DOCUMENTATION

ASSIGNMENT 3

STUDENT NAME: BÁLINT ROLAND
GROUP: 30424

TABLE OF CONTENTS

1. Objective

The main objective of this assignment is to deliver an *Order Management Application*, capable of interacting with a database composed of three tables, called clients, orders, products. The application should supply an easy-to-use graphical interface to allow the user the creation and manipulation of data. The desired outcome is an application which could be used in a real-life warehouse/ mini shopping center.

To achieve this, certain secondary aims should be fulfilled. The most important of them is to provide a correct and functional, relational database of at least three tables. Ensuring the connection between our MySQL database and Java program is essential, or else executing queries would be impossible. There should be certain constraints and validations from both levels, to filter user input.

Another secondary goal is to learn new Java concepts. In this context, we are interested in:

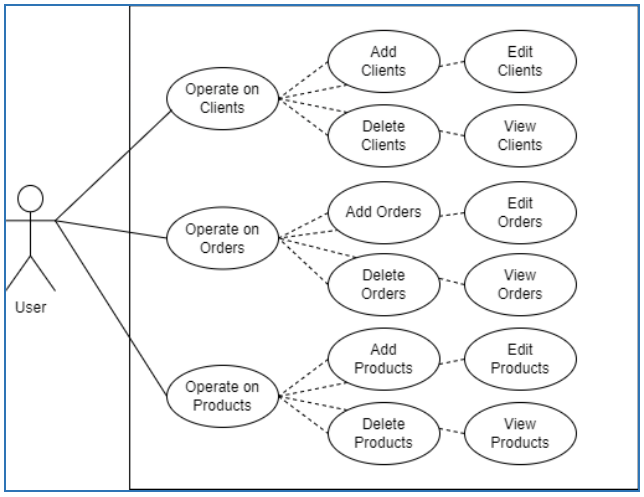
- Java Generics
- Reflection in Java
- Writing and generating Javadoc
- Making reusable code, reduce code repetition

The implementation phase of this project could also be a perfect opportunity to get more familiarized with Java Swing components (ex: *JTable*s, *JComboBox*es, *JLabel*s), specific Java libraries and the functionalities they provide (ex: *java.sql.Date*, *java.awt.event.MouseAdapter*, *java.beans*, *java.lang.reflect*).

2. Analysis, modeling, scenes, use cases

The theme of this assignment is easy to grasp, hence there is no need to explain what such application should do. It is important though to restrict the capabilities of the project to certain concrete functionalities, otherwise it is possible to quickly become overwhelmed and get distracted from the main goal. An order management application can be designed in many ways and in many levels- my goal was to fulfill every requirement of the problem statement, as well as make the project realistic, useable.

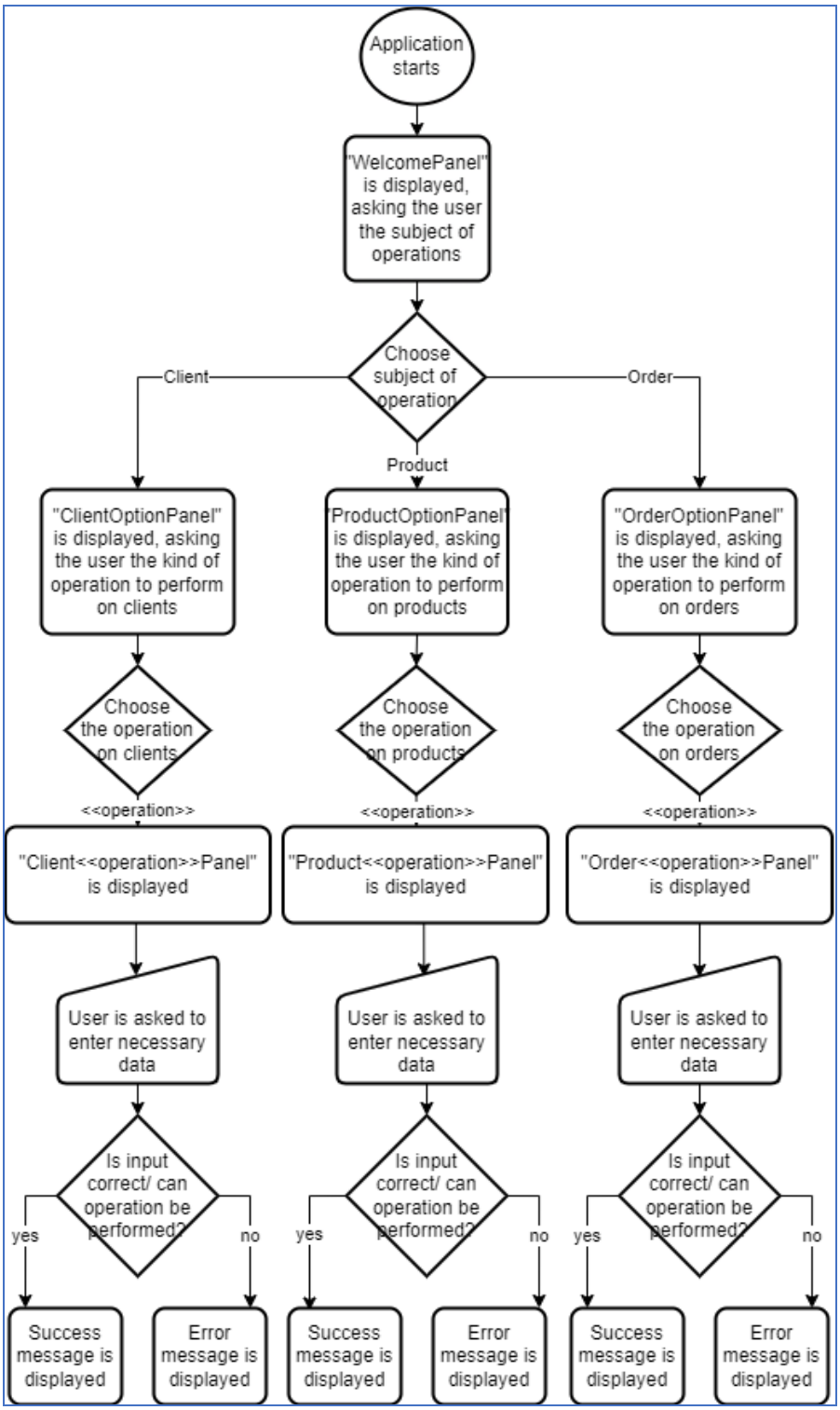
To clarify these goals, a use-case diagram will follow:



Use-Case Diagram

It is very important to reduce the problem to smaller chunks, to modularize it, since this is a key concept in Java, and in programming in general. Having this view over the problem, we can instantly start thinking about possible implementations of graphical user interface elements, of operations on the data, of similarities and differences between these smaller parts.

For me, even if this is against the recommendations, in case of these projects it is easier to think of the GUI first, and then ravel the backend around it. Obviously, there is no singleton correct solution for this synthesis. My approach was to give a separate panel for each functionality. Each operation will have a separate class. These may be reduced later, using generics and reflection where needed. The design choices enlisted so far give a structure for the whole program, resulting in the following flow of scenarios:



Flow Chart of Scenarios

To keep the diagram relatively simple, I avoided to display every operation separately (after decision “choose the operation on XYZ”, the possible choices for <<operation>> are Add, Edit, Delete and View), since the path taken afterwards is very similar, at least on an abstract level (except View, since in this case user does not need to enter any additional data). The flow of states looks like an ever-expanding tree, and judging only based on the earlier drawing, it looks unidirectional. This is another abstraction made by me to avoid overfilling the schematic: the user has the possibility in any step to go back to the previous state using a *back* button. It may seem like you have to make a horrendous number of clicks for nothing, however this kind of separation between operations on different entities seems very logical to me, the user needs to know at each moment where he or she is, and accidental operations should be minimized.

3. Design

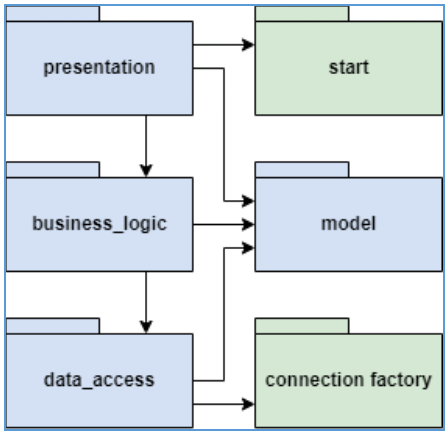
Since this project manages multiple operations with multiple inputs, it is hard to define an exact black box of the system. Keeping this in mind, a very generalized form of it might look as follows.

Overall System Design

By choice of subject, I mean choosing if we want to work with orders, clients, or products, while by operation on subject I mean the possible operations on the chosen type of entities (*add*, *delete*, *edit*, *view*). We talk about concrete subject of operation in case of the *delete* and *edit* options, since we need to select one specific entity to work with. Aside from these choices, the user must also supply the necessary (and correct) data if it’s the case. When adding a new entity, these data stand for a new object, while when editing, they will help in changing an existing object.

The outputs of the system aren’t concrete outputs (except the text file generated as a bill, which wasn’t included in the diagram since it’s just an extra feature), they rather refer to handling the database linked to the application and displaying its contents in a user-friendly way. The difference between current objects in database and “local” objects will be explained further on, in the implementation section.

To start off, let me introduce the package diagram:



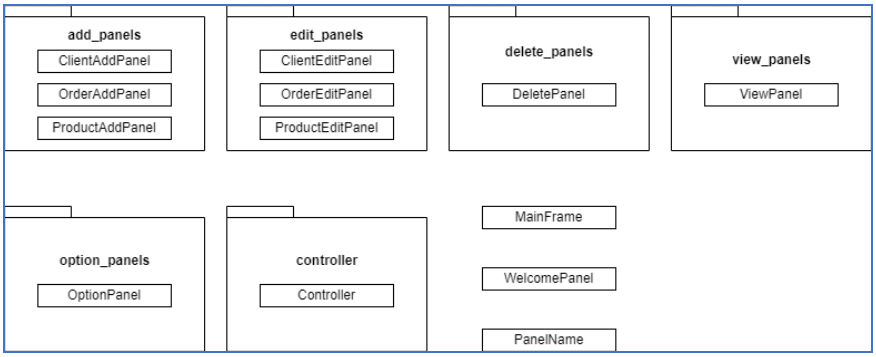
Package Diagram

The main idea behind the design of packages and their relation was to use a *layered architecture*. It was heavily inspired by the sample program provided. The main idea of such a layered structure is to define, as its name suggests, different layers for the implementation. Each layer has a different role, and they can communicate with the one “above” and “below” them. The minimal structure of such a layered architecture is represented by the packages colored with blue. Packages *connection_factory* and *start* are just an addition to make it even more elegant. The first

one ensures the connection with the database, with the usual setup, the later one holds the class *Start*, having the only *main* method of the project. This way if someone looks at the project, he/she can easily find the starting point of all, or he/she can modify the connection details (URL of host, username, password) without digging deep into the implementation. The rest of the packages have fundamental roles, which I will explain in detail in the following sections.

a) Package *presentation*

This package contains every class related to GUI, as well as the Controller class. Since there are many personalized interface components for each possible outcome, I further divided this package in some sub-packages. It may look like it's an exaggeration, but it helped me a lot during implementation, and in my opinion, it makes the project easier to follow.



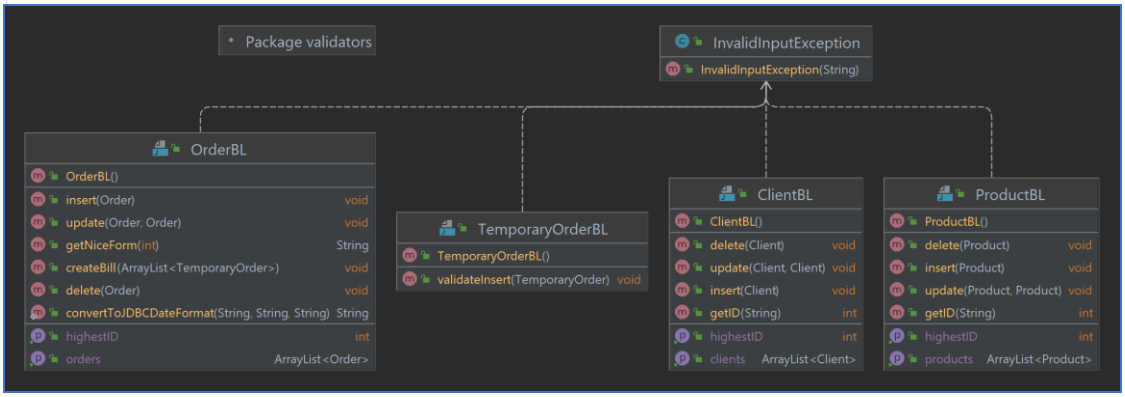
Package Diagram - Package "presentation"

My train of thought was the following: there are a lot of panels to be implemented, but essentially, I could use the same frame. It would be very inconvenient for the user to have to deal with a new frame popping up at every button press. In one of my past projects, I experimented with Java Swing's *card layout*. Even if it hardened the implementation at the beginning, I think the design choice I made was proper: I have a single *JFrame*, named *MainFrame*. It will hold info which will be displayed throughout the whole running state of the application: the title/ name of the project in the header, and the credentials in the footer. In the middle, the current panel will be displayed- but more on this later, in the implementation section.

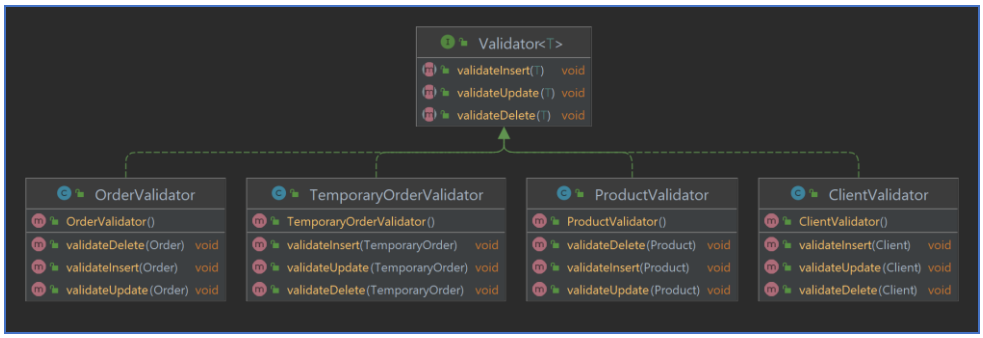
The controller package contains the Controller class, which has the usual role of a controller: adding action listeners, implementing them, listening to them, acting as asked using business logic layer's tools.

b) Package *business_logic*

This package sums up the business logic layer of this application. It has two main roles: supplying the necessary logic and validation for each type of data. Its class diagram:



Class Diagram - Package "business_logic"



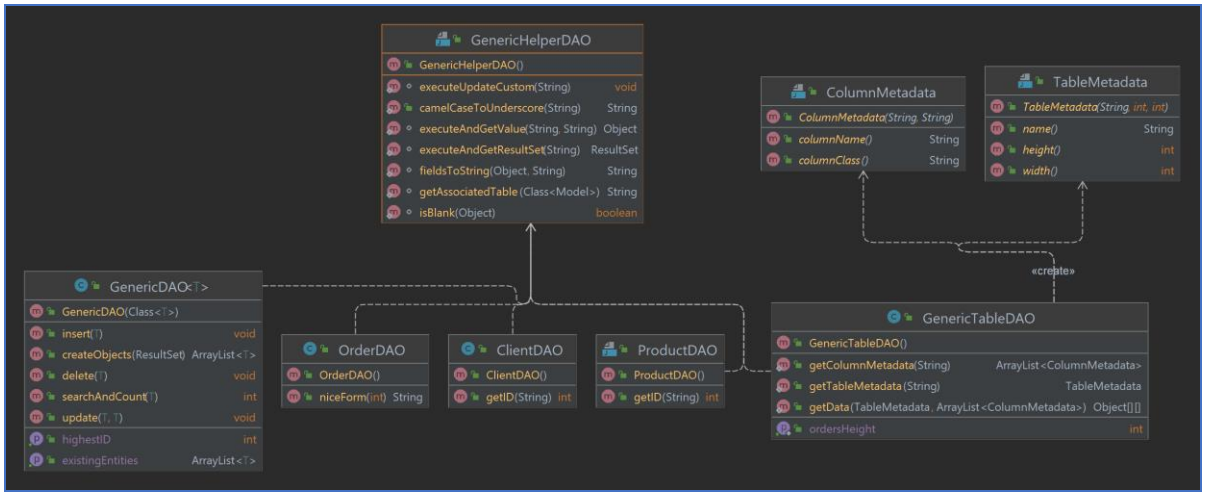
Class Diagram - Package "validators"

The little sub-package called *validators* contains the implementation of *Validator* interface for each modelling class. Validations need to be done before doing any of the *insert*, *update* or *delete* operations. We will see in the implementation section that it some cases there are no validations needed, but for now, it is nice to enforce these three methods on each validator class.

About the other class, we can see that there are certain type-specific operations needed, such as billing, handling conversions from user input to *java.sql.Date* in the case of orders or getting the ID of entities based on their unique characteristics, in case of clients and products. Even if most of these methods are database related, I did include them here to respect the layered architecture's constraints (presentation layer cannot interact directly with the data access layer). Using fancy words, I used the *delegation technique*, using simpler words, I just declared these methods, which will call the respective methods from the data access layer. Validation is done inside these methods, whenever needed.

c) Package *data_access*

This package holds the classes above:



Class Diagram - Package "data_access"

First, there are DAOs (Data Access Objects) for every class modelling rows from database's tables: *OrderDAO*, *ClientDAO*, *ProductDAO*. Since the requirement was to use the *reflection technique*, some queries were written such as they work for any type of model class- these can be found in the *GenericDAO* class. *Insert*, *delete* and *update* are all generic. The *createObjects* method is a modified version of the one present in the sample code, and it is used when filling the dropdowns in delete/ edit panels.

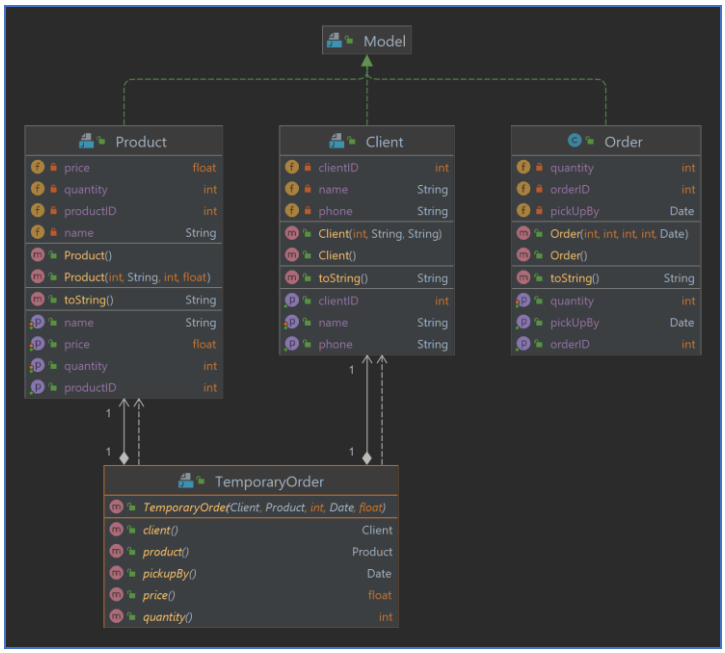
GenericTableDAO has the methods needed for viewing the current entities, i.e., for filling a *JTable* in a generic way. By delivering the table's name, this class can get the height and width of the table and every object to be inserted. These are data necessary for the constructor of *JTable*. I also, for the first time, experimented with Java

Records, *ColumnMetadata* and *TableMetadata* being two records holding data about what their name suggests, making the code easier to read and understand.

GenericHelperDAO is a static class (has only static methods) and contains non class related auxiliary methods to reduce code repetition and have code of better quality. There are certain methods which may look very random, at first glance it is hard to imagine their usage in the current context, but if we look at the implementation, everything will make sense.

d) Package *model*

This package encloses the classes which model rows of tables from the database. In order to make our application faster, and to apply our Java knowledge rather than wander around in the magic world of SQL queries, it is desired to work on these classes, making changes locally, then updating the database based on these objects' state.



Class Diagram - Package "model"

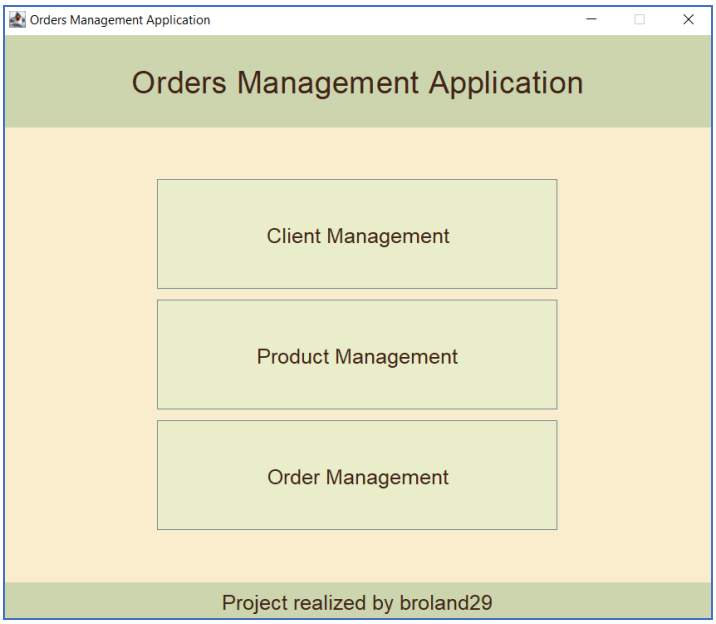
There is also an impostor among these classes: *TemporaryOrder* does not reflect any real database table. I introduced it to make the insertion of orders more realistic: an order can have multiple sub-orders, these can be added, removed, added again etc., all of these being expensive operations. We can do this locally but displaying the local orders in a suitable way (not just IDs, but names of products and clients) also requires database access. Hence, *TemporaryOrder*s were created, which are records as well, since they are just simple data holders, having every field final.

4. Implementation

Finally, I can talk about the actual code behind all these abstractions. To keep everything organized, I will present important things from each package in the order imposed by section 3.

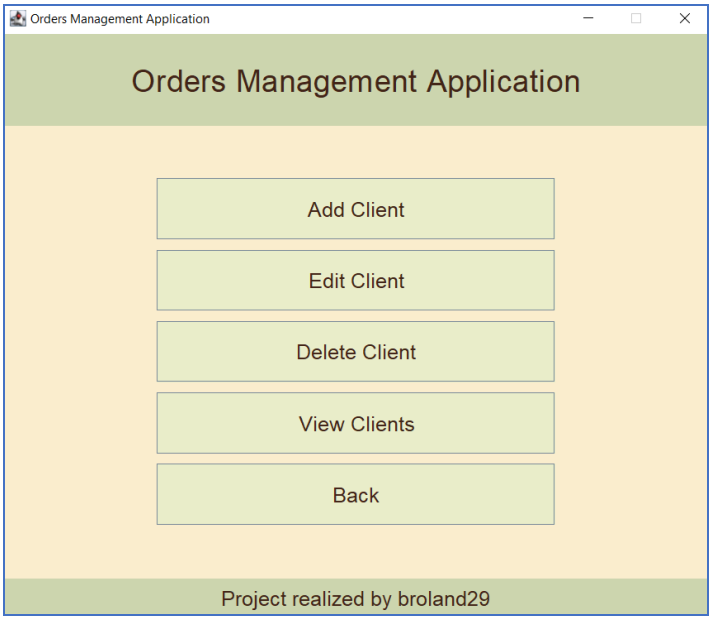
a. Package *presentation*

As mentioned before, every panel is inside the *MainFrame*'s *ContentPane*, which has *CardLayout* set. Each panel is instantiated at startup, but data is not filled yet- it is important to show up-to-date data for the user, hence data is fetched every time an operation button is fired. As a beginning, this is the first thing the user sees when he/she starts the application:



WelcomePanel

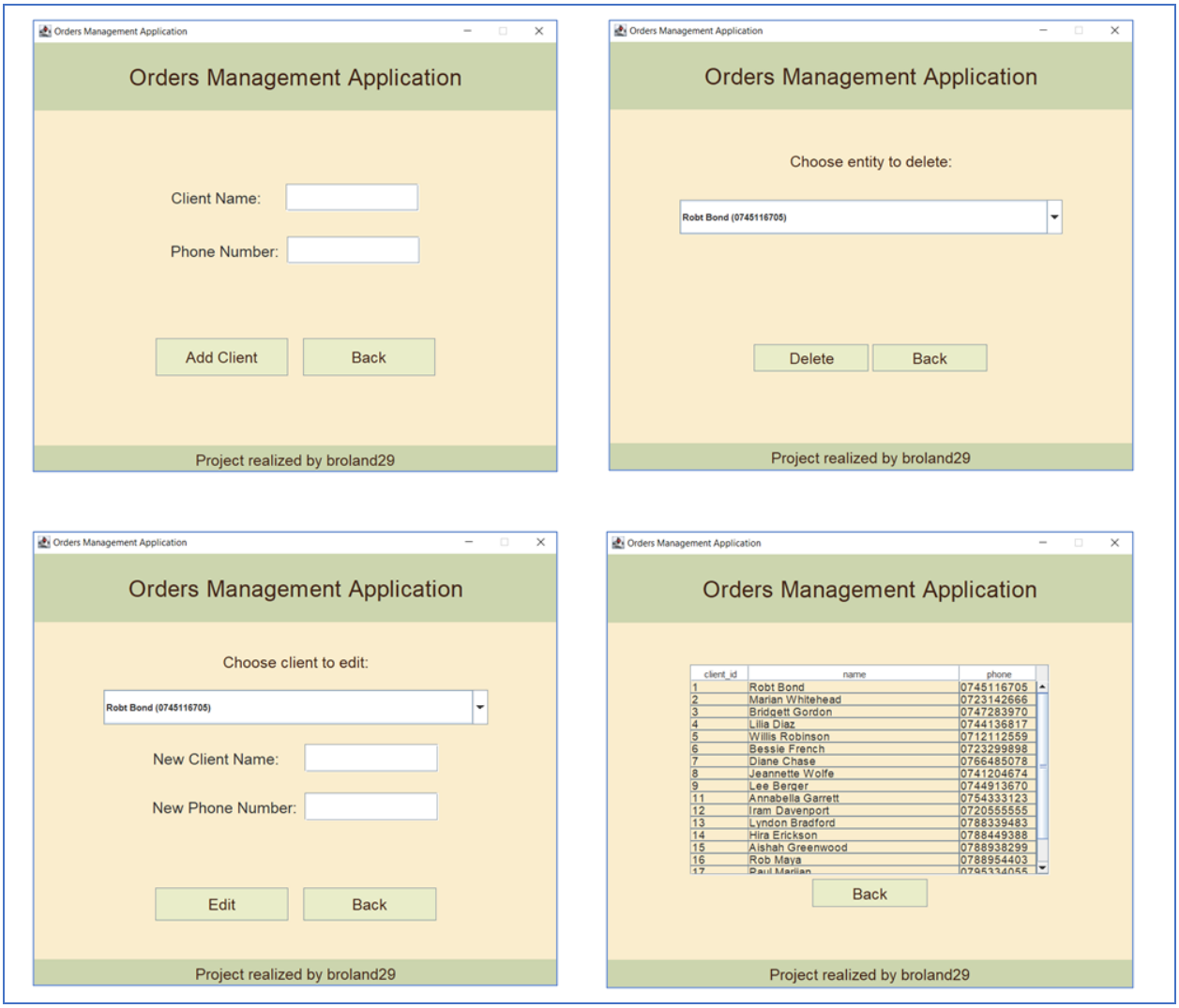
I tried to match the overall vibe of the previous projects by choosing soft and calming colors. By displaying the options on startup, the user can directly jump in action and choose the desired subject of the operations. After choosing one of the three options, the user is asked to further specify what he/she wants to do, by selecting the correct operation. These choice panels only differ in their text and the places they lead you to. If the *Client Management* option is selected, the next choice will be made using the following interface:



ClientOptionPanel

Now we can see that indeed, I was not lying, because the possibility to go back using the *back* button is provided. For now, going back from any of the three option panels leads to the *WelcomePanel*, but when coming back from a specific operation's panel (ex: *ClientAddPanel*, *OrderDeletePanel*, *ProductEditPanel*) the paths will differ. I used the idea of remembering the last panel, so that we can switch back without more complicated computations or switches. Note that in my implementation the back button always takes you back "one layer on the tree".

If user chooses to do operations on *Client*s, one of the following panels will appear:



Every Client-Related Panel

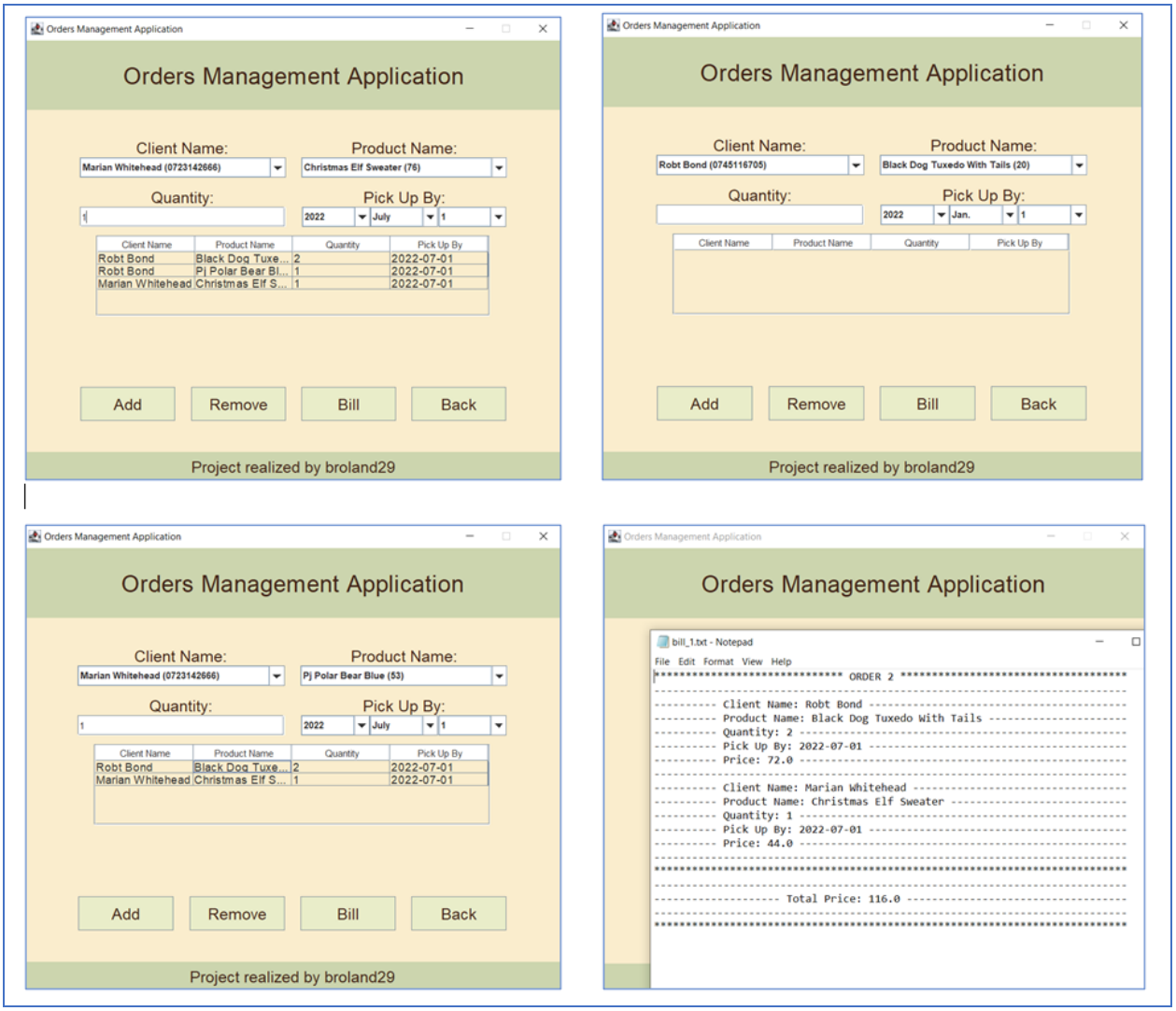
When adding a new client, user must enter the client's name and phone number. Its ID will be generated automatically (there is a query selecting the maximum ID for clients, this value is incremented and passed as argument for the constructor of *Client*). Editing a client is very similar, user has to enter the new data and the user he/she wants to edit (using the dropdown). Deleting a client is straight-forward, just select the client and press *delete*. Viewing clients happens through a table, displaying every available data.

The panels for operations on products are very similar, except they expect and show data related to products.

The order-related panels are a little different, given the quirky nature of orders. When talking about orders, we also talk about client and products. Hence, there needs to be a dropdown for both. Inserting new clients or products is done elsewhere; here, we restrict the user to choose from the existing ones. In order to add an order, we also need a quantity (entered through a regular text field) and the pick-up date (chosen from three separate dropdowns, one for the year, one for the month and one for the day). As I mentioned before, an order can be composed of multiple sub-orders. In my implementation, the sub-orders are unrelated, they do not affect each other, hence there can be a sub-order for one client on one pick up date and another one for another client, in another date. This is not the best approach, I have to admit, but my database was designed this way, only to support the minimal requirements for a functional application, on the principle that I will settle for less, as long as I am satisfied with the final product.

The user has a certain flexibility when composing an order, it is possible to add, and also to remove sub-orders. These pending orders affect the objects locally and do not have influence on the database yet. It can be instantly seen that, for example, after putting a certain amount of some product in our bucket list, the amount of that product on stock will decrease. When removing a temporary order, the reverse phenomena is observable.

When the user presses the *bill* button, a bill (text file) is generated, and its automatically opened up. This is the moment when the local changes are transposed into the database. The temporary orders are transformed into "real" Order objects, and the insert method is used on them. In the same time, the quantity is updated, by getting the related *Product* object from every *TemporaryOrder*, and performing an update query with their help. The following images depict different moments of adding an order into the system.



ClientAddPanel - Scenarios

Editing orders is done in a similar way (except there is no table involved). Deleting orders is done through a dropdown, in the same style as in the case of products and clients. Viewing the orders is also a little different from the other two views. Even if the idea is the same, there is a little trick made in the background, invisible to the user. The viewing tables for clients and products are a one-to-one mapping of the database tables into java swing tables. This would make no sense in the case of orders, since the orders table consists of client and product IDs. For checking which client or product has the given ID, the user would have to navigate back to client, respectively order management, search for them etc. – very inconvenient and it basically makes the whole table useless. To avoid this, instead of a usual *SELECT * FROM orders*, a custom query is ran, even if this kills the "generality" of the view class.

```
SELECT o.order_id AS 'id',          c.name AS 'client_name',
       c.phone AS 'client_phone',   p.name AS 'product_name',
       o.quantity AS '#',           o.pickup_by AS 'pick_up_by'
  FROM orders o
  JOIN clients c  1..n<->1: on c.client_id = o.client_id
  JOIN products p  1..n<->1: on o.product_id = p.product_id
 ORDER BY order_id
```

Query for OrderViewPanel

Similarly, the width of the tables is customized, because else product names, for example, cannot be seen, because the space is taken up by columns which do not need so much space (like ID or quantity). The program works fine without these additions; I know that the requirement was to make it generic, but I find the modifications made by me justified.

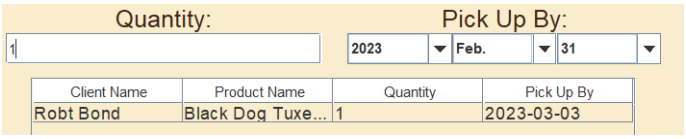
b. Package "business logic"

As I was explaining previously, the business logic holds classes which make it possible to validate and operate on model objects. First of all, the validation is done through validator classes. Each validator class implements the *Validator* interface, which has methods *validateInsert*, *valindateUpdate*, *validateDelete*. We will see that in some cases, there is no need to do a validation, but even if this is the case, using such an interface makes the code more readable and makes its possible future expansion easier.

Validating a client is straight-forward, but even here there are certain wicked things which are easy to forget. When inserting, the client's validator class checks if the user input is correct: the name should have a length between the minimum and maximum allowed length (the existence of minimum length makes empty inputs invalid). The phone number should obey the Romanian phone number format, i.e., it must have a length of 10, it must contain only digits, and its first two digits must be *07*. Also, a very important criteria is the uniqueness of the phone number. Even if every table has an ID field, this field has no significance for the user, its sole purpose is to speed up queries, as it is a good practice to use short integer IDs as primary keys in tables of relational databases. So, each table should have another column (excluding the ID) which is unique, so that the user can distinguish people from one another, avoiding conflicts and ambiguity. In the case of clients, this unique field is the phone number. When the user presses *Add Client*, a query is executed in the background, checking if the phone number is already in use or not. When updating a client, the same method is called for validating the "insertion" of the new version of the client (after all, updating is a kind of insertion as well). When deleting, there is a thing to check that may not come as intuitively, that is if the client is in an active order. Attempting to remove a client which has at least an order active would result in a foreign key constraint.

For products, the same ideas are used, except we need to check for the price to be greater than zero, and the quantity to be greater or equal to zero. I allow the zero value for quantity with the idea that there may be products which will arrive in the future, but we want to insert it into the database even if there is none of it in stock at the moment.

Before discussing the validation of orders, it will make more sense to first analyze the one of the temporary orders. Even if these are local, we shouldn't allow the user to introduce in the table orders which are impossible to make. This would create the illusion that everything is fine until the *bill* button is pressed- then we would have to make the error messages even more specific, showing the source of error, and there may be many wrong temporary orders that cannot be executed- the user would get very frustrated, with reason. In the case of temporary orders, there is no edit option (the user can easily drop the wrong temporary orders and insert a new, correct one). When addition is performed, we have to validate the quantity, since this is a user input, not a user choice from existing, correct data. Aside of this, we have to make sure that there are indeed enough products to satisfy the customer, and we also have to check the date- it should be in the future, after the current date. Something that I would call a not so elegant solution is that the days for each dropdown are fixed, for example, even if February is selected as month, there are options to select 30, 31 as day. Here I could have added some extra logic in the background to change the days according to the months selected, but I considered that this would take too much unnecessary effort. I checked how the *java.sql.Date* translates such a date, and for example, February 31 is transformed into March 3rd, so I think this is a logical way of correcting user input, and it is implemented by the java developers, so I suppose that it works in other cases too.



Correction of Dates

In the case of orders, since temporary orders are already valid, there is no further need to check when inserting their order form. However, when editing, no temporary order is involved- we have to perform a similar routine as in *TemporaryOrderValidator*'s *validateInsert*. Deleting an order can be, of course, without any preconditions.

Moving on, I also made a custom exception class called *InvalidInputException*, just to distinguish it from other kind of exceptions. Outside of package validators, we can find the classes standing for the business logic of each class apart. These combine the validations I explained previously with the data access objects' functionalities, packing up these two so that it can be used easily in the presentation layer (in the *Controller* class), without worrying about small details. There are also some methods which supply extra logic in some cases, for example, in class *OrderBL* we have method *convertToJDBCDateFormat*, which is very helpful both in the case of adding an order and editing an order, so that the conversion from user input into *java.sql.Date*, and from *java.sql.Date* to *MySQL date* is possible. In the same class we can find the definitions of constants *YEAR_OPTIONS*, *MONTH_OPTIONS*, *DAY_OPTIONS*, since this avoids ambiguity and repeating code (in *orderAddPanel* and *orderEditPanel*). The definition of the methods generating the bill can be found here as well.

c. Package *data_access*

This package is the home of the DAOs (Data Access Objects). To keep everything organized, using the previously applied tactic, there are separate classes for each model class. *ClientDAO* and *ProductDAO* carry methods for getting the ID of an entity based on its unique property (needed when transforming temporary orders into orders). *OrderDAO* embodies method *niceForm*, which executes a query similar to the one used when generating the view table for orders. This method is used in the *toString* of *Order*, but I will talk about this later. For now, it is enough to notice that while this may have seemed as a good idea at the beginning, it made working with *Order* class extremely inefficient, so much that it led to the introduction of a new class, *TemporaryOrder*.

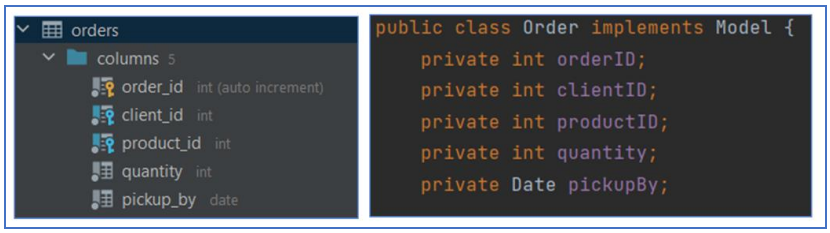
```
public String niceForm(int orderID){
    String query =
            "SELECT c.name, c.phone, p.name, o.quantity, o.pickup_by "      +
                    "  FROM orders o "                                      +
                    "  JOIN clients c on c.client_id = o.client_id "        +
                    "  JOIN products p on o.product_id = p.product_id"      +
                    " WHERE order_id = " + orderID                          ;

    ResultSet resultSet = executeAndGetResultSet(query);

    try {
        resultSet.next();
        return
                resultSet.getString( columnIndex: 1)          + " ("      +
                        resultSet.getString( columnIndex: 2)  + ") - "    +
                        resultSet.getString( columnIndex: 3)  + " ("      +
                        resultSet.getString( columnIndex: 4)  + ") - "    +
                        resultSet.getString( columnIndex: 5)              ;
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return null;
}
```

Method "niceForm"

One may wonder, why are there so few methods in these DAO classes… Well, it is because the other ones were made generic. These can be found in the *GenericDAO* class. This class will be instantiated once for every class apart, but it reduces code repetition enormously. It also caused a lot of headaches, but it was worth it. *Search*, *update*, *edit* and *delete* methods are implemented in such a way, that they accept entities of generic type T (following the convention), and they get the name of the table based on the class of T. The class of T, again, was a

big paradox to solve, since as far as I understood, the class of a generic cannot be known solely from the generic. There was a crazy one-liner (in the constructor of the *AbstractDAO* given to us as sample code) that I could not understand. I searched for it on the internet, and found the source of it (stackoverflow, of course), but when I digged deeper in the forums, I read that the code does not always work, and it's a bad practice- someone even called it "plain nasty", (see: last link it in the references section). So, my solution to this was to have a class field of type *Class<T>* in my *GenericDAO*, which gets value in the constructor. I don't know if it's correct, but it made more sense to me. I also took the *createObjects* method from the sample, and changed it to match my needs, but I talked about this already.

Class *GenericHelperDAO* is a static helper class, its methods mainly refer to executing a query in some way, with the purpose of making my code shorter and easier to understand. There are two methods, which provide some kind of mapping between the MySQL and Java part of the application. Method *camelCaseToUnderscore* had to be introduced in order not to break the naming conventions of both languages. We are talking about the fields of modelling classes and the column names of the tables.



Mapping of Table "orders"

The order mapping class is *getAssociatedTable*, which takes as argument a parameter of type *Class*, and returns the name of the table associated with that class, in a *String* form. It is important to match the real names of the databases, since this value is often used as an element of a query. Method *isBlank* simply checks if the value of the object is considered "blank" or not.

About records *TableMetadata* and *ColumnMetadata* I already talked. I think records are a nice feature in Java, they help in reducing boiler plate code and they also automatically implement other useful functionalities aside getters and a constructor, such as an appropriate *toString* method.

GenericTableDAO holds methods needed for generating *JTables* based on the database's current state, but I already explained this in the Design section.

d. Package model

This package consists of the classes which model data from database, as well as the *TemporaryOrder* record. One thing that I did not mention up to this point that I have an interface *Model*, which is simply a marker interface to show which are modelling classes. Again, I'm not sure if this is a good addition or not, but my motivation behind is that using this interface I was able to restrict some generic methods, for example, in the case of *GenericDAO*.

```
public class GenericDAO<T extends Model>{

    Class<T> modelClass;
```

First Lines of Class "GenericDAO"

Another neat invention of mine if that I used overriding on every model class's *toString*, in order to have a user-friendly String form of them which could be displayed in dropdowns or tables. While this in itself is a legitimate idea, the delivery of it might not be following the guidelines. I know that our teacher mentioned during the courses that the theory states that in order to override such methods, there are certain rules to be aware of, like checking if the value is null, etc. Well, in practice, I think that most programmers do not obey these rules. I just wanted a nice string form.

```java
/** Overridden toString method, showing data of interest*/
@Override
public String toString(){return name + " (" + quantity + ")";}
```

Overriding the "toString" Method of Class "Product"

However, in the case of orders, I have to admit that calling a query in each *toString* method is not the best idea. It would have been much smarter to collectively gather this kind of information, so that instead of executing one query per order we would execute only one. But I am satisfied with the current speed of my application.

5. Results

The result is the desired product, even if during the implementation phase my image of the final product changed its shape. My “testing framework” was my family, I let my sister and my dad play around with it. It was a clever idea to show them my project because my image of a potential user of this application matches their personality, from a programmer view: a person without any knowledge (and interest) in informatics, but someone who wants a working product, wants an easy-to-use interface, wants it to be flexible and match his/her needs. Us, programmers, often get stuck on insignificant details, like what color to use for the interface, how big should the button be, should we use if-s or switches, should we write generic code or just duplicate code and change it for our specific needs, what architecture to use, what libraries to use...

And while me, saying that these are insignificant details, might sound crazy, trust me, the end user does not care. For example, I spent hours and hours implementing the project, but what did I forgot? The most important thing: displaying the amount of money on bills. My little testers instantly saw this. Of course, every detail has its effects, on optimality, speed, quality of GUI, possibility to extend. Well written code is art. But in the end, in projects as this, the final product speaks for itself.

6. Conclusions

Overall, I really liked the idea of this project since the theme seemed very real to me. I can imagine my application being used in a small warehouse, in a local shop or business. This is why I took the time to bring it on an acceptable level, because I felt like rushing it for the good grade would have filled me with regret.

This is my biggest Java project up to this point, and I have learned a lot while implementing it. Layered architecture seems like a really logical design choice. I had to use the knowledge gathered in the previous semester on SQL queries. I had to deal with very small bugs in small details. For example, at first my product select query did not work. I took the query in a query console, ran it, returns 0 results (even if I searched for an existing item). As it turns out, you should never write queries on float numbers, because sometimes they just don’t work.

I used Javadoc for the first time, even if I heard about it, I never tried it out before. I wrote generic methods, generic classes, I experimented with reflection, but even now it feels like black magic. It took way more time to finish than I expected, but at least it is done, and done good.

Possible future improvements:

- Make program even faster by perfecting the queries and accessing the database less times
- Make sure design choices are right, ask a professional
- Make sure layered architecture is used as it should be
- Make the code even more generic (without making it less readable)

7. Bibliography

Materials given to us by the course teacher:

https://dsrl.eu/courses/pt/materials/PT2021-2022_Assignment_3.pdf
https://dsrl.eu/courses/pt/materials/A3_Support_Presentation.pdf
https://gitlab.com/utcn_dsrl/pt-layered-architecture

Drawings created in:

https://app.diagrams.net/

Javadoc:

https://www.jetbrains.com/help/idea/working-with-code-documentation.html#generate-javadoc

Cute doggy site for data in database:

https://www.poshpuppyboutique.com/

SQL dump:

https://dev.mysql.com/doc/workbench/en/wb-admin-export-import-management.html

Other:

https://www.geeksforgeeks.org/static-class-in-java/
https://www.computerhope.com/issues/ch001311.htm
https://www.behindthename.com/random
https://www.youtube.com/watch?v=EAxV_eoYrIg&t=452s
http://www.java2s.com/Tutorial/Java/0240__Swing/ResettingtheViewportPositionmoveJScrollPanetothetop.htm
https://stackoverflow.com/questions/6167585/open-excel-document-in-java
https://www.tutorialspoint.com/java-sql-date-valueof-method-with-example
https://www.tutorialspoint.com/how-to-add-a-new-row-to-jtable-with-insertrow-in-java-swing
file:///D:/downloads/Regular_Expressions_Cheat_Sheet.pdf
https://www.geeksforgeeks.org/stringbuilder-class-in-java-with-examples/
https://www.youtube.com/watch?v=jUcAyZ5OUm0
https://www.java-forum.org/thema/unchecked-cast-from-component-to-jcombobox-string.139501/
https://www.javadrive.jp/tutorial/jtable/index12.html
https://stackoverflow.com/questions/953972/java-jtable-setting-column-width
https://www.w3schools.com/mysql/mysql_join.asp
https://www.tutorialkart.com/pdf/java/how-to-update-an-element-of-arraylist-in-java.pdf
https://www.youtube.com/watch?v=sAReaaTxNGU&t=427s
https://web.archive.org/web/20180124022935/http://blog.xebia.com/acessing-generic-types-at-runtime-in-java/