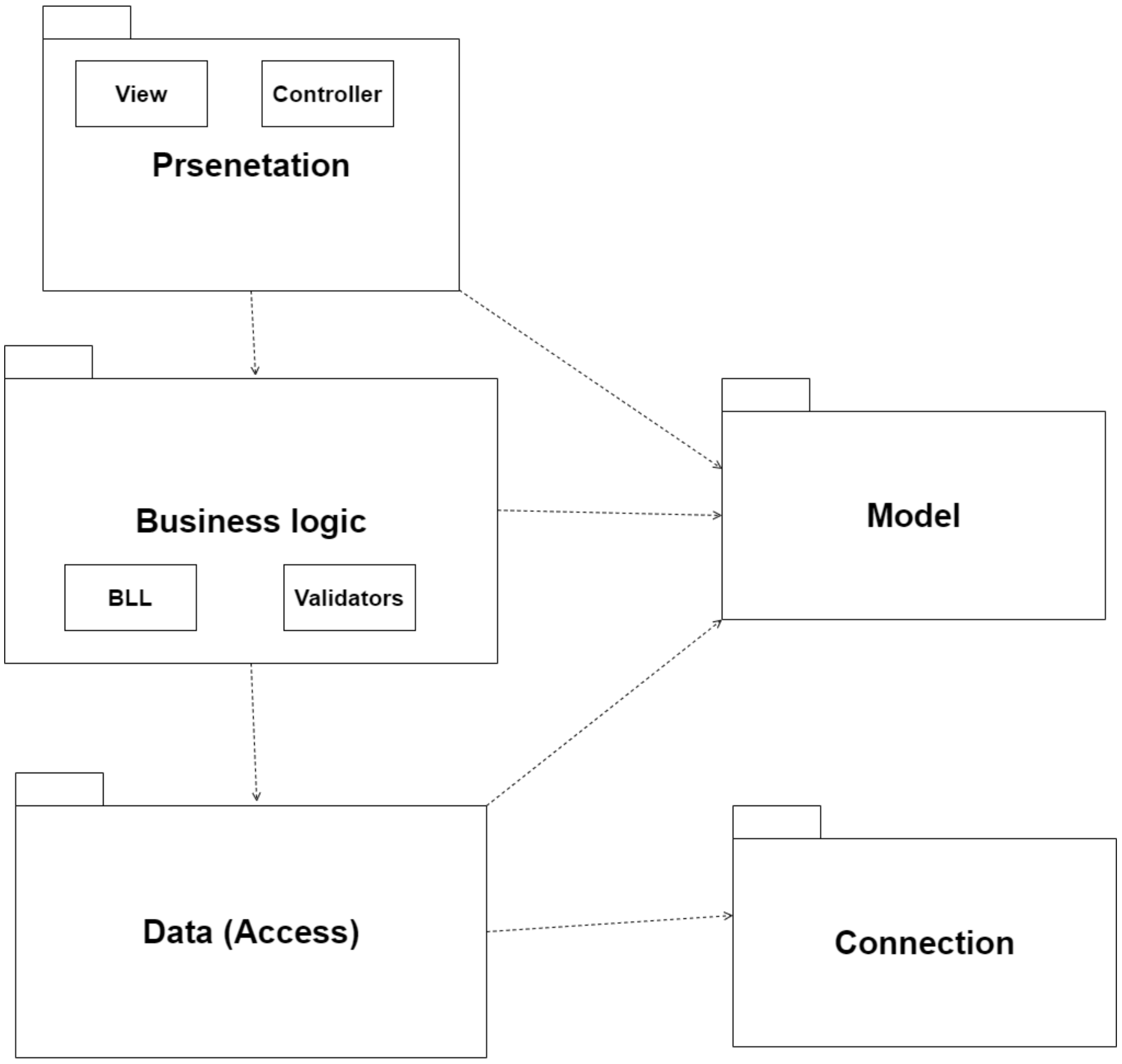
3. Design



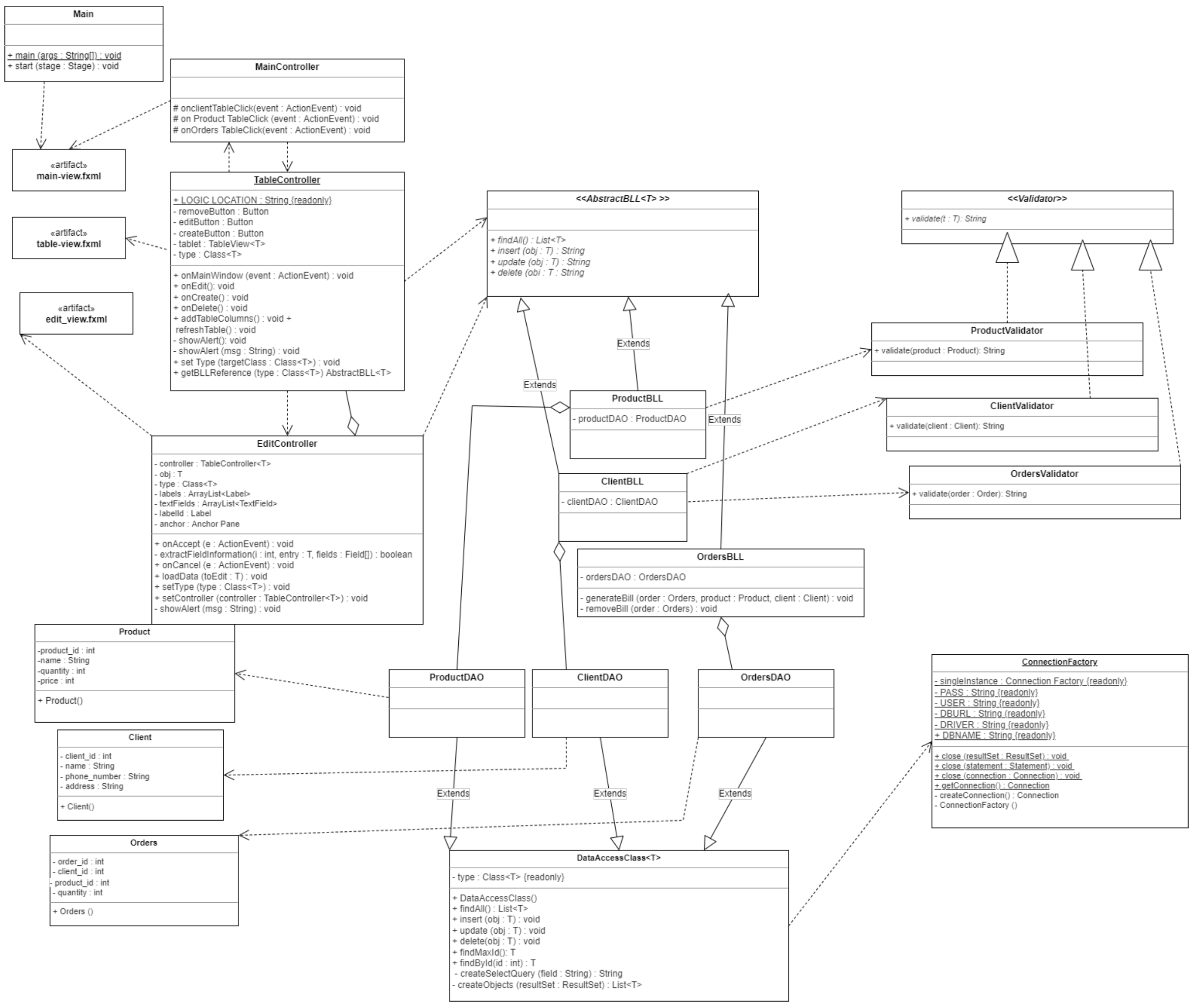
The orders management system has the following inputs and outputs: details on table entries, requested operation, table views, order bills and changes in the database.

Next, the design is broken down into packages:

- Controller: this package contains the classes that control the flow of the application. Because UI applications are event-driven, and JavaFX is no exception, this package decides what explicitly happens and waits for user input.
- Business_logic: Contains the business logic classes and the validators for each entry type that exists in the application. Only after the input is validated will the application proceed to make a request to the data access layer.
- Model: contains information about the way the data is modelled internally, i.e., the fields the entry types that are seen in the application have.
- Data: (or data access), this contains the classes that actually do operations in the database and have access to the connection.
- Connection: establishes a connection to the database. Without this class, there is no database to do queries and updates onto.
- View (implicit): this package includes the definition for the UI elements written in Oracle’s FXML language. It specifies the controller it is linked to for the action handlers.



The packages are further divided into classes. Such an illustration can be seen below:


Main
+ main (args : String[]) : void
+ start (stage : Stage) : void
MainController
# onclientTableClick(event : ActionEvent) : void
# on Product TableClick (event : ActionEvent) : void
# onOrders TableClick(event : ActionEvent) : void
«artifact» main-view.fxml
«artifact» table-view.fxml
«artifact» edit_view.fxml
TableController
+ LOGIC LOCATION : String {readonly}
- removeButton : Button
- editButton : Button
- createButton : Button
- tablet : TableView<T>
- type : Class<T>
+ onMainWindow (event : ActionEvent) : void
+ onEdit(): void
+ onCreate() : void
+ onDelete() : void
+ addTableColumns() : void + refreshTable() : void
- showAlert(): void
- showAlert (msg : String) : void
+ set Type (targetClass : Class<T>) : void
+ getBLLReference (type : Class<T>) AbstractBLL<T>
EditController
- controller : TableController<T>
- obj : T
- type : Class<T>
- labels : ArrayList<Label>
- textFields : ArrayList<TextField>
- labelId : Label
- anchor : Anchor Pane
+ onAccept (e : ActionEvent) : void
- extractFieldInformation(i : int, entry : T, fields : Field[]) : boolean
+ onCancel (e : ActionEvent) : void
+ loadData (toEdit : T) : void
+ setType (type : Class<T>) : void
+ setController (controller : TableController<T>) : void
- showAlert (msg : String) : void
<<AbstractBLL<T> >>
+ findAll() : List<T>
+ insert (obj : T) : String
+ update (obj : T) : String
+ delete (obi : T : String
<<Validator>>
+ validate(t : T): String
ProductValidator
+ validate(product : Product): String
ClientValidator
+ validate(client : Client): String
OrdersValidator
+ validate(order : Order): String
Extends
Extends
Extends
ProductBLL
- productDAO : ProductDAO
ClientBLL
- clientDAO : ClientDAO
OrdersBLL
- ordersDAO : OrdersDAO
- generateBill (order : Orders, product : Product, client : Client) : void
- removeBill (order : Orders) : void
Product
-product_id : int
-name : String
-quantity : int
-price : int
+ Product()
Client
- client_id : int
- name : String
- phone_number : String
- address : String
+ Client()
Orders
- order_id : int
- client_id : int
- product_id : int
- quantity : int
+ Orders ()
ProductDAO
ClientDAO
OrdersDAO
Extends
Extends
Extends
DataAccessClass<T>
- type : Class<T> {readonly}
+ DataAccessClass()
+ findAll() : List<T>
+ insert (obj : T) : void
+ update (obj : T) : void
+ delete(obj : T) : void
+ findMaxId(): T
+ findById(id : int) : T
- createSelectQuery (field : String) : String
- createObjects (resultSet : ResultSet) : List<T>
ConnectionFactory
- singleInstance : Connection Factory {readonly}
- PASS : String {readonly}
- USER : String {readonly}
- DBURL : String {readonly}
- DRIVER : String {readonly}
+ DBNAME : String {readonly}
+ close (resultSet : ResultSet) : void
+ close (statement : Statement) : void
+ close (connection : Connection) : void
+ getConnection() : Connection
- createConnection() : Connection
- ConnectionFactory ()

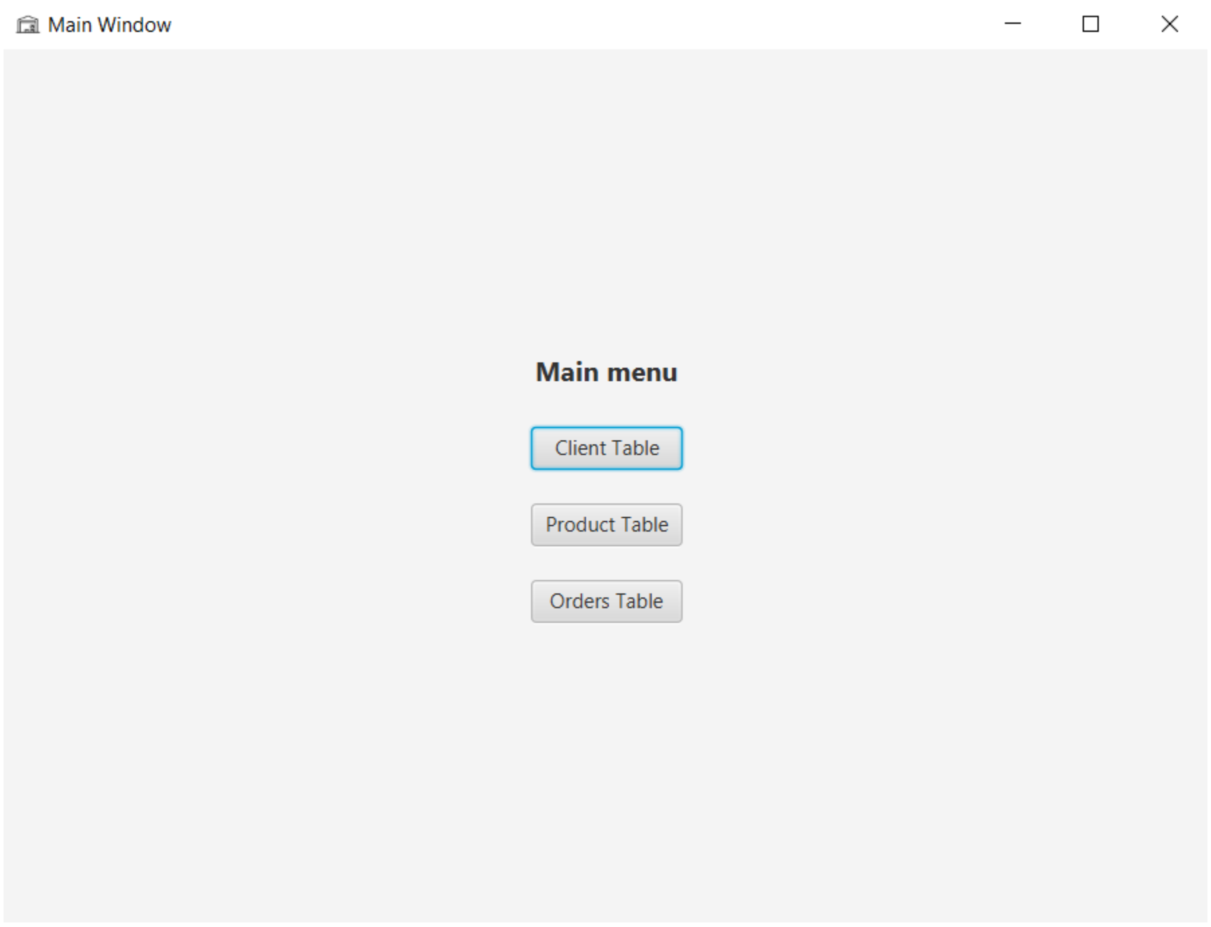
The data structure I considered for representing the model is a client, orders, and product table. The client table has an id, name, phone number and address. The product table has an id, name, quantity, and price. The orders table has an order id, a corresponding product id and client id, and a quantity.

The Controller switch between the windows.

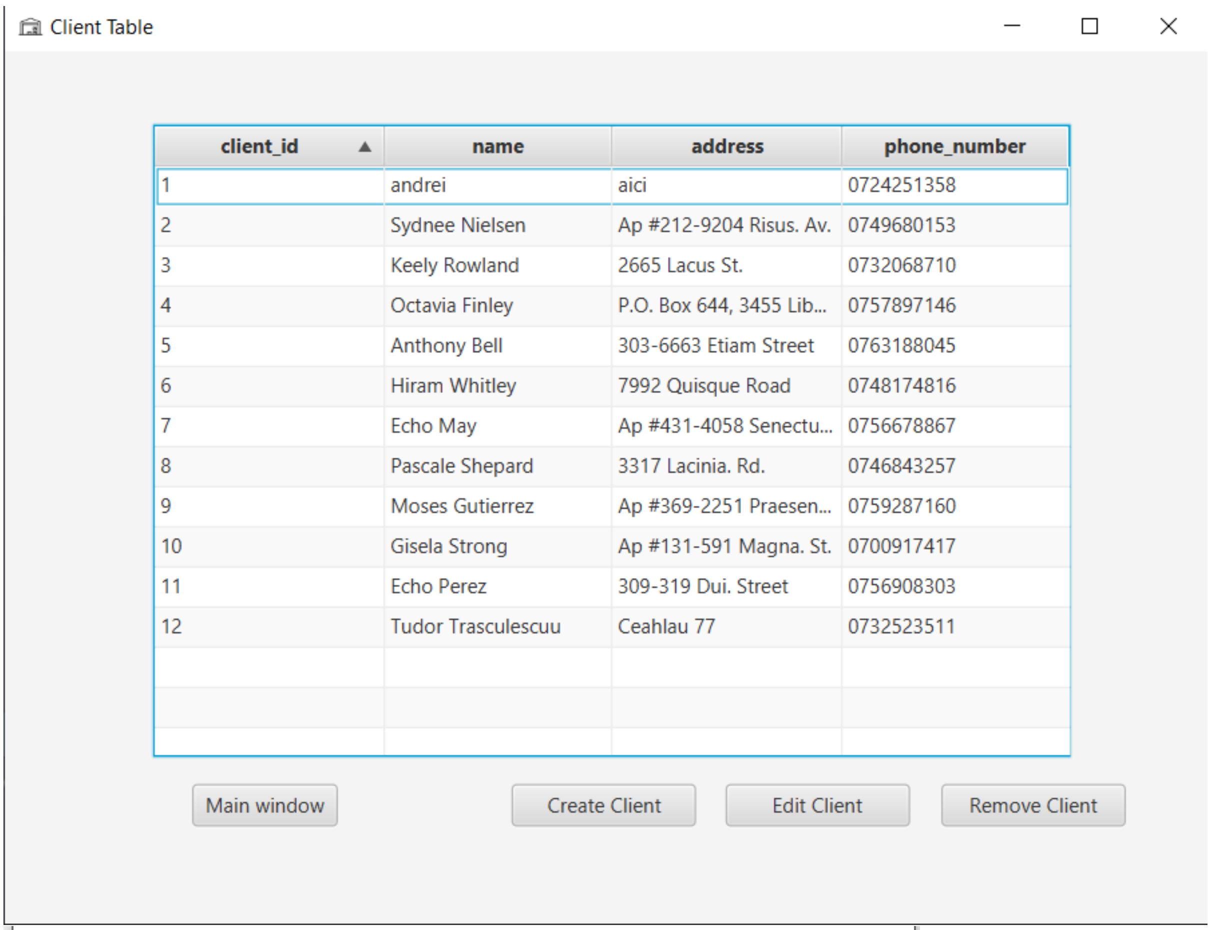
No special algorithms were used for this assignment

4. Implementation

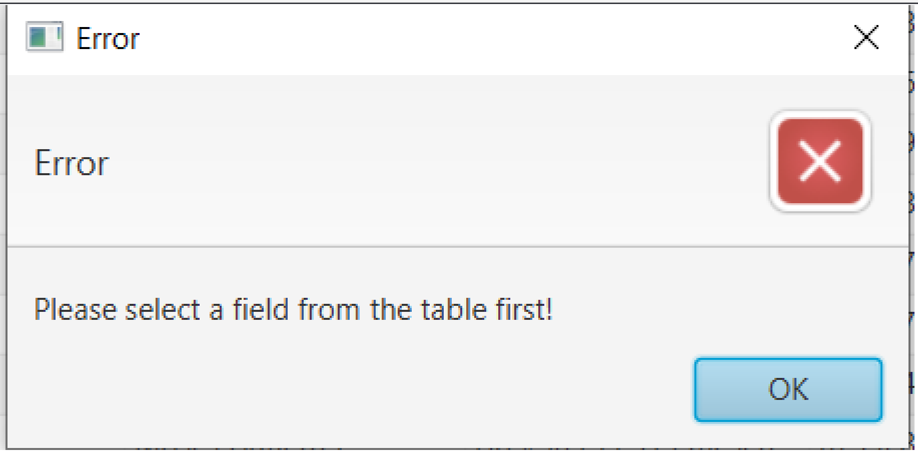
The main **GUI** window can be seen below. It consists of 3 simple buttons. Each button is tied to an action handler which will take the user to one of the 3 tables for operations: to products, orders or clients. There is a warehouse icon in the top left-hand corner.



One of the table views look something like this. In the middle we have a table view object. The columns are dynamically generated through reflection and there is only one FXML file that describes all three of the views. The button text is created dynamically when the scene is loaded behind the scenes. Thus, code reuse is favored, and time is saved on both maintenance and upgradeability. The first 2 buttons when successful load a form for entering data. If the user makes no selection in the table, remove, and edit buttons will throw an alert and the operation will be aborted.


Client Table

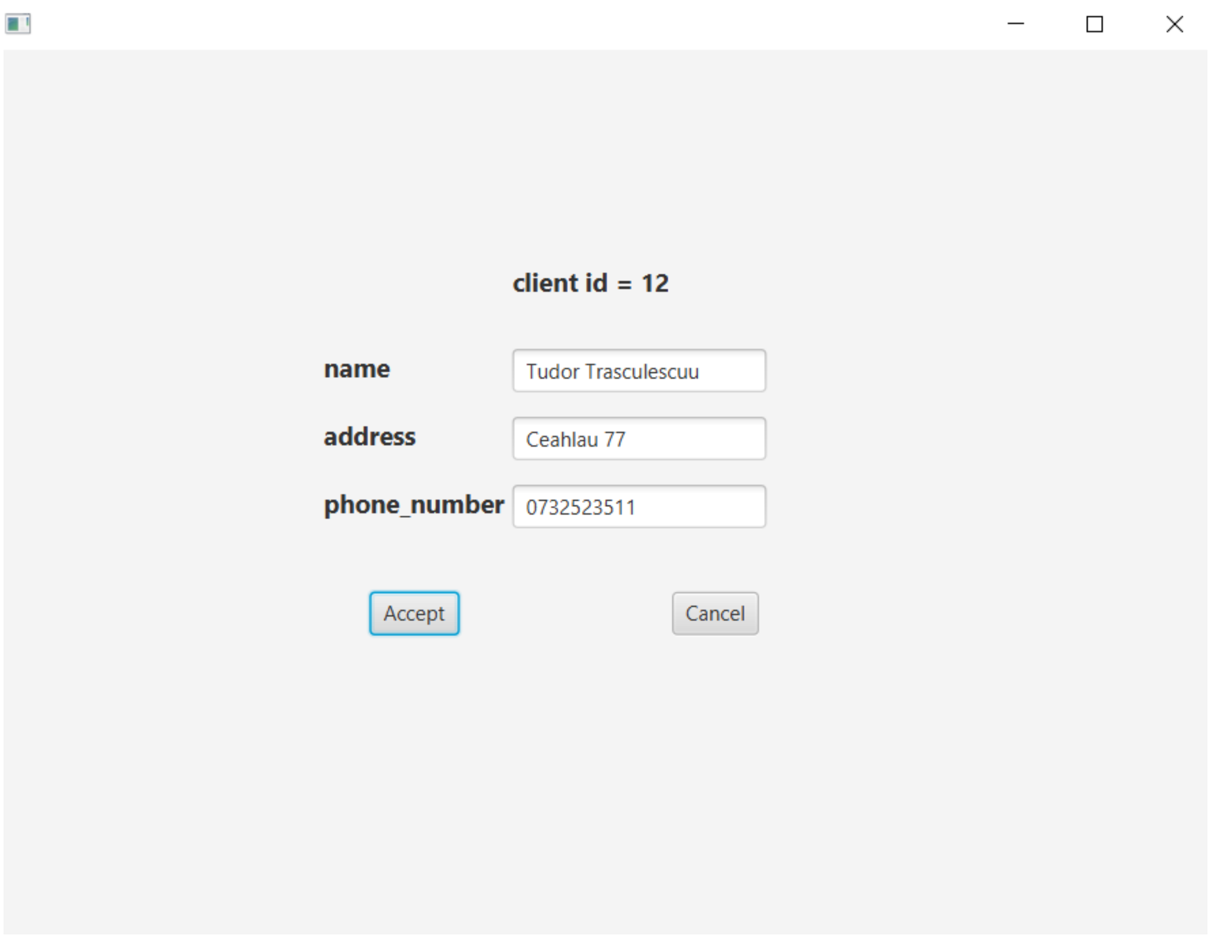
client_id	name	address	phone_number
1	andrei	aici	0724251358
2	Sydnee Nielsen	Ap #212-9204 Risus. Av.	0749680153
3	Keely Rowland	2665 Lacus St.	0732068710
4	Octavia Finley	P.O. Box 644, 3455 Lib...	0757897146
5	Anthony Bell	303-6663 Etiam Street	0763188045
6	Hiram Whitley	7992 Quisque Road	0748174816
7	Echo May	Ap #431-4058 Senectu...	0756678867
8	Pascale Shepard	3317 Lacinia. Rd.	0746843257
9	Moses Gutierrez	Ap #369-2251 Praesen...	0759287160
10	Gisela Strong	Ap #131-591 Magna. St.	0700917417
11	Echo Perez	309-319 Dui. Street	0756908303
12	Tudor Trasculescuu	Ceahlau 77	0732523511


Error

Error

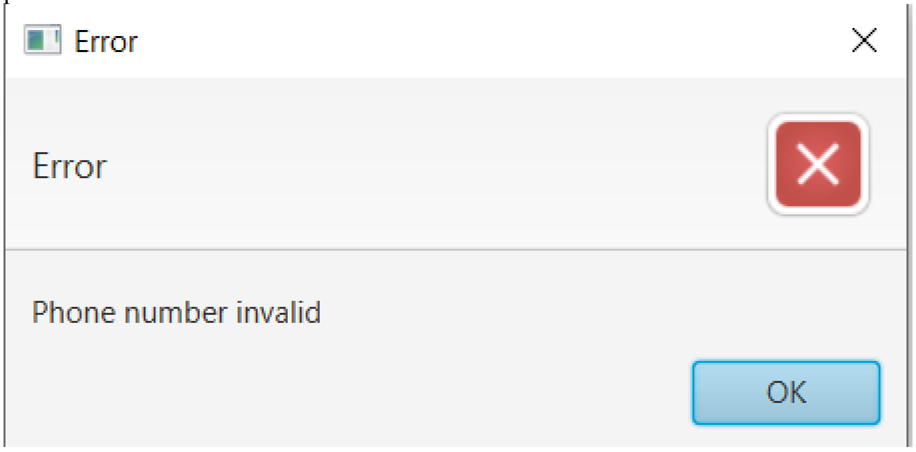
Please select a field from the table first!

Below is pictured what happens when the user wants to edit an entry in the table. The id is written at the top and the fields are populated with the selection from the table.


client id = 12

name	Tudor Trasculescuu
address	Ceahlau 77
phone_number	0732523511

An error will be returned if invalid data is introduced or if the data cannot be parsed. Extra error checks are made for the orders class because there is additional logic with respect to the product stock. Also, clients and products cannot be deleted unless there are no orders in the table that hold a reference to that specific client or product.

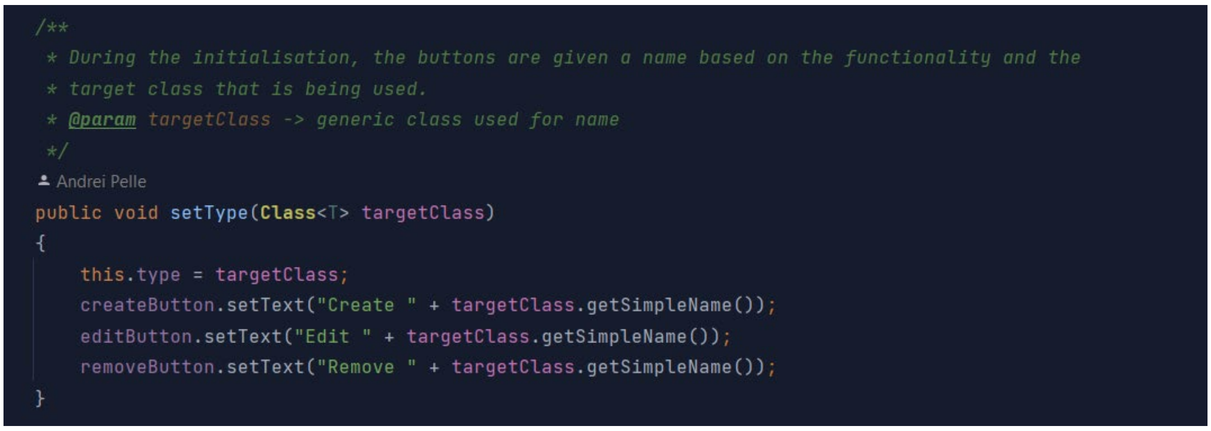

Error

Error

Phone number invalid

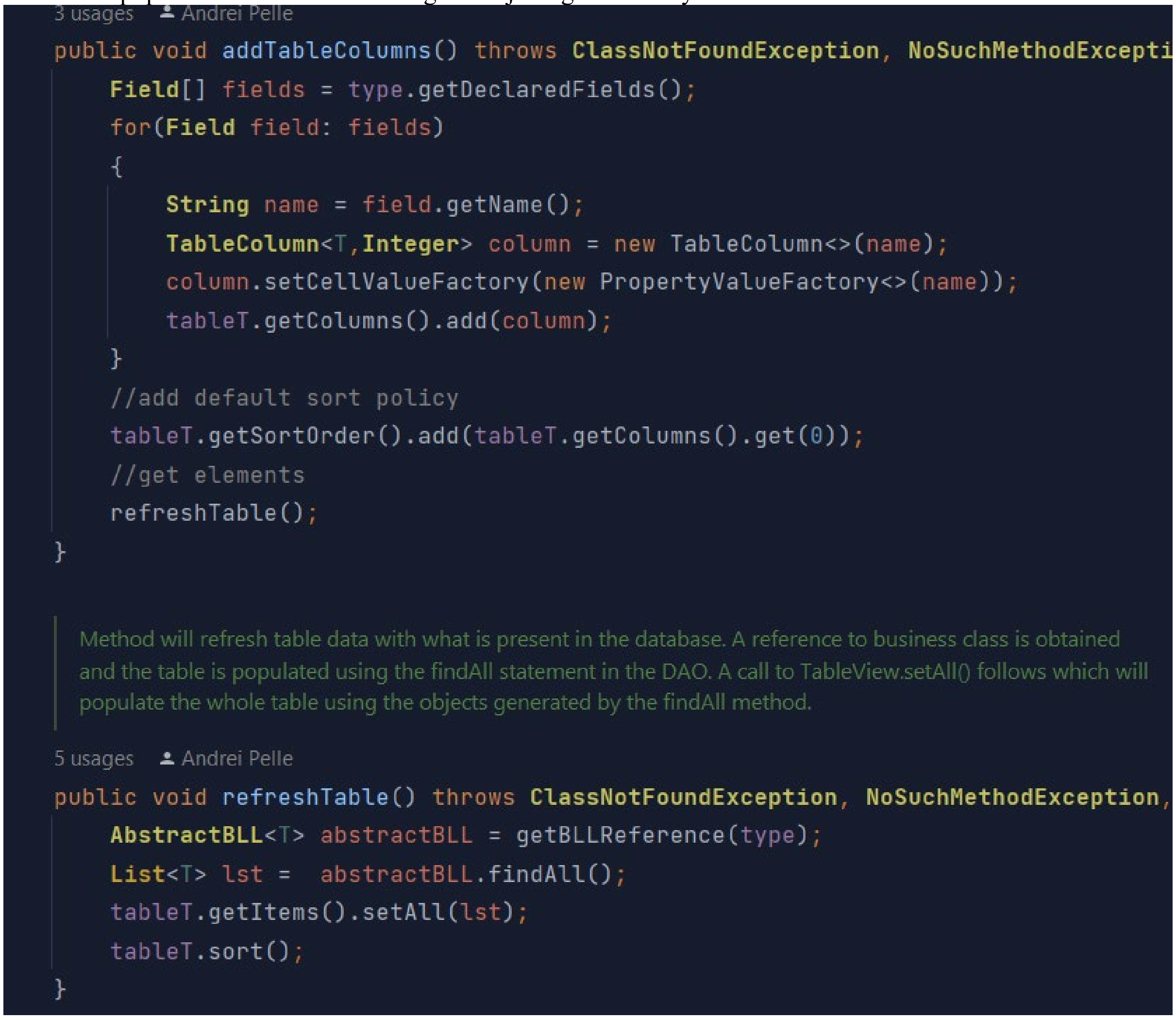
The **Main Controller** the main window. Very simple, has only 3 buttons. Clicking any of the buttons leads to the scene changing to the corresponding one. The scenes have a button that when pressed will cause a return to this main window. No other notable methods.

Following is the generic **table controller** for the table. The table will be shown on this scene, alongside with the buttons for create/update/delete.
Type parameters: <T> – -> type for table
Has 3 button FXML : create, edit and remove. They are set in a function that initializes the type.



```
/**
 * During the initialisation, the buttons are given a name based on the functionality and the
 * target class that is being used.
 * @param targetClass -> generic class used for name
 */
Andrei Pelle
public void setType(Class<T> targetClass)
{
    this.type = targetClass;
    createButton.setText("Create " + targetClass.getSimpleName());
    editButton.setText("Edit " + targetClass.getSimpleName());
    removeButton.setText("Remove " + targetClass.getSimpleName());
}
```

The following methods generates the table header using reflection. Columns are added for each field in the object and the table sorting strategy is set to be sorted on id. Then, the method to populate the table with data is called. Method will refresh table data with what is present in the database. A reference to business class is obtained and the table is populated using the findAll statement in the DAO. A call to TableView.setAll() follows which will populate the whole table using the objects generated by the findAll method.

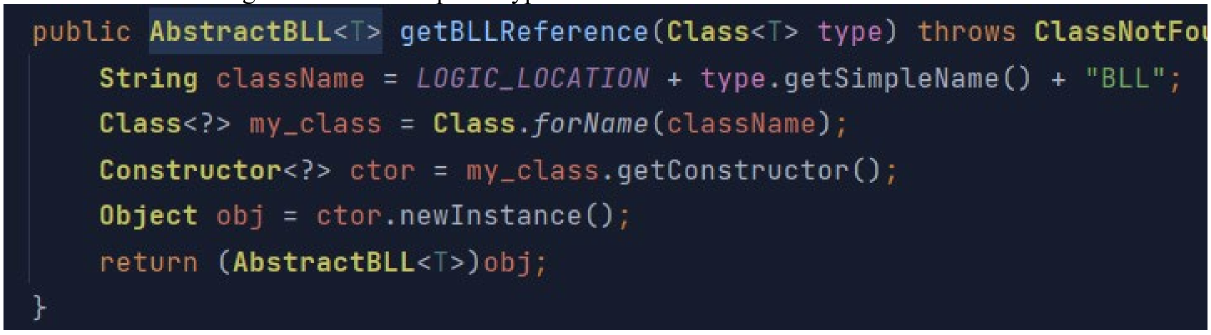


```
3 usages   Andrei Pelle
public void addTableColumns() throws ClassNotFoundException, NoSuchMethodExcepti
    Field[] fields = type.getDeclaredFields();
    for(Field field: fields)
    {
        String name = field.getName();
        TableColumn<T,Integer> column = new TableColumn<>(name);
        column.setCellValueFactory(new PropertyValueFactory<>(name));
        tableT.getColumns().add(column);
    }
    //add default sort policy
    tableT.getSortOrder().add(tableT.getColumns().get(0));
    //get elements
    refreshTable();
}

    Method will refresh table data with what is present in the database. A reference to business class is obtained
    and the table is populated using the findAll statement in the DAO. A call to TableView.setAll() follows which will
    populate the whole table using the objects generated by the findAll method.

5 usages   Andrei Pelle
public void refreshTable() throws ClassNotFoundException, NoSuchMethodException,
    AbstractBLL<T> abstractBLL = getBLLReference(type);
    List<T> lst =  abstractBLL.findAll();
    tableT.getItems().setAll(lst);
    tableT.sort();
}
```

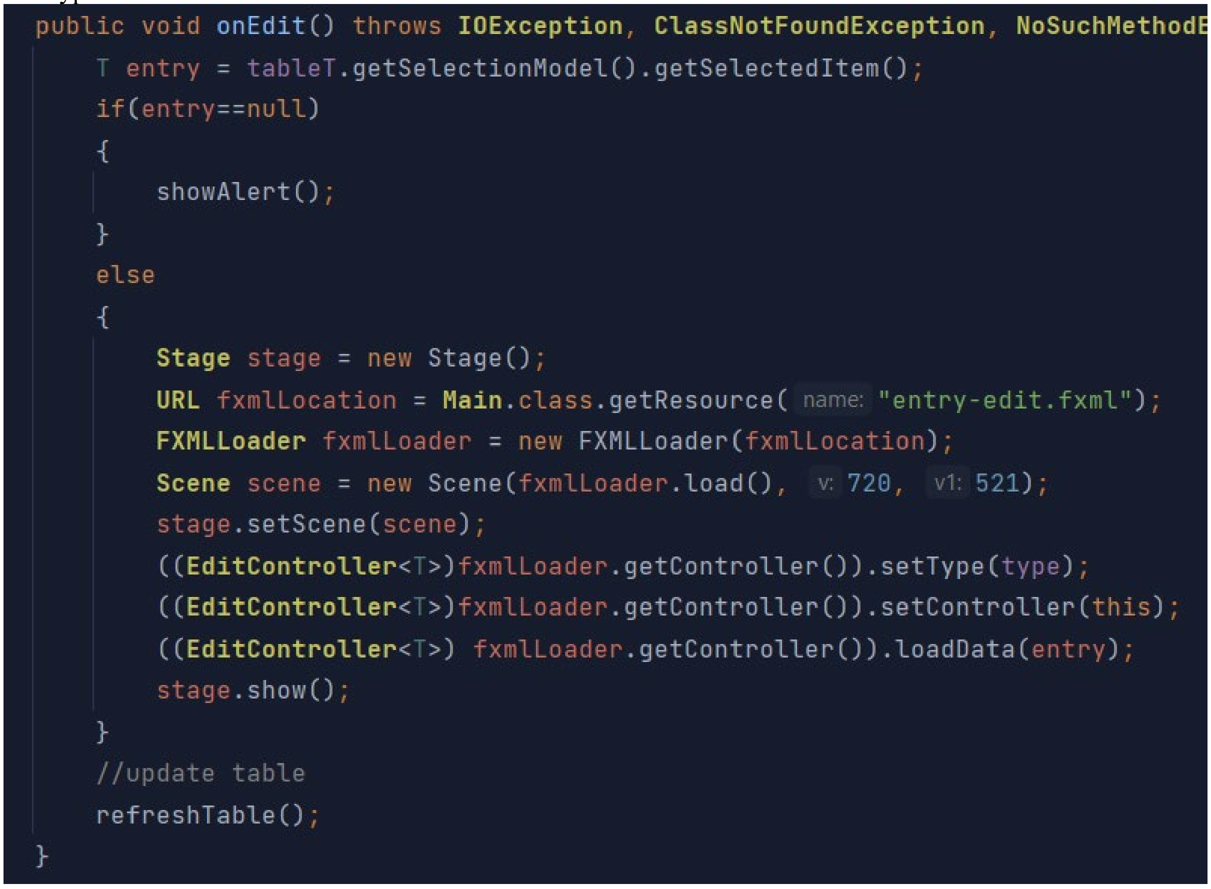
The most important method in this controller is the getBLLReference method which will dynamically search the logic package in order to find correct business logic class associated with the parameter. An abstract type will be returned, however the created object will be of type specific to the generic, not the abstract, so the called function will be desired ones thanks to run-time polymorphism provided by java.
Params: type – -> type of the business logic class that is being searched
Returns:-> business logic class of the required type



```
public AbstractBLL<T> getBLLReference(Class<T> type) throws ClassNotFou
    String className = LOGIC_LOCATION + type.getSimpleName() + "BLL";
    Class<?> my_class = Class.forName(className);
    Constructor<?> ctor = my_class.getConstructor();
    Object obj = ctor.newInstance();
    return (AbstractBLL<T>)obj;
}
```

I will describe just one of the instances of create, edit and delete and the rest of them are similar. First the selection from the table is fetched. If no selection has been made, then an alert is shown, and the operation will not continue.

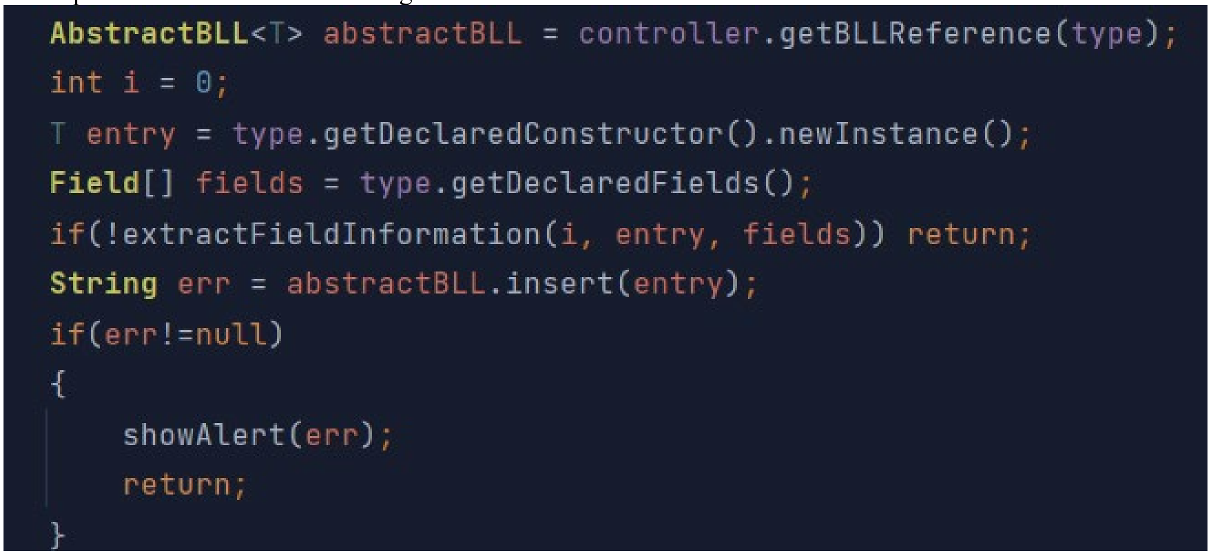
In the case of success, a new edit stage is created, and the information is loaded into it for further processing. The type is also set for the controller.



```
public void onEdit() throws IOException, ClassNotFoundException, NoSuchMethodE
    T entry = tableT.getSelectionModel().getSelectedItem();
    if(entry==null)
    {
        showAlert();
    }
    else
    {
        Stage stage = new Stage();
        URL fxmlLocation = Main.class.getResource( name: "entry-edit.fxml");
        FXMLLoader fxmlLoader = new FXMLLoader(fxmlLocation);
        Scene scene = new Scene(fxmlLoader.load(),  v: 720,  v1: 521);
        stage.setScene(scene);
        ((EditController<T>)fxmlLoader.getController()).setType(type);
        ((EditController<T>)fxmlLoader.getController()).setController(this);
        ((EditController<T>) fxmlLoader.getController()).loadData(entry);
        stage.show();
    }
    //update table
    refreshTable();
}
```

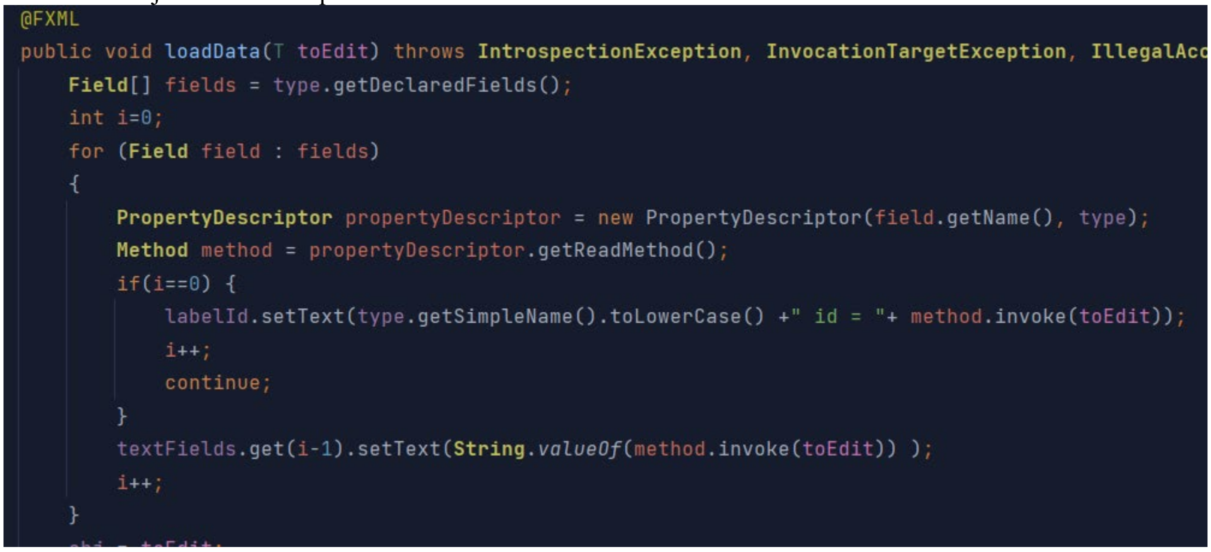
This edits controller class provides the Controller for the creation/update window. The text fields are dynamically created through reflection and placed at predefined positions on the window. Whenever an error is returned by any logic/database method calls, an alert will be shown that describes the error that was encountered.
Type parameters: <T> – -> type to be edited

If the user accepts the edit, creation, the field information will be extracted with reflection, an object will be created with the required fields and then passed onto the database for further processing. If a string was returned, showAlert() will be called and an alert will be displayed on the screen.
Params:
e – -> parameter used to close the stage



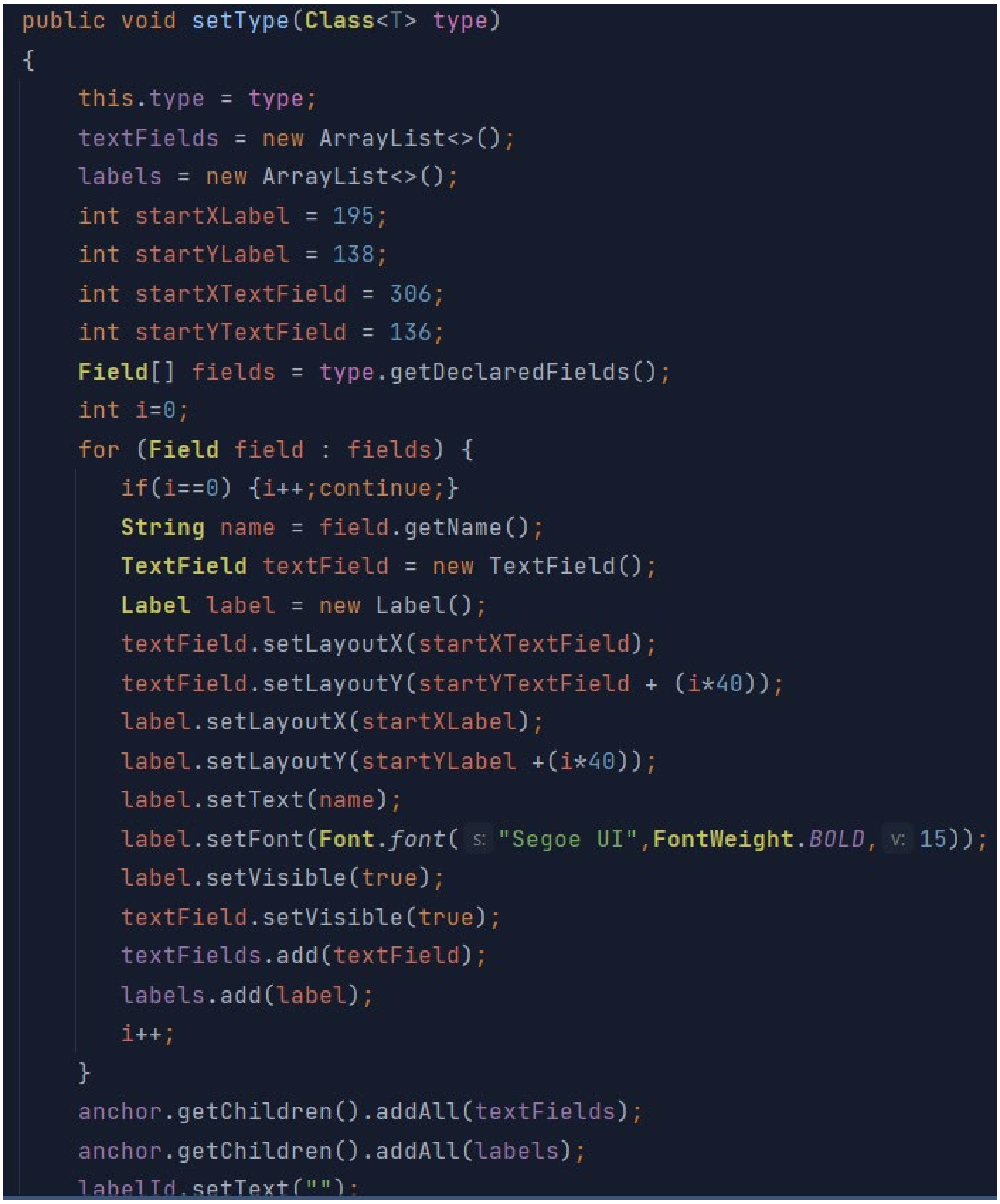
```java
AbstractBLL<T> abstractBLL = controller.getBLLReference(type);
int i = 0;
T entry = type.getDeclaredConstructor().newInstance();
Field[] fields = type.getDeclaredFields();
if(!extractFieldInformation(i, entry, fields)) return;
String err = abstractBLL.insert(entry);
if(err!=null)
{
    showAlert(err);
    return;
}
```

The load data method loads the object data in the text fields if the edit option is used.
The object received as parameter will be used to populate the text fields (which are stored in the TextField array) with the information already present in the database. Thus, it will be clear to the user what properties are actually changing and which are left as-is. The label is also set when i==0, however this is not an editable field for the user.
Params:
toEdit – -> object with the required information



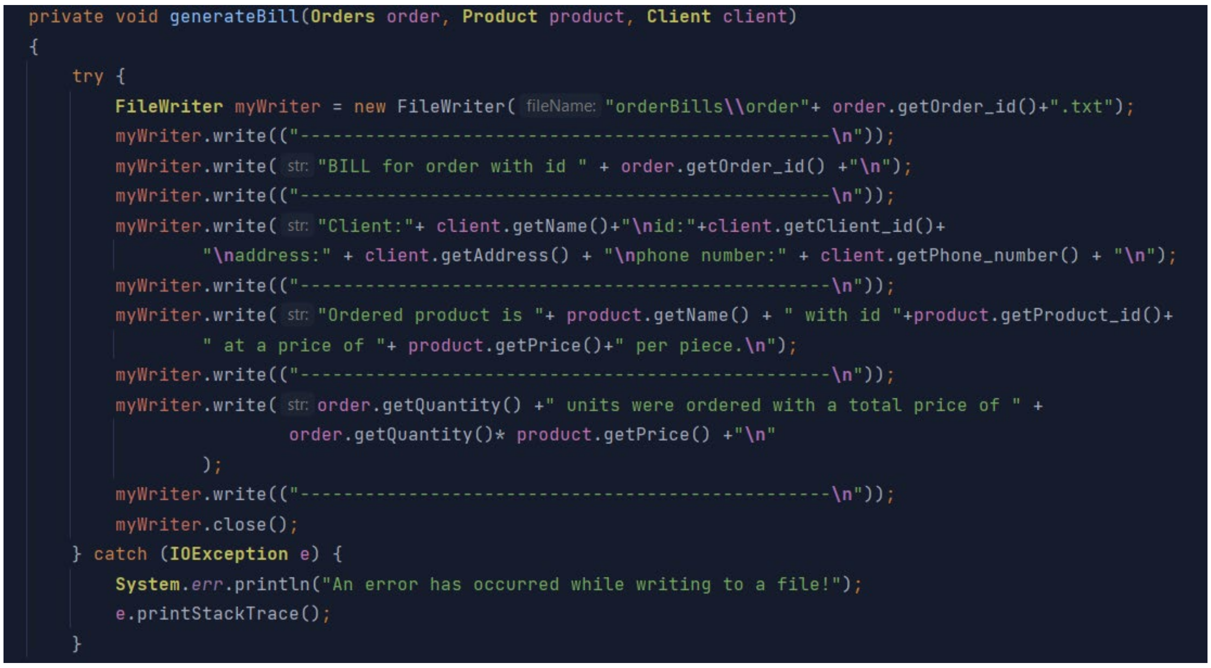
```java
@FXML
public void loadData(T toEdit) throws IntrospectionException, InvocationTargetException, IllegalAcc
    Field[] fields = type.getDeclaredFields();
    int i=0;
    for (Field field : fields)
    {
        PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);
        Method method = propertyDescriptor.getReadMethod();
        if(i==0) {
            labelId.setText(type.getSimpleName().toLowerCase() +" id = "+ method.invoke(toEdit));
            i++;
            continue;
        }
        textFields.get(i-1).setText(String.valueOf(method.invoke(toEdit)) );
        i++;
    }
```

Initially, the window has only the cancel and accept buttons. I generated the rest of the fields using reflection, set them at a certain x and y from the chosen starting location. The UI elements are held in a variable size list of label and text fields.
Params:
type – -> type parameter for controller: Order, Product or Client



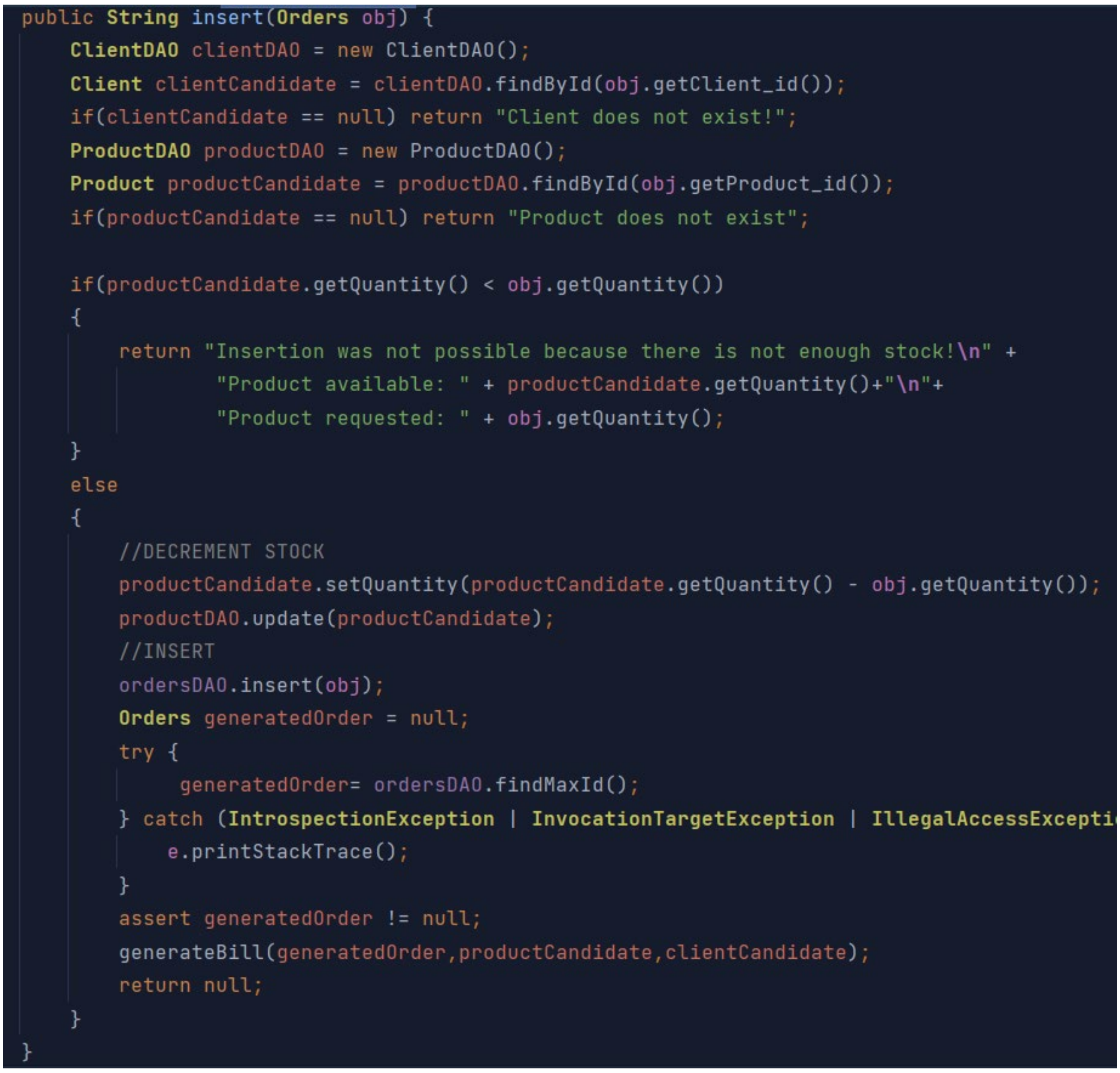
```
public void setType(Class<T> type)
{
    this.type = type;
    textFields = new ArrayList<>();
    labels = new ArrayList<>();
    int startXLabel = 195;
    int startYLabel = 138;
    int startXTextField = 306;
    int startYTextField = 136;
    Field[] fields = type.getDeclaredFields();
    int i=0;
    for (Field field : fields) {
        if(i==0) {i++;continue;}
        String name = field.getName();
        TextField textField = new TextField();
        Label label = new Label();
        textField.setLayoutX(startXTextField);
        textField.setLayoutY(startYTextField + (i*40));
        label.setLayoutX(startXLabel);
        label.setLayoutY(startYLabel +(i*40));
        label.setText(name);
        label.setFont(Font.font( s: "Segoe UI",FontWeight.BOLD, v: 15));
        label.setVisible(true);
        textField.setVisible(true);
        textFields.add(textField);
        labels.add(label);
        i++;
    }
    anchor.getChildren().addAll(textFields);
    anchor.getChildren().addAll(labels);
    labelId.setText("");
```

The bill generations happen in the order BLL. This method generates a bill for the required order. It also calculates the total order cost based on the unit price of each product. This sum is appended at the end of the file. Each file is dynamically created to contain
Params:
order – > order details to be written product
product – > product details to be written client
client – > client details to be written



```java
private void generateBill(Orders order, Product product, Client client)
{
    try {
        FileWriter myWriter = new FileWriter( fileName: "orderBills\\order"+ order.getOrder_id()+".txt");
        myWriter.write(("-------------------------------------------------\n"));
        myWriter.write( str: "BILL for order with id " + order.getOrder_id() +"\n");
        myWriter.write(("-------------------------------------------------\n"));
        myWriter.write( str: "Client:"+ client.getName()+"\nid:"+client.getClient_id()+
                "\naddress:" + client.getAddress() + "\nphone number:" + client.getPhone_number() + "\n");
        myWriter.write(("-------------------------------------------------\n"));
        myWriter.write( str: "Ordered product is "+ product.getName() + " with id "+product.getProduct_id()+
                " at a price of "+ product.getPrice()+" per piece.\n");
        myWriter.write(("-------------------------------------------------\n"));
        myWriter.write( str: order.getQuantity() +" units were ordered with a total price of " +
                        order.getQuantity()* product.getPrice() +"\n"
                );
        myWriter.write(("-------------------------------------------------\n"));
        myWriter.close();
    } catch (IOException e) {
        System.err.println("An error has occurred while writing to a file!");
        e.printStackTrace();
    }
```

An order can only be inserted if the specified client/product exists and there is enough stock. If the checks are successful, a bill is generated in a .txt format with the according order number. The error code is returned as a string.
Params:
obj – - order to insert
Returns: error code as string

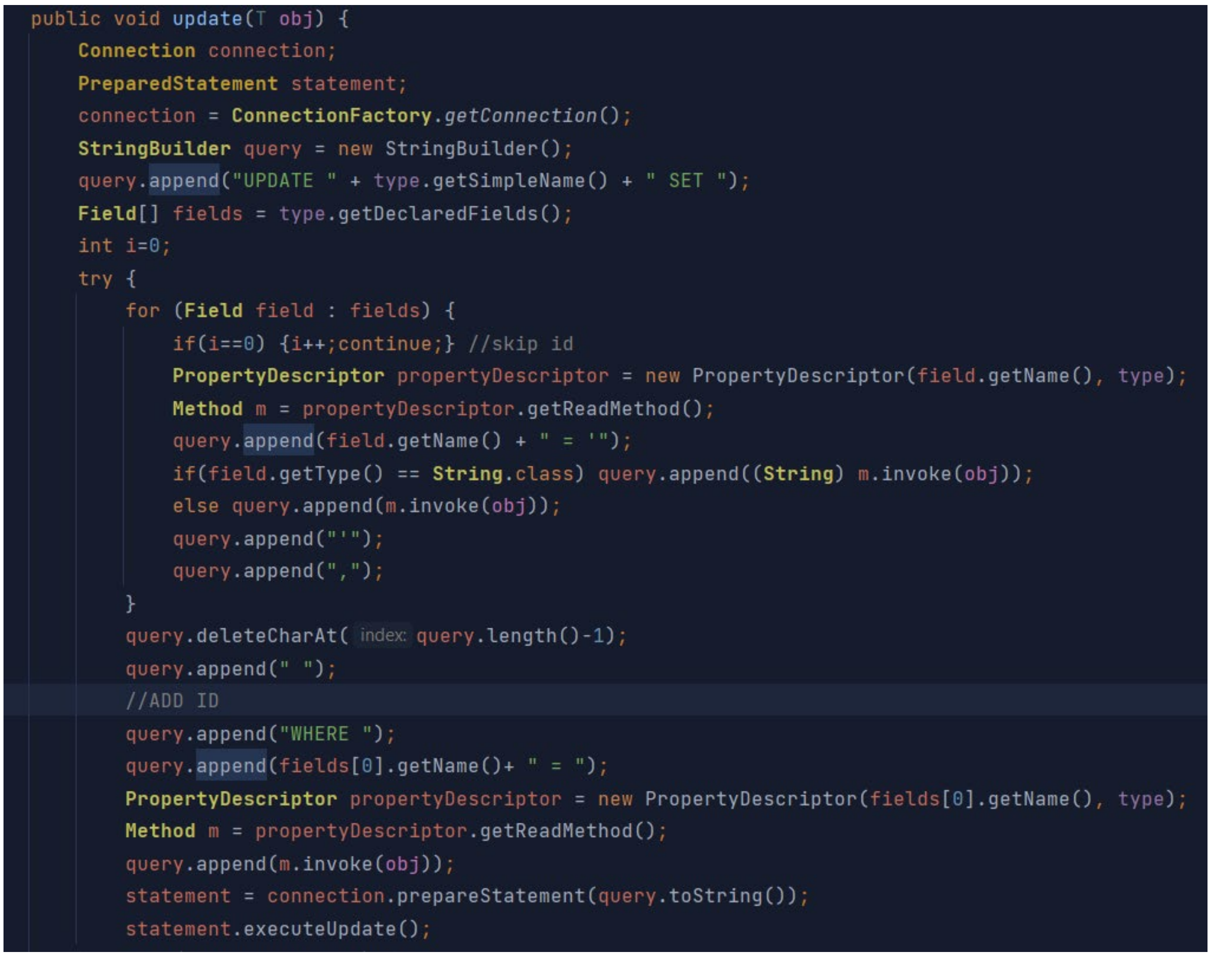


```java
public String insert(Orders obj) {
    ClientDAO clientDAO = new ClientDAO();
    Client clientCandidate = clientDAO.findById(obj.getClient_id());
    if(clientCandidate == null) return "Client does not exist!";
    ProductDAO productDAO = new ProductDAO();
    Product productCandidate = productDAO.findById(obj.getProduct_id());
    if(productCandidate == null) return "Product does not exist";

    if(productCandidate.getQuantity() < obj.getQuantity())
    {
        return "Insertion was not possible because there is not enough stock!\n" +
                "Product available: " + productCandidate.getQuantity()+"\n"+
                "Product requested: " + obj.getQuantity();
    }
    else
    {
        //DECREMENT STOCK
        productCandidate.setQuantity(productCandidate.getQuantity() - obj.getQuantity());
        productDAO.update(productCandidate);
        //INSERT
        ordersDAO.insert(obj);
        Orders generatedOrder = null;
        try {
             generatedOrder= ordersDAO.findMaxId();
        } catch (IntrospectionException | InvocationTargetException | IllegalAccessExcepti
            e.printStackTrace();
        }
        assert generatedOrder != null;
        generateBill(generatedOrder,productCandidate,clientCandidate);
        return null;
    }
}
```

The operations in the **DataAccessClass** are similar to each other. I will only comment on update and the rest are similar.

The update method will edit an object already present in the database. The fields are already updated by the controller layer, so we don't care about any of that. We will set all the fields as given. Thus, we call the field read method, read the properties, append them and then call the statement. Big difference is that this time we have an id which shall be used in the WHERE clause to identify the object in the database.
Params:
obj – -> object to be updated in the database. Does have an id, it shall be used to uniquely identify the object.



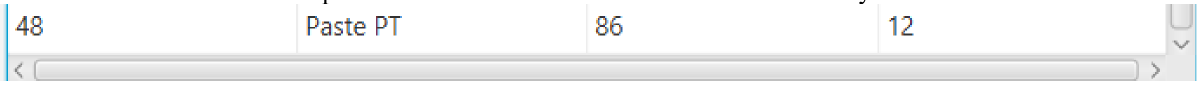
```java
public void update(T obj) {
    Connection connection;
    PreparedStatement statement;
    connection = ConnectionFactory.getConnection();
    StringBuilder query = new StringBuilder();
    query.append("UPDATE " + type.getSimpleName() + " SET ");
    Field[] fields = type.getDeclaredFields();
    int i=0;
    try {
        for (Field field : fields) {
            if(i==0) {i++;continue;} //skip id
            PropertyDescriptor propertyDescriptor = new PropertyDescriptor(field.getName(), type);
            Method m = propertyDescriptor.getReadMethod();
            query.append(field.getName() + " = '");
            if(field.getType() == String.class) query.append((String) m.invoke(obj));
            else query.append(m.invoke(obj));
            query.append("'");
            query.append(",");
        }
        query.deleteCharAt( index: query.length()-1);
        query.append(" ");
        //ADD ID
        query.append("WHERE ");
        query.append(fields[0].getName()+ " = ");
        PropertyDescriptor propertyDescriptor = new PropertyDescriptor(fields[0].getName(), type);
        Method m = propertyDescriptor.getReadMethod();
        query.append(m.invoke(obj));
        statement = connection.prepareStatement(query.toString());
        statement.executeUpdate();
```

5. Results

For testing purposes, I used the GUI alongside some data from the databases to test things. Below you can see the example of a bill successfully created:

```
-------------------------------------------------
BILL for order with id 7
-------------------------------------------------
Client: Anthony Bell
id:5
address:303-6663 Etiam Street
phone number:0763188045
-------------------------------------------------
Ordered product is Paste PT with id 48 at a price of 12 per piece.
-------------------------------------------------
14 units were ordered with a total price of 168
```

The product stock before the operation was initialized with the value of 100. After the order is created, the size was decreased to 86. The total price of the order has also been calculated correctly.



48	Paste PT	86	12

6. Conclusions

This homework was a good introduction to database management systems in java, and it helped me understand these mechanisms in java better, also it was a breeze to work with such mechanisms since they adhere to the java high level way of doing things so many things about databases that would usually be kind of hard were actually quite easy. The connection classes were already written, I just had to troubleshoot them when they refused to work.

As a future improvement, I could stylize the GUI more using CSS and perhaps show the border in red for invalid input like I did in the first assignment, perhaps provide a way for users to directly write into the table for creating and editing or even deleting table entries using the delete key instead of having to press the delete button.

7. Bibliography

1. https://dsrl.eu/courses/pt/ and everything contained within assignment 3 support presentation, assignment 3 resources, lecture notes, etc.
2. Java Programming Masterclass by Tim Buchalka – a course available on Udemy about Java concepts and programming techniques
3. https://stackoverflow.com/questions/tagged/java
4. Stack overflow for information on java databases and how to use table view and such
5. https://docs.oracle.com/javafx/2/api/javafx/scene/control/TableView.html
6. https://en.wikipedia.org/wiki/Non-functional_requirement#Examples