RUSU CARLA MARIA
GROUP 30421

Order management

RUSU CARLA MARIA
GROUP 30421

Objective

"Consider an application OrderManagement for processing customer orders for a warehouse. Relational databases are used to store the products, the clients and the orders. Furthermore, the application uses (minimally) the following classes:

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

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

OBJECTIVE	ITEM	DETAILS
MAIN OBJECTIVE	Order Management	The application should simulate the order management of a warehouse; The user should be able to create, find, update, view or delete products, clients, orders and the countries of destination. The ability to place an order and consequently obtain a bill is also required.
SECONDARY OBJECTIVES	Models	Using the MVC architecture, a model should be provided representing the tables of the database.
	Data access layer	A data access layer is necessary to communicate between the internal structure of the application and the database.
	Business layer	The business layer is the link between the GUI and the internal structure (data access layer). It contains the logic of the application by allowing certain CRUD events to make changes to the database.
	Presentation classes	These contain the GUI: namely the view and the controller. They are the direct link between the user and the business layer.
	Database connection	A database connection is necessary to read and change the entries of the warehouse's database. MySQL server and database were used.
	Utility programs/classes	Utility programs and classes were used to employ reflection techniques.

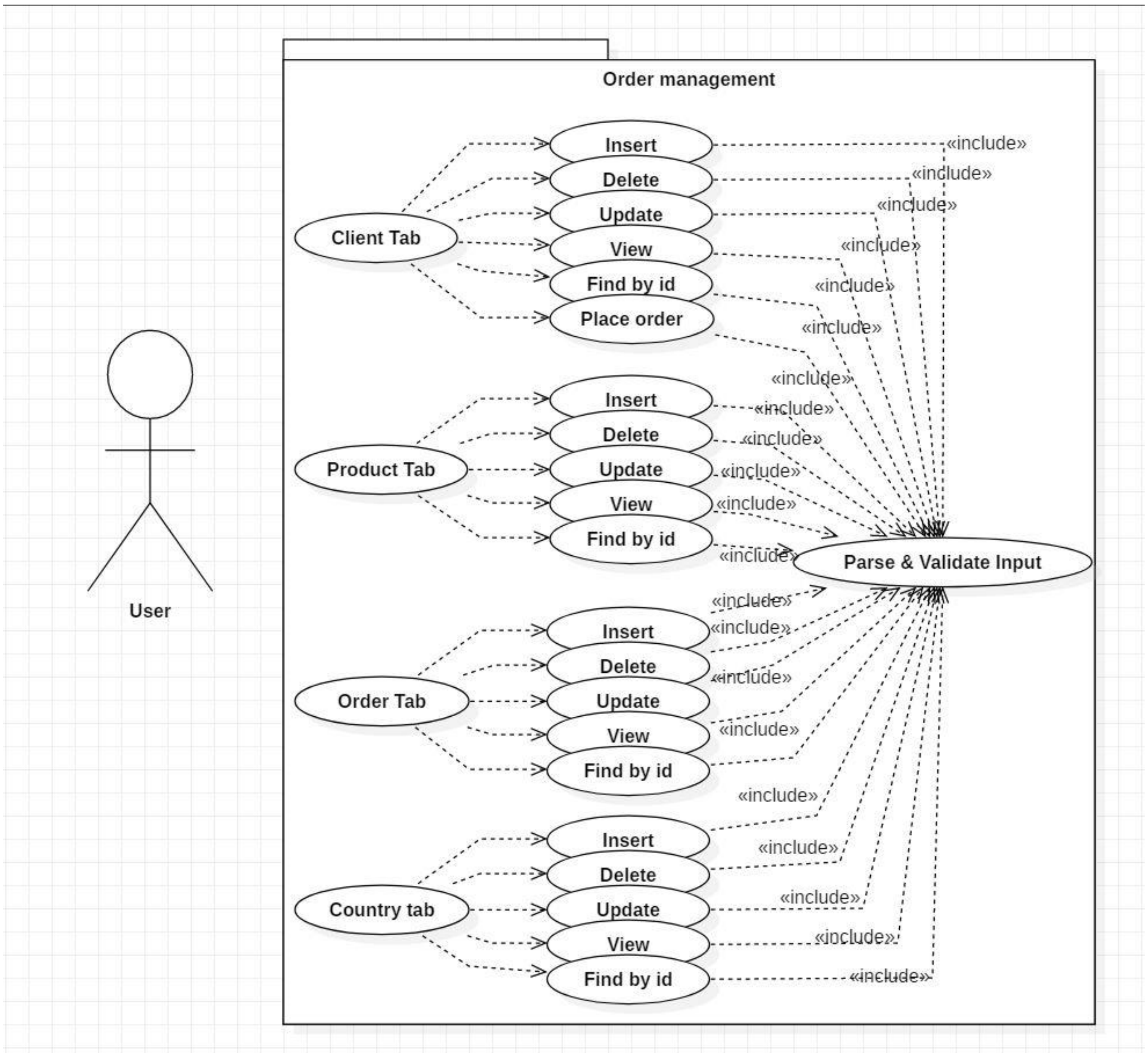
Problem analysis

- Choose the desired 'tab' or table to interact with (Clients, Products, Orders, Countries);
- Input the necessary data; depending on the tab, the fields will differ
    a) Identification numbers (PK)
    b) Other ids' (FKs)
    c) Other fields such as: name, address, age, email, product_name, stock, etc.
- The 'Delete', ' Place order' and 'Find by id' buttons need only valid ids to be inputted; 'Delete' erases an entry from the database if it exists; 'Place order' generates a bill for the given id in the application's folder; 'Find' generates an option pane with the information for the requested entry;
- The 'Insert' button does not need an id as it is automatically generated in the database; it inserts an entry with the given fields if the data is valid;
- The 'View' button does not need any input; it creates and populates a table on the right side of the GUI with the table's entries
- The 'Update' button needs the id; all other fields are the information with which the requested record will be updated, if it exists;
- The processing of the inputs is responsible for validating, parsing and converting the Strings into specific types that are stored in the corresponding attributes of the model. This is done by the application, with no action required from the user.

  With the use of regular expressions and String comparison, the Strings are validated;

- The results can be viewed either by pressing 'View' or by refreshing the database in the MySQL environment; the bill can be viewed as a .txt file

RUSU CARLA MARIA
GROUP 30421



- Main success scenario:
    1. User selects the desired tab, introduces the input parameters and clicks one of the buttons
    2. The application validates, parses and converts the input
    3. The result is either displayed as a .txt bill (place order) or can be viewed by refreshing the table (click 'View'); the find by id button offers the result in a dialog pane
- Alternative (error) sequences:

    a) Blank input (not applicable for 'View' button)

    1. User leaves some inputs blank (empty) and selects a button
    2. The app tries to validate the input and fails
    3. An error message dialog box is issued stating that the fields are empty;
    4. The scenario returns to first step

b) Non-existent entries

1. User introduces non-existent data and selects insert on the order tab

2. The app tries to validate the input and fails

3. An error message dialog box is issued stating that the input only accepts existing clients and products;

4.The scenario returns to step 1

c) Incorrect email field

1. User introduces data and selects insert on client tab

2. The app tries to validate the input and fails

3. An error message dialog box is issued stating that the input is not a valid email;

4.The scenario returns to step 1

RUSU CARLA MARIA
GROUP 30421

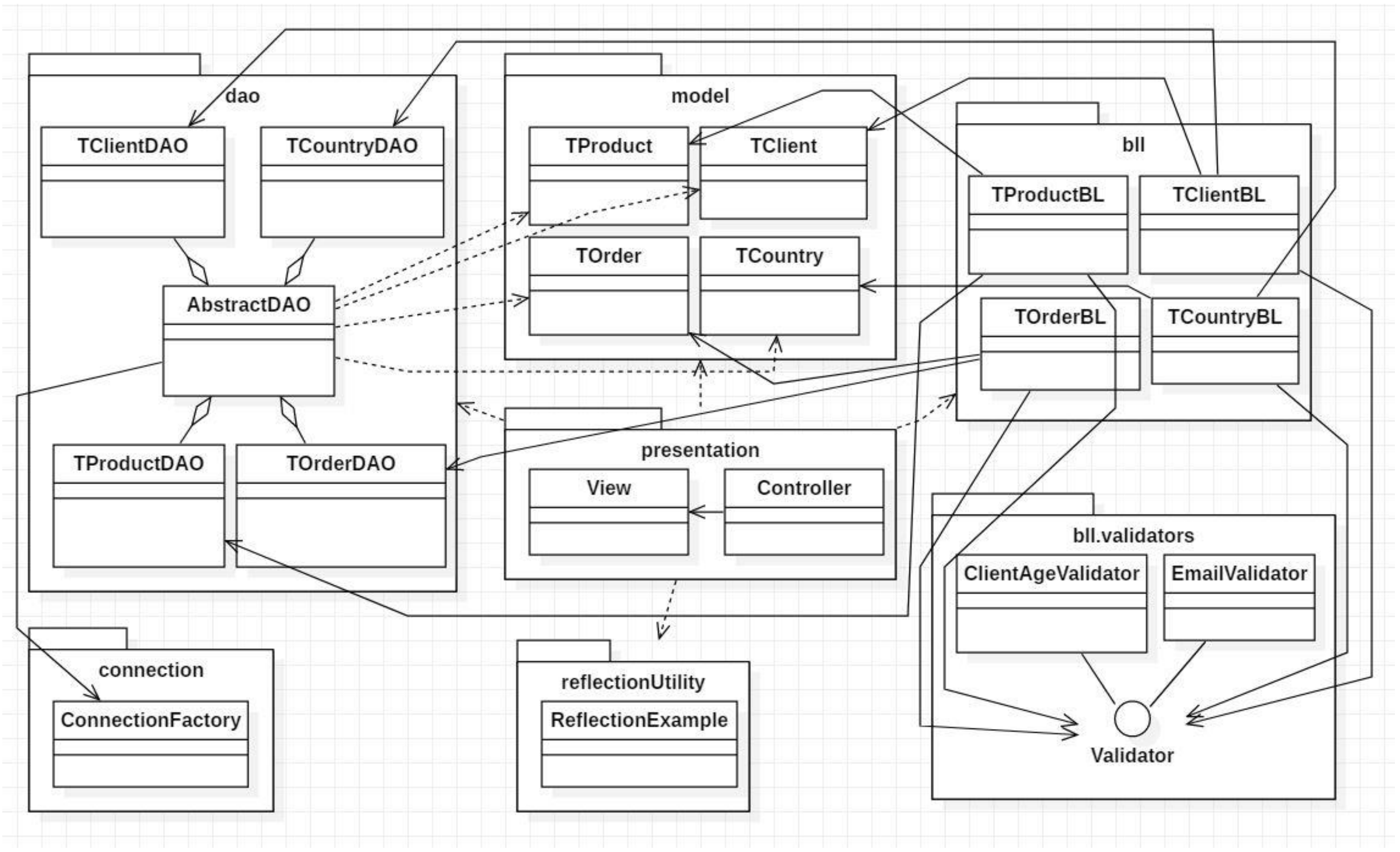
Design

The application is made up of seven packages and is structured on the MVC architecture: model (contains the client, country, order, product models for the database tables), presentation (contains the view class with the graphical elements and the controller which maps the action to the button and transfers the input into the system), bll (the business layer which links the GUI to the data access layer), bll.validators (utility validator classes), connection (makes the connection to the database), dao (the data access layer which links the business layer with the internal models), reflectionUtility(other utility classes used for reflection techniques). The project thus uses both an MVC architecture and a layered one (model, data access, business logic and presentation);

- ***Model package***
  Contains classes that map data similarly to the database; the column names match the attributes;
- ***DAO package***
  Contains the access layer classes; Uses reflection through the parameterized class AbstractDAO; Facilitates communication between models and business layer; exchanges information with the database
- ***BLL package***
  Has the business layer classes; represents the logic of the application; links the GUI with the data access layer; validates the input;
- ***Presentation package***
  Contains the GUI of the application: the view and the controller of the MVC; is the link between the user and the business layer;
- ***Connection package***
  Represents the connection between the application and the database; Assures the server is up and running;
- ***Other packages***
  Contain utility programs for validating the inputs or for employing reflection techniques when checking/ updating/ retrieving information;

The UML diagram displayed below offers a complete understanding of the relationship between the classes. The attributes and the methods were not included so as to provide a less cluttered, easier to understand diagram;

RUSU CARLA MARIA
GROUP 30421



Implementation

A detailed explanation of the implemented classes and algorithms is given below. As a disclaimer, the code was inspired by the examples provided by UTCN-DSRL; the projects given there were used as a skeleton on which the application was built;

- ***TClient, TOrder, TProduct, TCountry classes***
  They have the same attributes as their corresponding tables in the database; all fields have getters and setters; method toString() is overridden; a number of constructors are provided for the different operations applied;
- ***AbstractDAO<T> class***
  The class is parameterized in order to realize reflection, such that it contains all the operations (CRUD) for any generic class that is used as parameter; this makes the code reusable, easier to adapt and minimises the number of repeated lines of code greatly; a logger is used to record warnings and exceptions;

  Classes ***TClientDAO, TProductDAO, TCountryDAO and TOrderDAO*** will not be explained as they are just children of ***AbstractDAO<T>*** in which the **T** is replaced by the respective models

  a) ***Constructor:*** gets the class of the parameter in order to use reflection (Method, Class, Field, etc. are used)
  b) ***createSelectQuery():*** creates a SQL query as a String; it has the form "SELECT * FROM *class_name* WHERE *field* = ?" where class_name is the same as the table title and field is the primary key's name (eg.: id_tclient)
  c) ***createDeleteQuery():*** creates a SQL query as a String; it has the form "DELETE FROM *class_name* WHERE *field* = ?" where class_name is the same as the table title and field is the primary key's name (eg.: id_tclient)
  d) ***createInsertQuery():*** creates a SQL query as a String; it has the form "INSERT INTO *class_name* (*atrribute1, attribute2, … ,attributeN*) VALUES (?,?,…,?)" where class_name is the same as the table title and attributeN is the columns' name (eg.: id_tclient, name, age, etc.); the '?' will be replace by values in the methods that use these queries by using a prepared statement;
  e) ***createUpdateQuery():*** creates a SQL query as a String; it has the form "UPDATE *class_name* SET *attribute1* = ?,…, *attributeN* = ? WHERE field = ? " where class_name is the same as the table title, field is the primary key's name (eg.: id_tclient) and attributeN is the columns' name (eg.: id_tclient, name, age, etc.); the '?' will be replace by values in the methods that use these queries by using a prepared statement;
  f) ***findAll():*** connects to the database through ConnectionFactory class; initialises a prepared statement with the string "SELECT * FROM *class_name*", initialises a result set for the query, executes said query and stores the result in the result set; makes use of createObjects() to generate a List<> based on the result set; closes the result set, the statement and the connection;
  g) ***findById():*** connects to the database through ConnectionFactory class; initialises a prepared statement with the method createSelectQuery,

initialises a result set for the query, executes said query and stores the result in the result set; makes use of createObjects() to generate a List<> based on the result set and takes only the first element (and the only one since ids are unique); closes the result set, the statement and the connection;

h) ***createObjects():*** receives a result set as parameter and returns a List<> of those fields in the result set;

through the use of reflection; it begins by iterating the result set, creating an instance of type T (the type given as parameter) and parses the object's fields; for each field, it stores the corresponding value of the result set, initialises a property descriptor (which is used to find certain methods associated with the field/attribute) and a method on which it calls the setter of the field (*Method method = propertyDescriptor.getWriteMethod();method.invoke(instance, value);*)

the updated instance is added to the list;

i) ***insert():*** connects to the database through ConnectionFactory class; initialises a prepared statement with the method createInsertQuery(), initialises a result set for the query, prepares the statement through reflection (by parsing the fields to get the wanted values) , executes an update and stores the generated key in the result set; closes the result set, the statement and the connection;

j) ***update():*** is very similar to the insert() method; the only difference is the method createInsertQuery() is replaced by createUpdateQuery() and the fact that the id (PK) of the entry is also used in the prepared statement;

k) ***deleteById():*** connects to the database through ConnectionFactory class; initialises a prepared statement with the createDeleteQuery(), executes an update and returns 0 if successful, -1 otherwise; closes the result set, the statement and the connection;

- ***TClientBL, TOrderBL, TProductBL, TCountryBL classes***
  Are virtually the same; they validate the input through the ***Validator*** classes; make use of both the DAO layer and the models in map them one-to-one with their respective methods; error messages are provided if the methods do not execute successfully
- ***ReflectionExample class***
  Contains methods that use reflection in order to retrieve information, check the stock and update the stock for a certain input parameter; the last two are used exclusively for the 'Insert' order action, while the retrieval of data can be employed on any model object; the implementation will not be explained as it is quite similar to the one used in the AbstractDAO class in createObjects or even insert() and update();
- ***Controller and View classes***
  The View is pretty straight forward; it contains the components and containers and initialises them, as well as maps the listeners to the buttons

  a) ***Controller():*** the constructor initialises the attributes

b) ***addListeners():*** deals with the action to be performed when a button is clicked; it parses the input and provides corresponding error messages;

   each button is dealt with separately and accordingly; the input is prepared; the DAO and model objects are created and the corresponding method is applied (delete for the delete buttons, insert for insert, etc.);

   the 'Place order' button creates a bill for a given client (given by id) that is stored in a .txt file

   the 'View' buttons create a table through reflection; the createTable() method receives a list of objects as parameters and returns a JTable populated with the entries of the list;

c) ***Main():*** instantiates the basic components to run the application

RUSU CARLA MARIA
GROUP 30421

Results

The results for generating a bill for client with id 2 and client with id 3 are given below as examples:

```
client_information:
id_tclient=2
name=PATRICIA
address=53 Idfu Parkway
email=PATRICIA.JOHNSON@sakilacustomer.org
age=25
id_tcountry=1
country_of_destination: FRANCE

list_of_products:
GRAPHITE,       amount: 10,   price_per_unit: 6,       total_price: 60
HB PENCILS,     amount: 15,   price_per_unit: 3,       total_price: 45
total_amount_to_pay: 105
```

```
client_information:
id_tclient=3
name=ROBERT
address=613 Korolev Drive
email=ROBERT.BAUGHMAN@sakilacustomer.org
age=20
id_tcountry=3
country_of_destination: ZIMBABWE

list_of_products:
B2 PENCILS,     amount: 30,   price_per_unit: 4,       total_price: 120
total_amount_to_pay: 120
```

Conclusions

To conclude, the application meets the requirements: has an OOP design, is organized in packets, its methods are under 30 lines and it respects Java naming conventions. An intuitive graphical user interface on which CRUD operations can be done is provided, as well as the ability to create bills and creating a table using reflection techniques.

While designing this application I have further developed my OOP knowledge by acquiring some insight on layered architecture and I have expanded my understanding on the MVC style. For the first time, I had the chance to learn about the power of reflection, generics and the singleton pattern, how they relate to reusability and efficiency in coding. By the end of the project I can safely assert that I have enhanced my understanding of database applications in java.

Bibliography

- http://users.utcluj.ro/~igiosan/teaching_poo.html
- http://users.utcluj.ro/~cviorica/PT2019/
- http://coned.utcluj.ro/~salomie/PT_Lic/4_Lab/HW3_Tema3/Tema3_HW3_Indications.pdf
- http://tutorials.jenkov.com/java-reflection/index.html
- https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html