Assignment 3 documentation
Warehouse management system

Litan Virgil

Group 30425

TA: Claudia Pop

Contents

1. Functional Requirements 3
2. Objectives 3
    2.1 Main Objective 3
    2.2 Secondary Objectives 3
3. Problem analysis 3
4. Design 3
    4.1 Data structures 3
    4.2 Class diagram 3
    4.3 Algorithms 3
5. Implementation 4
6 Conclusions and Further Development 9

1. Functional Requirements

The application should allow order management for processing customer orders for a warehouse. The user should be able to insert customers and products and also assign products to customers through orders.

Also, bills can be generated for each order that is currently stored inside the database.

An aspect of paramount importance is designing the application using a layered architecture.

2. Objectives

Design and implement a application considering an order management application for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders.

2.1 Main objectives

Developing an application be means of layered architecture design and create a correct data base structure.

2.2 Secondary objectives

High accessibility of the database values and tables in order to create different edge-case scenarios that enhance the testing process.

Another important aspect of the application is designing a comprehensible set of logs that would allow an advanced user to fully understand the mechanisms that underlie the interactions between the layers of the process.

Another objective would be designing the application such a way that it could be easily further developed by separating the models, the general user interface and the main controllers.

3. Problem analysis

The main use case is an administrator creating Customers and Products and creating links between those using orders. The main actors being the user, the main controller, the bussines logic layer (BLL) and the data acces layer (DAL). The user selects the customer table, the selects a customer (the one that will have a new order). After the customer is selected the user should click the button " Select ". The application stores the selected customer id and creates a new order for the customer, also switches the table to the one with Products. After that, the user has to select a product and click select, the product will be added to the current opened order and the stock will be decremented (instantly the modification will appear in the database), if the product is out of stock a message will be displayed in the console log. After all desired products are selected, the user should press the button end task.

Another use case is the administrator wanting to insert, update or delete records. In order to do that one should select the desired target and click one of the dedicated buttons for that operation. After this, a window will pop up and the data should be entered, then the " done " button will be clicked.

Lastly, the " generate reports " button is used to generate a text file for each order from the table containing all the products in that order and the total price.

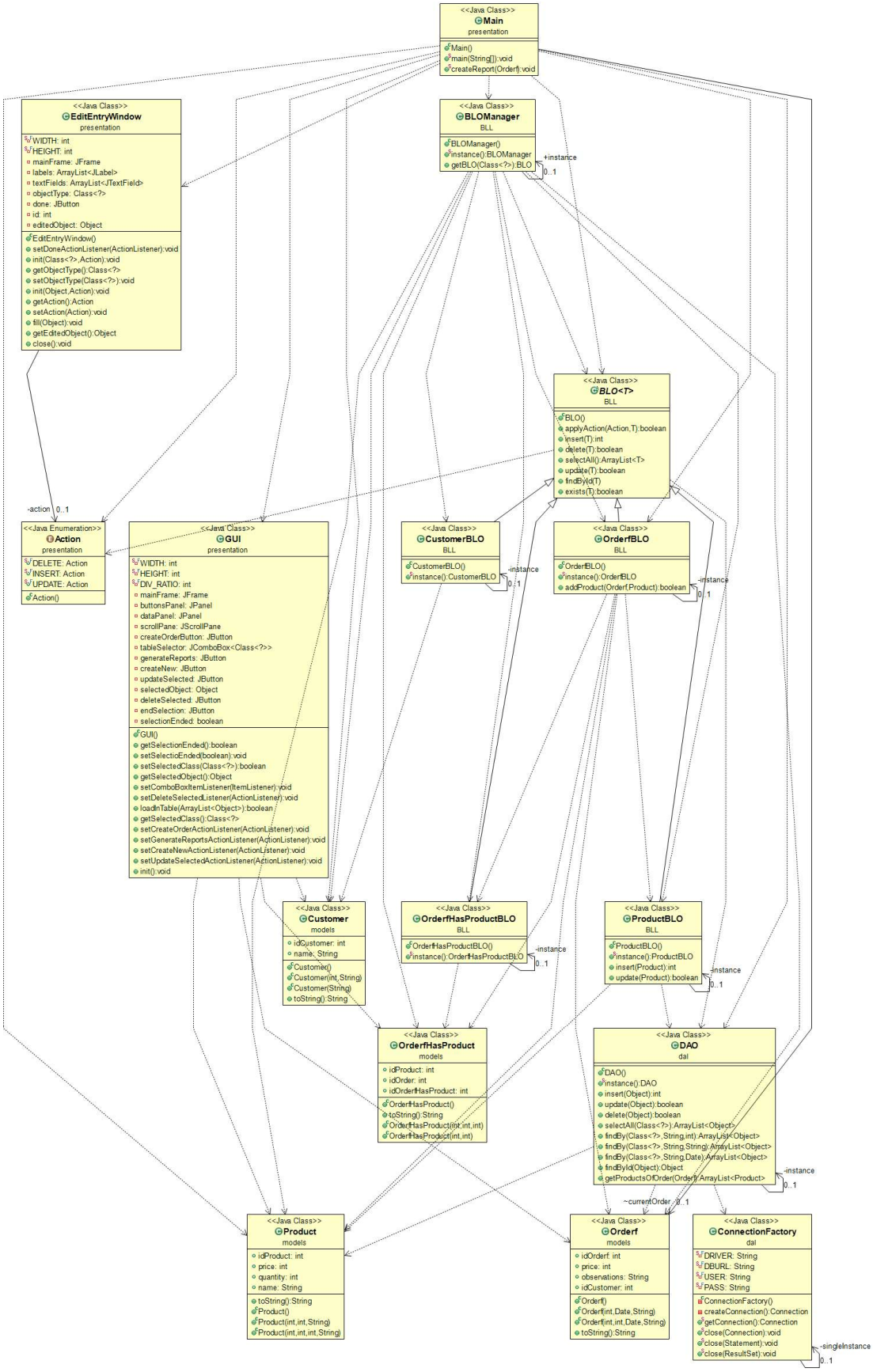
4. Design

4.1 Data structures

The used data structures are basic arrays and lists.

4.2 Class diagram

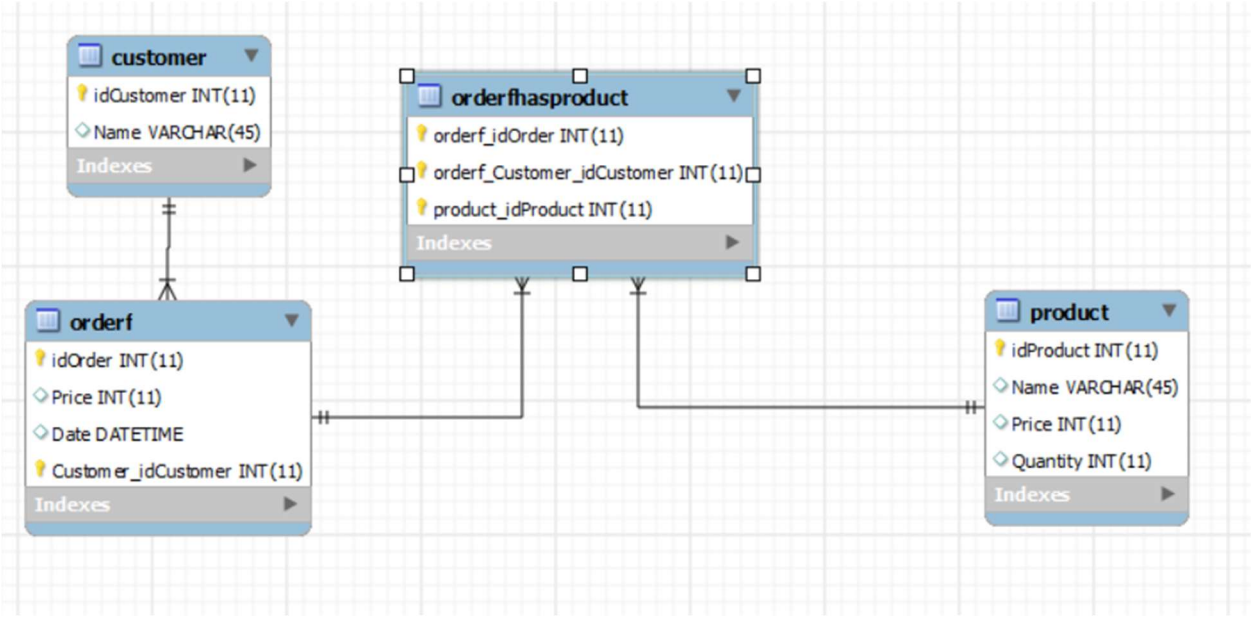
An important design pattern used in this application was singleton. This was used in order to ensure that there are no duplicate connection to the database and to avoid unnecessary multiple instances.


<<Java Class>>
Main
presentation
Main()
main(String[]):void
createReport(Order):void
<<Java Class>>
EditEntryWindow
presentation
WIDTH: int
HEIGHT: int
mainFrame: JFrame
labels: ArrayList<JLabel>
textFields: ArrayList<JTextField>
objectType: Class<?>
done: JButton
id: int
editedObject: Object
EditEntryWindow()
setDoneActionListener(ActionListener):void
init(Class<?>,Action):void
getObjectType():Class<?>
setObjectType(Class<?>):void
init(Object,Action):void
getAction():Action
setAction(Action):void
fill(Object):void
getEditedObject():Object
close():void
<<Java Class>>
BLOManager
BLL
BLOManager()
instance():BLOManager
getBLO(Class<?>):BLO
+instance
0..1
<<Java Class>>
BLO<T>
BLL
BLO()
applyAction(Action,T):boolean
insert(T):int
delete(T):boolean
selectAll():ArrayList<T>
update(T):boolean
findById(T)
exists(T):boolean
-action
0..1
<<Java Enumeration>>
Action
presentation
DELETE: Action
INSERT: Action
UPDATE: Action
Action()
<<Java Class>>
GUI
presentation
WIDTH: int
HEIGHT: int
DIV_RATIO: int
mainFrame: JFrame
buttonsPanel: JPanel
dataPanel: JPanel
scrollPane: JScrollPane
createOrderButton: JButton
tableSelector: JComboBox<Class<?>>
generateReports: JButton
createNew: JButton
updateSelected: JButton
selectedObject: Object
deleteSelected: JButton
endSelection: JButton
selectionEnded: boolean
GUI()
getSelectionEnded():boolean
setSelectioEnded(boolean):void
setSelectedClass(Class<?>):boolean
getSelectedObject():Object
setComboBoxItemListener(ItemListener):void
setDeleteSelectedListener(ActionListener):void
loadInTable(ArrayList<Object>):boolean
getSelectedClass():Class<?>
setCreateOrderActionListener(ActionListener):void
setGenerateReportsActionListener(ActionListener):void
setCreateNewActionListener(ActionListener):void
setUpdateSelectedActionListener(ActionListener):void
init():void
<<Java Class>>
CustomerBLO
BLL
CustomerBLO()
instance():CustomerBLO
-instance
0..1
<<Java Class>>
OrderfBLO
BLL
OrderfBLO()
instance():OrderfBLO
addProduct(Orderf,Product):boolean
-instance
0..1
<<Java Class>>
Customer
models
idCustomer: int
name: String
Customer()
Customer(int,String)
Customer(String)
toString():String
<<Java Class>>
OrderfHasProductBLO
BLL
OrderfHasProductBLO()
instance():OrderfHasProductBLO
-instance
0..1
<<Java Class>>
ProductBLO
BLL
ProductBLO()
instance():ProductBLO
insert(Product):int
update(Product):boolean
-instance
0..1
<<Java Class>>
OrderfHasProduct
models
idProduct: int
idOrder: int
idOrderfHasProduct: int
OrderfHasProduct()
toString():String
OrderfHasProduct(int,int,int)
OrderfHasProduct(int,int)
<<Java Class>>
DAO
dal
DAO()
instance():DAO
insert(Object):int
update(Object):boolean
delete(Object):boolean
selectAll(Class<?>):ArrayList<Object>
findBy(Class<?>,String,int):ArrayList<Object>
findBy(Class<?>,String,String):ArrayList<Object>
findBy(Class<?>,String,Date):ArrayList<Object>
findById(Object):Object
getProductsOfOrder(Orderf):ArrayList<Product>
-instance
0..1
~currentOrder
0..1
<<Java Class>>
Product
models
idProduct: int
price: int
quantity: int
name: String
toString():String
Product()
Product(int,int,String)
Product(int,int,int,String)
<<Java Class>>
Orderf
models
idOrderf: int
price: int
observations: String
idCustomer: int
Orderf()
Orderf(int,Date,String)
Orderf(int,int,Date,String)
toString():String
<<Java Class>>
ConnectionFactory
dal
DRIVER: String
DBURL: String
USER: String
PASS: String
ConnectionFactory()
createConnection():Connection
getConnection():Connection
close(Connection):void
close(Statement):void
close(ResultSet):void
-singleInstance
0..1

5. Implementation

5.1 Database structure

The database was designed in order to solve the many-to-many issue between product and orderf tables. The name Orderf was chosen instead of Order because order is a sql reserverd word.



5.2 Java models

In order tot represent the records from each table a set of models was necessary. Two constructors were necessary in order to create object with an id (when reading to the

database) or without (when writing to the database).

```java
public class Customer {
    public Customer() {
        super();
    }

    public int idCustomer;
    public String name;

    public Customer(int idCustomer, String name) {
        super();
        this.idCustomer = idCustomer;
        this.name = name;
    }

    public Customer(String name) {
        super();
        this.name = name;
    }

    @Override
    public String toString(){
        return ("Customer has id: " + idCustomer + " and name: " + name);
    }
}
```

5.3 Connection Factory

In order to connect to the database properly a dedicate object was created following the " Factory ". The connection factory uses some constant values describing the driver, server and port to connect and return the connection to higher levels.

```java
private Connection createConnection() {
    Connection connection = null;
    try {
        System.out.println("[ConnectionFactory] Connecting to the database");
        connection = DriverManager.getConnection(DBURL, USER, PASS);
    } catch (SQLException e) {
        System.out.println("[ConnectionFactory] An error occured while trying to connect to the database");
        e.printStackTrace();
    }
    return connection;
}

public static Connection getConnection() {
    return singleInstance.createConnection();
}
```

5.4 Data access object

In order to create a more abstract mechanism of connecting to the database the data access layer was introduced. To confer more generality of the code an reflective approach was considered. Thus, a single data access object was needed.

```
public int insert(Object o){
    int retVal = -1;
    java.sql.Connection dbConnection = ConnectionFactory.getConnection();
    java.sql.PreparedStatement insertStatement = null;
    String insertStatementString = "INSERT INTO " + o.getClass().getSimpleName().toLowerCase() + " (";
    for (Field field : o.getClass().getDeclaredFields()) {
        if (!field.getName().equals("id" + o.getClass().getSimpleName())){
            insertStatementString += field.getName() + ",";
        }
    }
    insertStatementString = insertStatementString.substring(0, insertStatementString.length() - 1);
    insertStatementString += ") VALUES (";
    for (Field field : o.getClass().getDeclaredFields()) {
        if (!field.getName().equals("id" + o.getClass().getSimpleName())){
            try {
                if (field.getType() == String.class){
                    insertStatementString += "'" + field.get(o) + "'" + ",";
                }
                else{//assume int
                    insertStatementString += field.get(o) + ",";
                }
            } catch (IllegalArgumentException | IllegalAccessException e) {
                e.printStackTrace();
                return (-1);
            }
        }
    }
    insertStatementString = insertStatementString.substring(0, insertStatementString.length() - 1);
    insertStatementString += ")";
    try {
        insertStatement = dbConnection.prepareStatement(insertStatementString, Statement.RETURN_GENERATED_KEYS);
    } catch (SQLException e1) {
        e1.printStackTrace();
        return (-1);
    }
```

All reflective methods create the corresponding statement string based on the class name (which corresponds with the table name from the database) and the class fields (also correspond with the tables).

5.5 Business logic layer

The business logic layer is used in order to control the data flow between the presentation layer and the data access layer. Given the fact that each model could have different rules for different operations each objects needs and dedicated BLO (Business Logic Object). But, in order to avoid code duplication an generic abstract class that would implement all operations

in a general manner was considered

```
public int insert(T t){
    if (t == null) return (-1);
    return (DAO.instance().insert(t));
}

public boolean delete(T t){
    if (t == null) return (false);
    return (DAO.instance().delete(t));
}

@SuppressWarnings("unchecked")
public ArrayList<T> selectAll(){
    return (ArrayList<T>) (DAO.instance().selectAll(((Class<?>) ((ParameterizedType) getClass()
}

public boolean update(T t){
    if (t == null) return (false);
    return (DAO.instance().update(t));
}

@SuppressWarnings("unchecked")
public T findById(T t){
    if (t == null) return (null);
    return ((T)DAO.instance().findById(t));
}

public boolean exists(T t){
    return (findById(t) != null);
}
```

Afterwards each dedicated BLO extends the generic BLO.

```
public class OrderfBLO extends BLO<Orderf>{
```

In order to have a more loosely coupled design the method apply operation was considered. It receives an object and an action to be performed with that object and it applies it.

```
public boolean applyAction(Action action, T t){
    switch (action){
    case INSERT : insert (t); return true;
    case UPDATE : update (t); return true;
    case DELETE : delete (t); return true;
    }
    return false;
}
```

Also, in order to allow further use of reflection in higher-level layers a Business Logic Layer Manager was considered. The main responsibility of this object is to return the appropriate

Business Logic Object for a given type.

```
@SuppressWarnings("unused")
public BLO getBLO(Class<?> type){
    if (type.toString().equals(Customer.class.toString())){
         return (CustomerBLO.instance());
    }
    else if (type.toString().equals(Orderf.class.toString())){
        return (OrderfBLO.instance());
    }
    else if (type.toString().equals(OrderfHasProduct.class.toString())){
        return (OrderfHasProductBLO.instance());
    }
    else if (type.toString().equals(Product.class.toString())){
        return (ProductBLO.instance());
    }
    else {
        return (null);
    }
}
```

Also, in the business logic object of the order a method was added, namely " addProduct "

```
public boolean addProduct(Orderf o, Product p){
    if ((o == null) || (p == null)) return false;

    //checkQuantity
    if (p.quantity == 0){
        System.out.println("[OrderfBLO] " + p.name + " is out of stock!");
        return (false);
    }
    //decrease quantity
    p.quantity--;
    ProductBLO.instance().update(p);
    //add to total price
    o.price += p.price;
    //update order
    update(o);
    //create relationship
    if (OrderfHasProductBLO.instance().insert(new OrderfHasProduct(p.idProduct, o.idOrderf)) != -1){
        return true;
    }
    else{
        System.out.println("[OrderfBLO] an error occured while inserting the relationship");
        return false;
    }
}
```

This method receives as parameters an order and a product and adds the product to that order checking and managing the stock of the product and adding the corresponding record in the OrderHasProduct table.

5.6 Action

An enum named Action is used in order to manage in a more regular manner the actios that can be performed by the user.

```
public enum Action {
DELETE,
INSERT,
UPDATE
}
```

5.7 GUI

The gui is used in order to display data and to allow the user to interact with the application. The main method is the one that receives an ArrayList of objects and reflexively displays its contents on a JTable. The user is able to select the table using a drop down – type control. Also, the table is able of returning the currently selected object.

```
String[] columnNames = new String[objects.get(0).getClass().getDeclaredFields().length];
String[][] rowsData = new String[objects.size()][objects.get(0).getClass().getDeclaredFields().length];
int i = 0;

for (Field field : objects.get(0).getClass().getDeclaredFields()) {
    columnNames[i++] = field.getName();
    System.out.print(field.getName() + "\t");
}
System.out.println();
i = 0;
for (Object o : objects){
    int j = 0;
    for (Field field : objects.get(0).getClass().getDeclaredFields()) {
        try {
            if (field.getType() == Date.class){
                System.out.println("[GUI] TODO convert date type");//TODO
            }
            else{
                rowsData[i][j++] = field.get(o).toString();
            }

            System.out.print(field.get(o) + "\t\t");
        } catch (IllegalArgumentException | IllegalAccessException e) {
            e.printStackTrace();
        }
    }
    i++;
    System.out.println();
}
final JTable table = new JTable(rowsData, columnNames);
table.setBounds(0, 0, dataPanel.getWidth(), dataPanel.getHeight());
table.setPreferredScrollableViewportSize(new Dimension(500, 70));
table.setFillsViewportHeight(true);
//TODO add selected action listener
table.getSelectionModel().addListSelectionListener(new ListSelectionListener(){
    @Override
    public void valueChanged(ListSelectionEvent e) {
        selectedObject = objects.get(table.getSelectionModel().getMaxSelectionIndex());
    }
```

5.8 EditEntryWindow

In order to edit or create an new entry in the table the EditEntryWindow is used.

Firstly the frame is populated with labels that display the fields names and text fields used by the user to insert data. This is done via reflection.

```
public void init(Class<?> objectType, Action action){
    this.objectType = objectType;
    this.action = action;
    mainFrame.getContentPane().removeAll();
    labels.clear();
    textFields.clear();
    mainFrame.setBounds(10, 10, WIDTH, HEIGHT);
    mainFrame.setTitle(action.toString() + " " + objectType.getSimpleName());
    mainFrame.setLayout(null);
    int i = 0;
    final int elemWidth = WIDTH / (objectType.getDeclaredFields().length - 1);
    for (Field field : objectType.getDeclaredFields()) {
        if (!field.getName().equals("id" + objectType.getSimpleName())){
            try {
                JLabel label = new JLabel(field.getName());
                label.setBounds(elemWidth * i, HEIGHT / 16, elemWidth, HEIGHT / 4);
                labels.add(label);
                JTextField textField = new JTextField();
                textField.setBounds(elemWidth * i, HEIGHT / 4, elemWidth, HEIGHT / 4);
                textFields.add(textField);
                i++;
            } catch (IllegalArgumentException e) {
                e.printStackTrace();
            }
        }
    }
```

An overload of the method init is used in order to also fill the text fields with the current values of the object if an update is performed.

```
public void init(Object object, Action action){
    init(object.getClass(), action);
    for (Field field : object.getClass().getDeclaredFields()) {
        if (field.getName().equals("id" + object.getClass().getSimpleName())){
            try {
                id = field.getInt(object);
            } catch (IllegalArgumentException | IllegalAccessException e) {
                e.printStackTrace();
            }
        }
    }
    fill(object);
    if (action == Action.DELETE){
        editedObject = object;
        done.doClick();
    }
}
```

From outside the EditEntryWindow class the action listener of the “ done ” button is set. Thus, when pressing the button the action is performed with the new / edited object, the table is reloaded (in order to display the changes) and the editEntryWindow is closed.

```
eew.setDoneActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent e) {
        try{
            BLOManager.instance().getBLO(gui.getSelectedClass()).applyAction(eew.getAction(), eew.getEditedObject());
            gui.loadInTable(BLOManager.instance().getBLO(gui.getSelectedClass()).selectAll());
        }
        catch (Exception e1){
            System.out.println("[Main] An error ocurred.");
            e1.printStackTrace();
        }
        eew.close();
    }

});
```

In order to fill the fields in case of an update being perfomerd the method " fill " is used. It performs this actions by means of reflection.

```
public void fill(Object object){
    int i = 0;
    for (Field field : object.getClass().getDeclaredFields()) {
        if (!field.getName().equals("id" + object.getClass().getSimpleName())){
            try {
                textFields.get(i).setText(field.get(object).toString());
                i++;
            } catch (IllegalArgumentException | IllegalAccessException e) {
                e.printStackTrace();
            }
        }
    }
}
```

The " getEditedObject " parses the texts from the textFields and creates an object of the required class (class is stored in the init method) and returns the object.

```
 public Object getEditedObject(){
    int i = 0;
    if (action == Action.DELETE){
        return editedObject;
    }
    try {
        Object o = Class.forName(objectType.getName()).getConstructor().newInstance();
        for (Field field : objectType.getDeclaredFields()) {
            if (!field.getName().equals("id" + o.getClass().getSimpleName())){
                if (field.getType() == String.class){
                    field.set(o, textFields.get(i).getText());
                }
                else{//assume int
                    field.set(o, Integer.parseInt(textFields.get(i).getText()));
                }
                i++;
            }
            else{
                field.set(o, id);
            }
        }
        return (o);
    } catch (InstantiationException | IllegalAccessException | IllegalArgumentException | InvocationTargetException
            | NoSuchMethodException | SecurityException | ClassNotFoundException  e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
        return (null);
    }

 }
```

5.9 MainController

The main controller binds the front-end of the application and the back-end. In order to achieve that the action listeners to all user-intractable controllers are set here.
Firstly, when the users click the button " generate reports " the sequence of code bellow is executed and for all current orders in the database a report is generated.

```
gui.setGenerateReportsActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent arg0) {
        ArrayList<Object> orders = DAO.instance().selectAll(Orderf.class);
        for (Object o : orders){
            createReport((Orderf)o);
        }
    }
});
```

A generic action listener is used for all modifier buttons (insert, update, delete). The edit entry window is initialized with the action of the button and the currently selected object.

```
ActionListener modifierAL = new ActionListener(){

    @Override
    public void actionPerformed(ActionEvent e) {
        JButton button = (JButton)e.getSource();
        Action action = Action.valueOf(button.getText());
        if (action == Action.INSERT){
            eew.init(gui.getSelectedClass(), Action.INSERT);
        }
        else{
            eew.init(gui.getSelectedObject(), action);
        }
    }};
```

In order to have a responsive application the following action listener was attached to the combo box. This being fired each time the user changes the selection in the combo-box.

```
gui.setComboBoxItemListener(new ItemListener(){
    @Override
    public void itemStateChanged(ItemEvent arg0) {
        gui.loadInTable(BLOManager.instance().getBLO(gui.getSelectedClass()).selectAll());
    }
});
```

6 Conclusions and Further Development

The development of this application was a very good exercise of OOP design but much more important it offered an insight of database related applications. Further development consists of more restrictive general user interface and more comprehensive(meaningful colors and current state display).