**Order Management Documentation**

1. Assignment objective

Consider an application **OrderManagement** for processing client orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application should be structured in packages using a layered architecture presented in the support presentation 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

It should also use reflection techniques for working with the data.

Secondary objectives:

* Create a bill for every order
* Generic classes for data access and business logic that use reflection

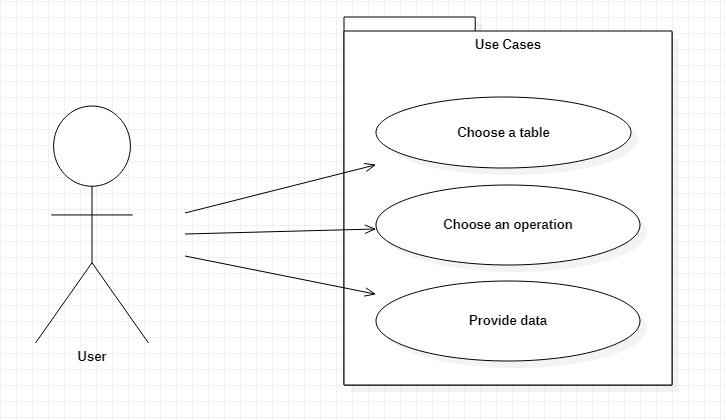
1. Problem analysis
2. Problem analysis

The connection to the database must be created with a factory class and closed after each operation. The operations will be insertion of a new row, the update of a row with different values and the deletion of a row. Queries must be created with input data and sent to the database to be executed.

1. Modelling of the problem

The user will be able to choose the table he wants to work with, then the operation he wants to execute, and then input data. In the same time, he will be able to see the database table, which updates in real-time. If he inserts data, he will have to provide the id and name, and then possibly other data for the rest of the fields. For editing an existing entry, he will have to provide the id of the row to be modified and then the values he wants to change. For deleting, he will only have to specify the id. If an order is inserted, a bill will be created. Any changes made will be saved to the database.

1. Use cases & scenarios



The actor is the user that chooses the table, operation, inputs data for that operation and executes.

Choose a table – Success: One of the 3 available tables has been chosen.

-Failure: Failure is impossible, even for monkey users, I’ll make sure of that.

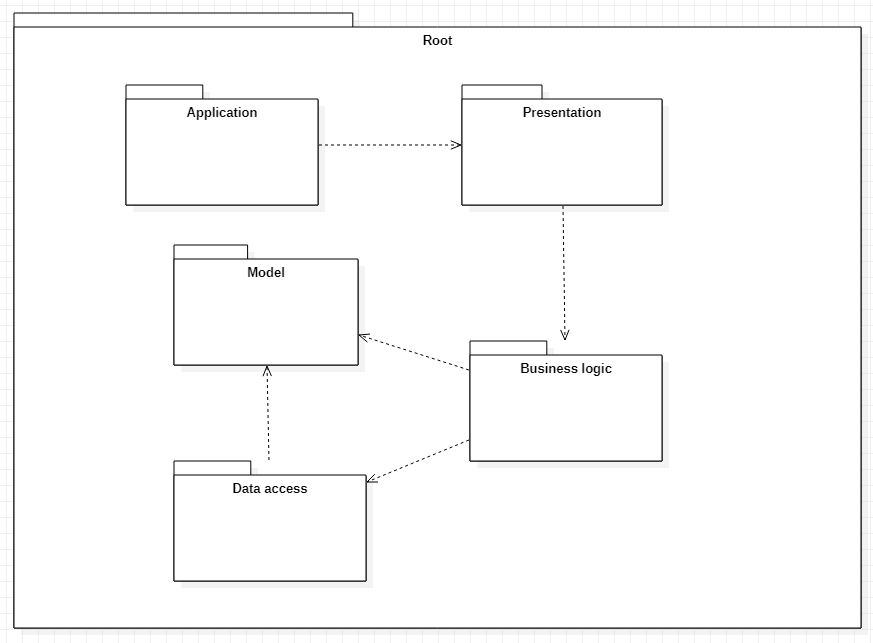
Choose an operation – Success: One of the 3 available operations has been chosen.

-Failure: Once again, impossible, a default operation will be chosen at the start.

Provide data – Success: The input data is valid for each field

-Failure: If the data is invalid (new id/name already exists, target id for update/delete does not exist, the type of a field is wrong, like string instead of integer, and other no common-sense mistakes)

1. Design
2. Packages

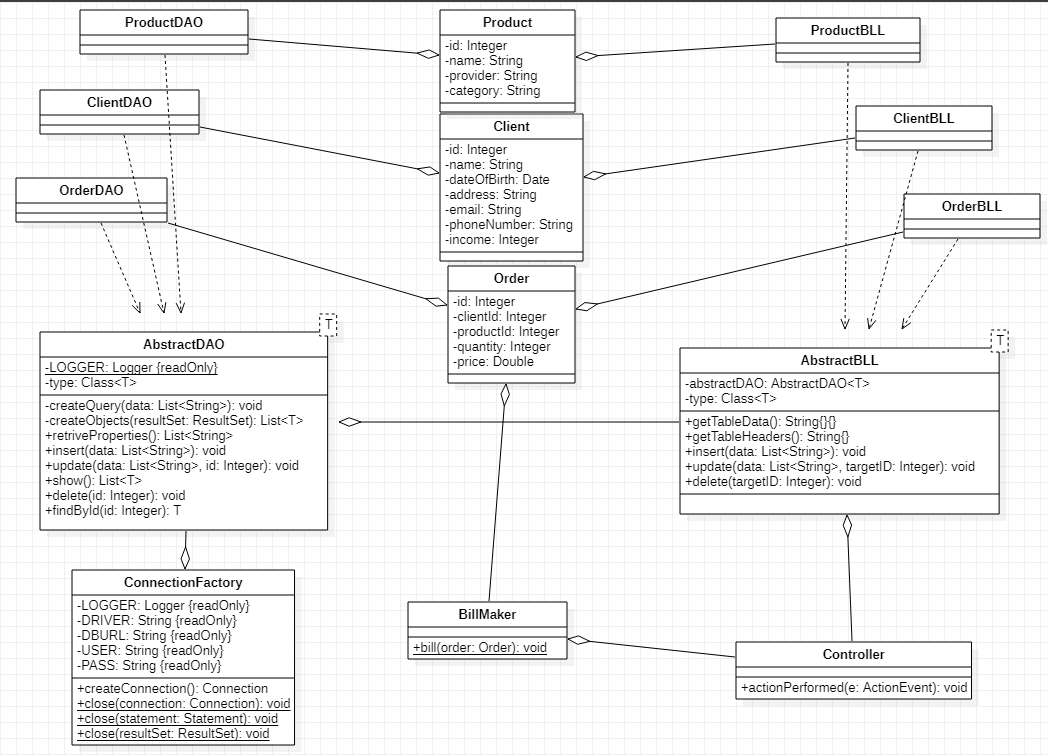


As we can see, the project clearly respects the MVC design.

* Application: holds the main function and is used to start up the GUI
* Presentation: it holds the interface components and the controllers
* Business logic: contains the classes that work with the data and the data access functions
* Data access: holds the classes that work with the database
* Model: contains the classes that specialize in holding data
* Root: the application package that holds all the other packages, basically the project directory

1. Classes

Here is the UML diagram of the classes:



1. Data structures

There should be data structures for the clients, products and orders.

* Client: has an Integer id, String name, Date dateOfBirth, String address, String email, String phoneNumber and Integer income
* Product: has an Integer id, String name, String provider and String category
* Order: has an Integer id, Integer clientId, Integer productId, Integer quantity and a Double price

1. Interfaces

I don’t think that I will use interfaces, but the AbstractDAO and AbstractBLL will work somewhat like abstract classes (even though they are normal classes), with a version for each model class that fills in the generics.

1. Algorithms

We will use reflection techniques in the “abstract” classes in order to extract the fields of the model class and use it to build the queries for the database. We might need the values of the fields, but similar reflective algorithms will be used for that. Recreating an object out of the fields returned from a query might seem a bit difficult since you can’t call the constructor of a generic type in the normal way, however there are workarounds for that, as long as you know what type the generic can be.

1. Design decisions

There will be buttons at the top to switch between the panels. Each panel will correspond to a table in the database. There will be a table in the interface that mimics the one in the database. Three radio buttons will be available for choosing the operation and a button for executing it. Finally, there will be a number of labels and textboxes corresponding to each field of the model class of that table.

1. Implementation
2. Classes and methods

Here are the classes and their methods, divided by packages. Of course, all of them have their needed getters, setters and constructors that I did not include here.

* BusinessLogic:

*/\*\*  
 \* Reflective class for business logic functions on database  
 \*  
 \** ***@param*** <*T*> *The model class  
 \*/*public class AbstractBLL<T>

private AbstractDAO<T> abstractDAO;  
  
private final Class<T> type;

@SuppressWarnings("unchecked")  
public AbstractBLL(AbstractDAO<T> abstractDAO) {  
 this.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];  
 this.abstractDAO = abstractDAO;  
}

*/\*\*  
 \* Gets all the table's fields and returns it as a matrix for the swing table  
 \*  
 \** ***@return*** *The table data as a string matrix  
 \*/*public String[][] getTableData()

*/\*\*  
 \* Uses reflection to get the column names from the table  
 \*  
 \** ***@return*** *The array of headers  
 \*/*public String[] getTableHeaders()

*/\*\*  
 \* Inserts a new element in the database  
 \*  
 \** ***@param*** *data the values of the new row  
 \** ***@throws*** *SQLException Exception from sql  
 \** ***@throws*** *IllegalAccessException Exception from accessing fields  
 \*/*public void insert(ArrayList<String> data) throws SQLException, IllegalAccessException

*/\*\*  
 \* Updates an existing element from the database with new values  
 \*  
 \** ***@param*** *data the new values of the row  
 \** ***@param*** *targetID the id of the row to be modified  
 \** ***@throws*** *SQLException Exception from sql  
 \** ***@throws*** *IllegalAccessException Exception from accessing fields  
 \** ***@throws*** *ObjectNotFoundException There is no row with that id in the table  
 \*/*public void update(ArrayList<String> data, Integer targetID) throws SQLException, IllegalAccessException,  
 ObjectNotFoundException

*/\*\*  
 \* Deletes an element from the database  
 \*  
 \** ***@param*** *targetID the id of the row to be deleted  
 \** ***@throws*** *SQLException Exception from sql  
 \** ***@throws*** *ObjectNotFoundException There is no row with that id in the table  
 \*/*public void delete(Integer targetID) throws SQLException, ObjectNotFoundException

*/\*\*  
 \* AbstractBLL version for clients  
 \*/*public class ClientBLL extends AbstractBLL<Client>{  
 public ClientBLL(ClientDAO abstractDAO) {  
 super(abstractDAO);  
 }  
}

*/\*\*  
 \* AbstractBLL version for orders  
 \*/*public class OrderBLL extends AbstractBLL<Order> {  
 public OrderBLL(OrderDAO abstractDAO) {  
 super(abstractDAO);  
 }  
}

*/\*\*  
 \* AbstractBLL version for products  
 \*/*public class ProductBLL extends AbstractBLL<Product>{  
 public ProductBLL(ProductDAO abstractDAO) {  
 super(abstractDAO);  
 }  
}

*/\*\*  
 \* Class for creating the bills  
 \*/*public class BillMaker

*/\*\*  
 \* Creates a bill for a given order  
 \*  
 \** ***@param*** *order The order  
 \*/*public static void bill(Order order)

* Dao

*/\*\*  
 \* Class for managing database connection.  
 \*/*public class ConnectionFactory  
 private static final Logger *LOGGER* = Logger.*getLogger*(ConnectionFactory.class.getName());  
 public static final String *DRIVER* = "com.mysql.cj.jdbc.Driver";  
 public static final String *DBURL* = "jdbc:mysql://localhost:3306/ptdb";  
 public static final String *USER* = "root";  
 public static final String *PASS* = "";

private static final ConnectionFactory *singleInstance* = new ConnectionFactory();  
  
 private ConnectionFactory() {  
 try {  
 Class.*forName*(*DRIVER*);  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 }

*/\*\*  
 \* Creates a connection, using the url, username and password of the class.  
 \*  
 \** ***@return*** *The connection  
 \*/*public Connection createConnection()

*/\*\*  
 \* Closes a connection.  
 \*  
 \** ***@param*** *connection The connection to be closed  
 \*/*public static void close(Connection connection)

*/\*\*  
 \* Closes a statement.  
 \*  
 \** ***@param*** *statement The statement to be closed  
 \*/*public static void close(Statement statement)

*/\*\*  
 \* Closes a resultSet.  
 \*  
 \** ***@param*** *resultSet The resultSet to be closed  
 \*/*public static void close(ResultSet resultSet)

*/\*\*  
 \* Reflective class for working with the database  
 \*  
 \** ***@param*** <*T*> *The model class  
 \*/*

public class AbstractDAO<T> {  
 protected static final Logger *LOGGER* = Logger.*getLogger*(AbstractDAO.class.getName());  
  
 private final Class<T> type;  
  
 @SuppressWarnings("unchecked")  
 public AbstractDAO() {  
 this.type = (Class<T>) ((ParameterizedType) getClass().getGenericSuperclass()).getActualTypeArguments()[0];  
 }

*/\*\*  
 \* Creates a query that gets all rows  
 \*  
 \** ***@return*** *The query  
 \*/*private String createShowQuery()

*/\*\*  
 \* Creates a query that gets a specific row  
 \*  
 \** ***@return*** *The query  
 \*/*private String createSelectQuery(String field)

*/\*\*  
 \* Creates a query that deletes a row  
 \*  
 \** ***@return*** *The query  
 \*/*private String createDeleteQuery(String field)

*/\*\*  
 \* Creates a query that inserts a new row  
 \*  
 \** ***@return*** *The query  
 \*/*private String createInsertQuery(ArrayList<String> data) throws IllegalAccessException

*/\*\*  
 \* Creates a query that updates a row  
 \*  
 \** ***@return*** *The query  
 \*/*private String createUpdateQuery(ArrayList<String> data, String dbField, Integer targetID)  
 throws IllegalAccessException

*/\*\*  
 \* Creates a list of objects from a resultSet  
 \*  
 \** ***@param*** *resultSet The results of a select query  
 \** ***@return*** *The list of results in T form  
 \*/*private List<T> createObjects(ResultSet resultSet)

*/\*\*  
 \* Gets the names of the attributes of T  
 \*  
 \** ***@return*** *A list of the names of the attributes  
 \*/*public final List<String> retrieveProperties()

*/\*\*  
 \* Inserts a new row in the database  
 \*  
 \** ***@param*** *data values of the new row  
 \** ***@throws*** *SQLException Exception from sql  
 \** ***@throws*** *IllegalAccessException Exception from accessing fields  
 \*/*public void insert(ArrayList<String> data) throws SQLException, IllegalAccessException

*/\*\*  
 \* Updates an existing element from the database with new values  
 \*  
 \** ***@param*** *data the new values of the row  
 \** ***@param*** *id the id of the row to be modified  
 \** ***@throws*** *SQLException Exception from sql  
 \** ***@throws*** *IllegalAccessException Exception from accessing fields  
 \*/*public void update(ArrayList<String> data, Integer id) throws SQLException, IllegalAccessException

*/\*\*  
 \* Gets all rows of the table from the database  
 \*  
 \** ***@return*** *A list with the rows as T objects  
 \*/*public List<T> show()

*/\*\*  
 \* Deletes an element from the database  
 \*  
 \** ***@param*** *id the id of the row to be deleted  
 \** ***@throws*** *SQLException Exception from sql  
 \*/*public void delete(Integer id) throws SQLException

*/\*\*  
 \* Searches for a row with matching id  
 \*  
 \** ***@param*** *id The searched id  
 \** ***@return*** *The row as a T object  
 \*/*public T findById(Integer id)

*/\*\*  
 \* AbstractDAO version for clients  
 \*/*public class ClientDAO extends AbstractDAO<Client>{  
}

*/\*\*  
 \* AbstractDAO version for orders  
 \*/*public class OrderDAO extends AbstractDAO<Order>{  
}

*/\*\*  
 \* AbstractDAO version for products  
 \*/*public class ProductDAO extends AbstractDAO<Product>{  
}

* Model

*/\*\*  
 \* Enumeration for the actions  
 \*/*public enum OperationsConstants {  
 *ADD*,  
 *EDIT*,  
 *DELETE*}

*/\*\*  
 \* Exception for not finding a searched row in the table  
 \*/*public class ObjectNotFoundException extends Exception{  
}

*/\*\*  
 \* Model class for client  
 \*/*public class Client  
 private Integer id;  
 private String name;  
 private Date dateOfBirth;  
 private String address;  
 private String email;  
 private String phoneNumber;  
 private Integer income;

*/\*\*  
 \* Model class for product  
 \*/*public class Product  
 private Integer id;  
 private String name;  
 private String provider;  
 private String category;

*/\*\*  
 \* Model class for order  
 \*/*public class Order  
 private Integer id;  
 private Integer clientId;  
 private Integer productId;  
 private Integer quantity;  
 private Double price;

* Presentation

*/\*\*  
 \* Controller for pressing the button and executing a command  
 \*  
 \** ***@param*** <*T*> *The model class  
 \** ***@param*** <*BLL*> *The BLL class  
 \** ***@param*** <*DAO*> *The DAO class  
 \*/*public class ControllerAction<T, BLL extends AbstractBLL<T>, DAO extends AbstractDAO<T>> implements ActionListener {  
 private ControllerRadioButtons controllerRadioButtons;  
 private ArrayList<JTextField> textFields;  
 private JTextField textFieldTargetId;  
 private BLL abstractBLL;  
 private JTable table;

*/\*\*  
 \* Gets the JTable and creates it first if it is null  
 \*  
 \** ***@return*** *The JTable  
 \*/*public JTable getTable()

*/\*\*  
 \* Executes the command chosen by the radio buttons with the BLL methods  
 \*  
 \** ***@param*** *e ActionEvent  
 \*/*@Override  
public void actionPerformed(ActionEvent e)

*/\*\*  
 \* Creates an array with data from the textboxes, representing the values of the row  
 \*  
 \** ***@return*** *An array of strings from the textboxes  
 \*/*private ArrayList<String> createDataArray()

*/\*\*  
 \* Updates the JTable  
 \*/*private void updateTable()

*/\*\*  
 \* Controller for the radio buttons  
 \*/*public class ControllerRadioButtons implements ActionListener  
 private ArrayList<JRadioButton> radioButtons;  
 private ArrayList<JLabel> labels;  
 private ArrayList<JTextField> textFields;  
 private OperationsConstants opConst;  
 private JLabel labelTargetId;  
 private JTextField textFieldTargetId;

*/\*\*  
 \* Sets the opConst accordingly and changes the visibility of the fields for easier interface usage  
 \*  
 \** ***@param*** *e ActionEvent  
 \*/*@Override  
public void actionPerformed(ActionEvent e)

*/\*\*  
 \* A reflective JPanel  
 \*  
 \** ***@param*** <*T*> *The model class  
 \** ***@param*** <*BLL*> *The BLL class  
 \** ***@param*** <*DAO*> *The DAO class  
 \*/*public class AbstractPanel<T, BLL extends AbstractBLL<T>, DAO extends AbstractDAO<T>> extends JPanel {  
 private final Class<T> typeData;  
 private final Class<BLL> typeBLL;  
 private final Class<DAO> typeDAO;  
 private final ControllerAction<T, BLL, DAO> controllerAction;  
 private final ControllerRadioButtons controllerRadioButtons;

*/\*\*  
 \* The panel with the radio buttons and the execute button  
 \*/*private class ActionPanel extends JPanel

*/\*\*  
 \* The panel with the table  
 \*/*private class TablePanel extends JPanel

*/\*\*  
 \* The reflective panel for the textboxes  
 \*/*private class TextFieldsPanel extends JPanel

}

*/\*\*  
 \* AbstractPanel version for clients  
 \*/*public class ClientPanel extends AbstractPanel<Client, ClientBLL, ClientDAO>{  
}

*/\*\*  
 \* AbstractPanel version for orders  
 \*/*public class OrderPanel extends AbstractPanel<Order, OrderBLL, OrderDAO> {  
}

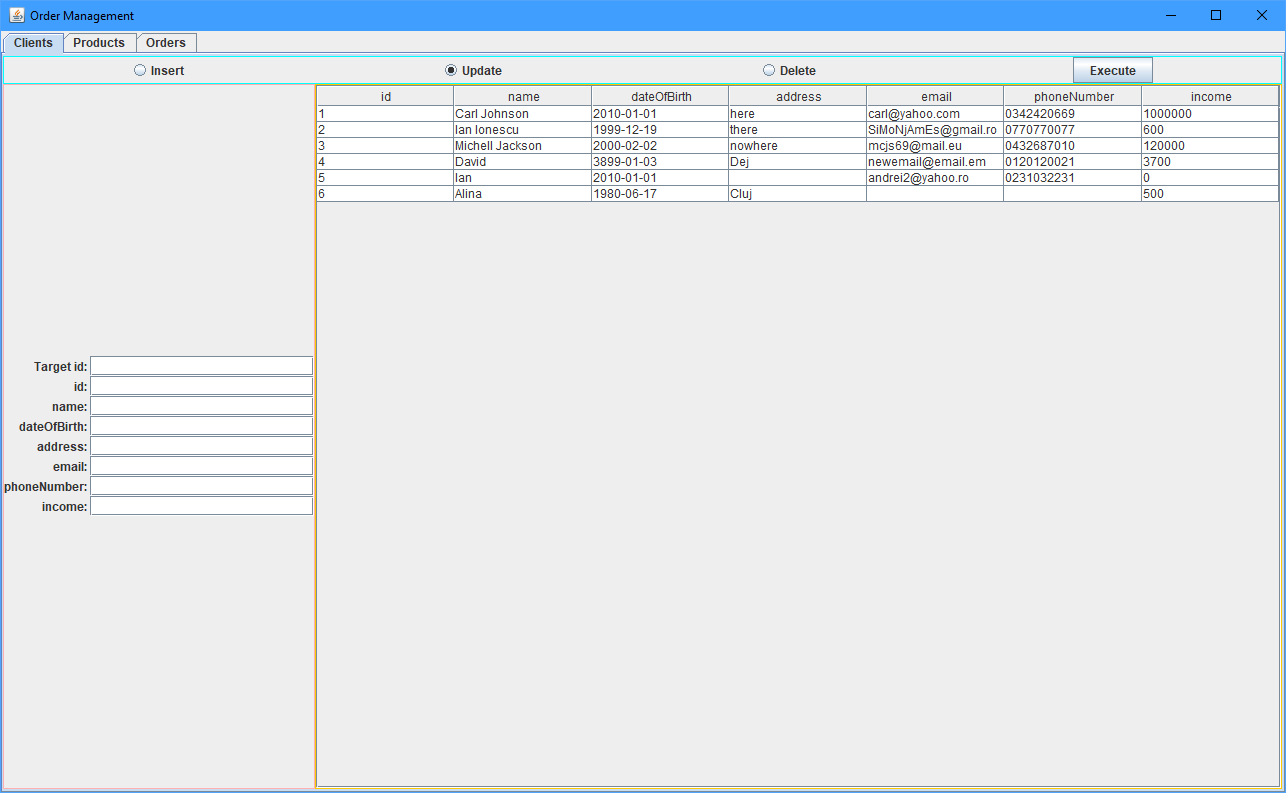
*/\*\*  
 \* AbstractPanel version for products  
 \*/*public class ProductPanel extends AbstractPanel<Product, ProductBLL, ProductDAO> {  
}

*/\*\*  
 \* Main frame.  
 \*  
 \* Completes some initializations for the GUI.  
 \* Creates the panels where all the interactions will take place.  
 \*/*public class MainFrame extends JFrame {  
  
  
 public MainFrame(String title) {  
 super(title);  
  
 // Set layout manager  
 setLayout(new BorderLayout());  
  
 //Add Swing components to content pane  
 Container container = getContentPane();  
  
 JTabbedPane tabbedPane = new JTabbedPane();  
  
 tabbedPane.addTab("Clients", null, new ClientPanel(), "For client management");  
 tabbedPane.addTab("Products", null, new ProductPanel(), "For product management");  
 tabbedPane.addTab("Orders", null, new OrderPanel(), "For order management");  
  
 container.add(tabbedPane, BorderLayout.*CENTER*);  
 }  
}

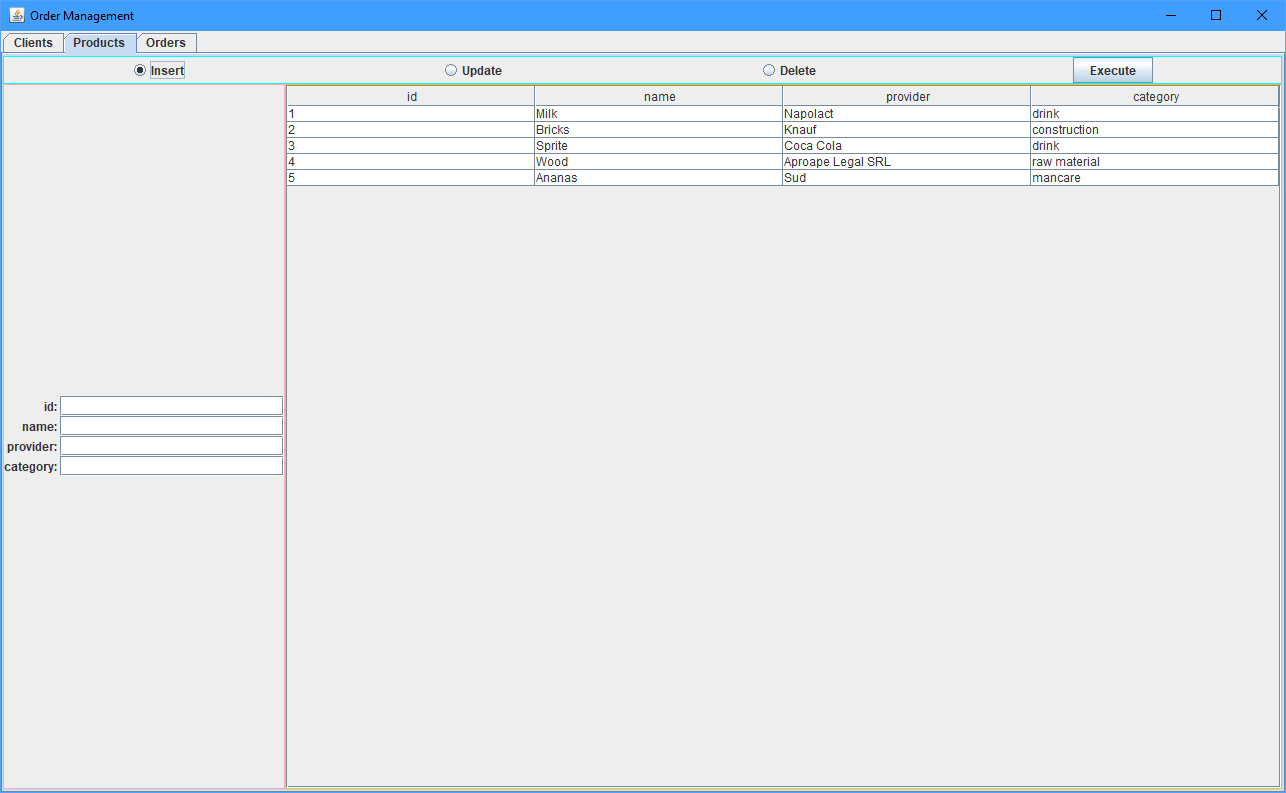
* Start

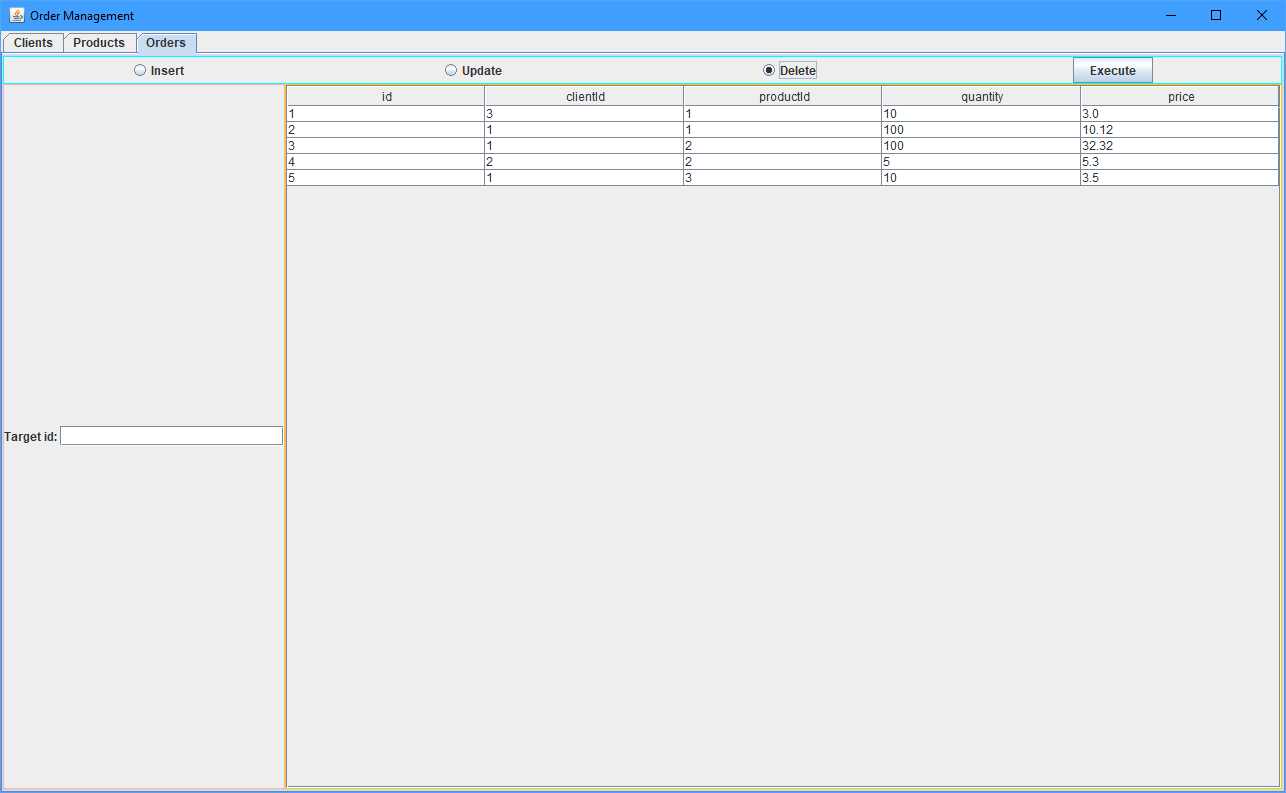
public class Main {  
  
 public static void main(String[] args) {  
  
 SwingUtilities.*invokeLater*(new Runnable() {  
 */\*\*  
 \* Starts a thread for the Swing graphical interface and completes the setup for the frame.  
 \*/* @Override  
 public void run() {  
 JFrame frame = new MainFrame("Order Management");  
 frame.setSize(1300, 800);  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 frame.setVisible(true);  
 }  
 });  
 }  
}

1. User interface



Up-left we have the tabs for changing the tables. Bellow that we have the operation selection and the execute button. In the center we have the table that mimics the database. Lastly, on the left we have the textboxes for the operations. Their number varies depending on the number of columns in the table, and the operation selected. If it is insert, the number of textboxes will be the same as the columns, if it is update, it will have one extra at the top representing the id of the row you want to change and if it is delete, only one textbox will be visible, that being the one for the target id.





1. Results

Following multiple tests and a lot of time spent debugging, I managed to get it to run correctly in all tested cases. There are various error messages for when the user tries to use the interface in a way that it is not meant to be used.

Most errors are handled by the MySql database. Combining a few strings from the textboxes into a query for sql is pretty simple, and any type mismatch, incorrect format, null field that shouldn’t be or duplicate unique field will be caught by MySql and an appropriate error message will be sent back, which is then displayed to the user as a dialog box. The number of possible errors is not very high, therefore there shouldn’t be any problems.

The connection with the database and the execution of the queries works as intended, the results can be seen in real time, either in the interface table, or in the database table (if you want to be sure). If a problem with the connection appears, then an error will be thrown, however there is not much else that can be done because such an exception can only be caused by something outside the code and therefore outside of hte program’s control.

1. Conclusion

This was an interesting project that helped me learn how to connect a java program to a sql database, which will undoubtedly be very useful in my career. The reflection techniques and generic abstracts are also very interesting as I was able to make a GUI panel that autogenerates the correct number of textboxes for each field of a data type. In order to create a new one for a different model class you only need to extend it with a new class and specify it’s generics. If I had to make, let’s say, 20 panels for 20 tables in a database, it would be as easy as using a for each loop to read the elements of a list (comparison of difficulty, not actual implementation). It seems a bit convoluted at first and I couldn’t rewrite some methods off the top of my head, but it is nonetheless very useful and important to know the idea.

This could be improved by expanding it for more tables, or maybe even making it fully reflective by not needing the model classes. It could also add more operations like search by a field, not necessarily just the id, like a filter. It could also be used to create some kind of reports, like how many orders over a given price have been ordered and other useful information that can be extracted with a simple (or complex) query.

1. Bibliography

<https://docs.oracle.com/en/>

<https://stackoverflow.com/>