Technical University of Cluj-Napoca

Year 2016-2017

Department of Computer Science

Programming Techniques

3rd Assignment

Student: Gyarmathy Tímea

Group: 30423

Table of Contents

[Specification of the problem 3](#_Toc480532185)

[Other classes and packages can be added to implement the full functionality of the application. 3](#_Toc480532186)

[Requirements 3](#_Toc480532187)

[Description 3](#_Toc480532188)

[The analysis of the problem 3](#_Toc480532189)

[Modeling 4](#_Toc480532190)

[The database 4](#_Toc480532191)

[The application 4](#_Toc480532192)

[Use cases 5](#_Toc480532193)

[Scenarios 6](#_Toc480532194)

[Design 7](#_Toc480532195)

[Design concepts 7](#_Toc480532196)

[Package diagram 8](#_Toc480532197)

[Class diagram 8](#_Toc480532198)

[The Model 9](#_Toc480532199)

[The Data Access 9](#_Toc480532200)

[The Business Logic 12](#_Toc480532201)

[The Presentation 13](#_Toc480532202)

[Class Main 14](#_Toc480532203)

[Sequence diagram 15](#_Toc480532204)

[Graphical User Interface 15](#_Toc480532205)

[Implementation and testing 19](#_Toc480532206)

[Results 20](#_Toc480532207)

[Conclusions 20](#_Toc480532208)

[What have I learned? 20](#_Toc480532209)

[Further developments 20](#_Toc480532210)

[Bibliography 20](#_Toc480532211)

# Specification of the problem

Consider an order management application 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:

1. Domain specific classes: **OrderDB**, **Customer** and **Product**
2. Business Logic (warehouse-specific processing) classes**: OrderProcessing**, **WarehouseAdministration, ClientAdministration**
3. Presentation classes**:** GUI related classes
4. Data access classes**:** Database access related classes

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

# Requirements

1. Analyze the proposed application, determine the structure and behavior of its classes and draw an extended UML class diagram.
2. Design, implement and test the application classes. Use javadoc for documenting classes.
3. Define, design and implement a system of utility programs (examples: reports for under-stock, totals, filters, etc.).
4. Design and implement a comprehensive demo driver for the order management application.

# Description

Databases are widely used to store large amounts of data, thus applications on managing these data are common and necessary nowadays. Databases contain tables which are related in one way or another to each other, and for efficiency they should contain information relative to one distinguished real world model. However, data from a database has access rights relative to the person accessing them, so an administrator can add, update and delete data, while a simple user should only view, or retrieve information, the actions upon them should be implemented and done in the background of a user application.

Warehouses store large amounts of products, which can be managed, updated, and a customer can place an order consisting of several products which will be retrieved from the storage.

Modeling a customer order processing system for a warehouse should also consider these concepts, hence the application has separate options for when a customer accesses it than when an administrator does. The basic CRUD (Create, Retrieve, Update, Delete) operations are implemented and directly accessible by the warehouse administrator, and indirectly used by the customer when placing an order.

When the user opens the application, he can choose whether he wants to perform administrative operations, like adding, editing, deleting or listing customers or, respectively, products, or he wants to place an order. When placing an order, the user choses a customer, and adds products to its order, each in a desired quantity. When the order was finalized, the total cost will be shown to the user and a “bill” will be generated: a text file in the folder ‘logs’.

# The analysis of the problem

The database should contain at least three tables, for storing customer, product and order data. Customer table holds only relevant information of a customer like name, e-mail and telephone number, and each customer should be uniquely identified by an ID number. Similarly products will be characterized by a unique ID, their name, price, weight, and quantity, which refers to the available quantity of the referred product in the warehouse. The order table will realize the many-to-many relationship between these two concepts, thus an entry will consist of the customer’s ID, the product’s ID and the quantity ordered. A full order consists of retrieving all the entries with the customer’s ID, getting all ordered products by him this way.

The application will be constructed in a layered architecture manner, having a set of classes working directly on the database, business logic classes processing actions and working with these data access classes, and a presentation layer connecting the user with the database actions. All of these work with the model of the real world ‘objects’: customer, product, order constructed in accordance to the database tables.

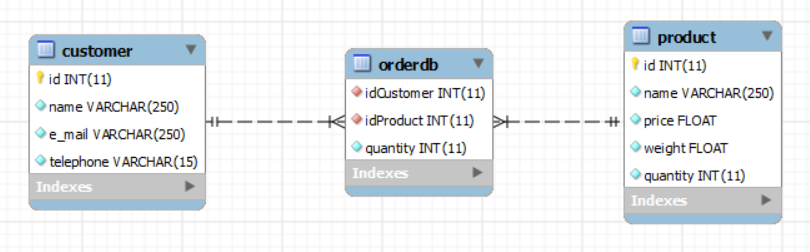
# Modeling

## The database

Contains three tables: Customer, OrderDB and Product.

Many-to-many relationship is realized between Customer and Product through table OrderDB, as a customer may order many products, and a product may be ordered by many customers.

It looks like the following:



***Fig. 1.*** *The scheme of the database: tables and the connections between them*

Note that the quantity field from Product refers to the available stock of that product in the warehouse and the quantity field from Order, to the ordered amount of product.

The script to create the database can be found in the folder of the project, and when run, if the database existed before, tables are completely recreated.

## The application

Modeled in layered architecture, using generics and reflection, the following layers can be distinguished:

* Presentation Layer - contains the classes defining the user interface, namely those in the package view and also in controller, as it refers to the control and evolution of the user interface
* Business Layer – contains the classes that encapsulate the application logic, namely packages businessLogic, using the classes from validators, as these hold the constraints for the data in the database
* Data Access Layer –contains the classes containing the queries and the database connection, package dataAccess
* Model – package model, contains classes mapped to the database table, it is not a layer, as classes from all layers use it

The application has a user friendly interface which evolves according to the actions taken by the current user.

# Use cases

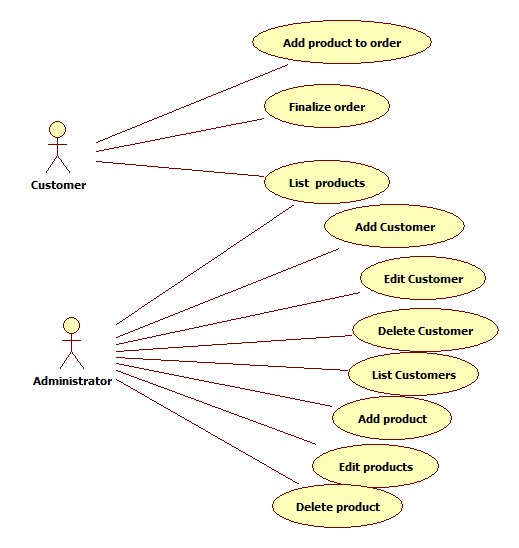
The user (actor) has to select the preferred action, insert the requested input if it is the case, and react to error messages if any.

When first opening the application, two options appear in front of the user: Customer or Administrator. The user chooses according to its role.

If Administrator was chosen, a series of available actions appear for the user, as seen in the use case diagram below. These actions each create their own specific frame which appears in front of the user once an action was chosen. For example if Add Customer was chosen, the labeled input boxes appear, which have to be filled with the corresponding data. If Edit customers was chosen for example, an editable table appears with all the customers currently in the table, and every change upon their data will be saved in the database. This submenu (Administrator) also contains a Back button which redirects to the main menu.

If Customer submenu was chosen, a list appears with all the registered customers and the user has to choose the customer who places the order. After this, the list with the available products appear and a frame with the ongoing order as well. By clicking on a product, a frame appears with its data and an input box in which the user has to input the wanted amount. If in this frame, the “Add to order” button was pressed, the stock is immediately decremented and the quantity of requested product is added to the order, as it can also be observed in the order frame. In the order frame, pressing the “Order” button finalizes the order, displays the total price and generates the bill.

***Fig. 2.*** *The list of packages in the application and the classes contained in them*



## Scenarios

Precondition: The user successfully launches the application.

Success:

1. Main menu appears, two buttons can be pressed.
2. User presses the ‘Customer’ button.
3. A list with the registered customers appears.
4. User chooses a customer to place the order.
5. A list with the available products and a window with the ongoing order appears.
6. User clicks on a product.
7. A window with the data on the selected product appears and an input box for the desired quantity.
8. User inputs the desired quantity for the product.
9. User presses “Add to order” button.
10. Ongoing order frame is updated, selected product add window is closed.
11. User can return to step 6 or execute the next step.
12. User presses the “Order” button in the ongoing order frame.
13. Total price appears under the ordered products, a bill text file is generated in the logs folder of the application.
14. User closes the application.

Alternative scenarios:

**First:** User clicks on an “Administrator” button in the main menu.

1. Submenu appears with the available actions for the administrator.
2. User selects desired action.
3. User completes desired action.
4. User clicks on “Back” button.
5. Return to step 1.

**Second:** User clicks on an empty field in the customers list.

1. Error message displayed on a separate pop-up window that informs the user about the invalid input.
2. User closes error window.
3. Return to step 4.

**Third:** User clicks on an empty field in the products list.

1. Error message displayed on a separate pop-up window that informs the user about the invalid input.
2. User closes error window.
3. Return to step 6.

**Fourth:** User enters invalid input for desired product quantity.

1. Error message displayed on a separate pop-up window that informs the user about the invalid input.
2. User closes error window.
3. Return to step 8.

# Design

## Design concepts

As mentioned before, design is based on a layered architecture model, having at least four layers represented as packages in the parentheses:

|  |
| --- |
| * dataAccessLayer (dataAccess) * businessLayer (businessLogic, validators) * model(model) * presentation (view, controller) |

Besides the classes sorted into these packages, the application will also have a Main class in the default package which contains only the start instruction of the application in its main function.

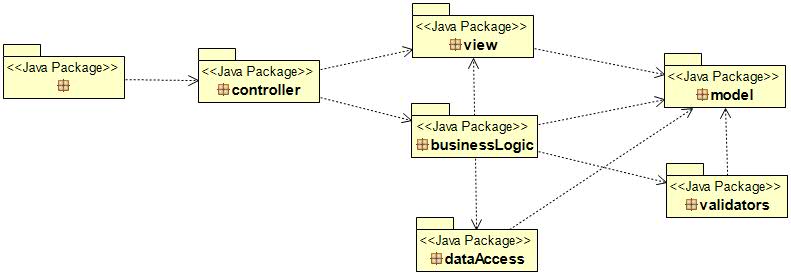
In the package model, during the design process, the class which models the orders was named OrderDB to avoid misunderstandings with the SQL statement ORDER BY.

Package validators contains the interface Validator and the classes which implement it. These classes are used to verify the correctness of the data which is to be inserted in the database. We can refer to them as constraints applied to the data.

The package view contains numerous classes, as the application uses many pop-up windows, which all are implemented in separate classes.

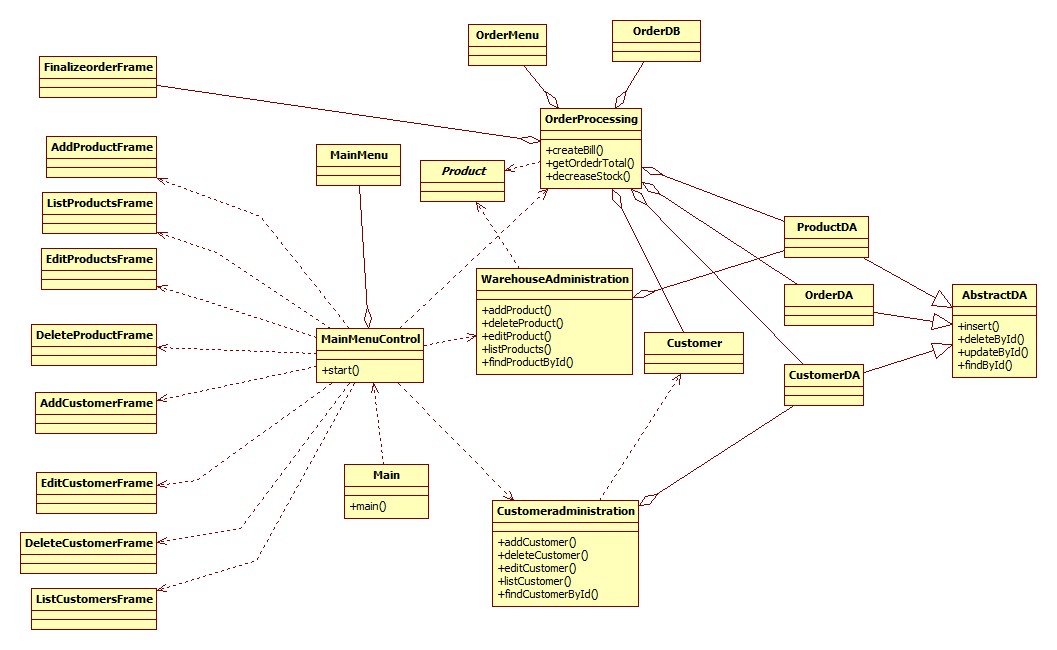
Generics are used when implementing the data access classes, as in AbstractDA class, all the CRUD operations are implemented using a generic object type T, and this class is extended by classes CustomerDA, OrderDA and ProductDA. In order to be able to implement generic methods for working with the information in the database, reflection techniques are used. The connection to the database is realized through the static methods in class Database.

## Package diagram



## Class diagram

Summarized class diagram (some non-relevant classes, and some getter, setter and helper methods were omitted for comprehension):



### The Model

#### Class Customer

Models relevant information on a single customer: his ID, his name, e-mail and telephone number. The class only contains getters and setters for these properties, as well as various constructors. Its fields are in accordance to the table in the database.

#### Class Product

Models relevant information on a product: unique ID, name, price, weight and quantity. The class only contains getters and setters for these properties, as well as various constructors. Its fields are in accordance to the table in the database.

#### Class OrderDB

Models relevant information on an order entry from the database: a customer ID, the product ID, quantity. The class only contains getters and setters for these properties, as well as a constructor.

### The Data Access

#### Class Database

Realizes the connection with the database created in MySQL. In order to do this, a JDBC driver was added to the project and connection strings were defined in this class:

**private** **static** **final** String ***DRIVER*** = "com.mysql.jdbc.Driver";

**private** **static** **final** String ***DBURL*** = "jdbc:mysql://localhost:3306/warehouse?autoReconnect=true&useSSL=false";

To realize correctly and efficiently the connection, a singleton object was created within this class, as a static member, because only one instance of this object is needed to be used across the whole system.

Methods implemented in this class are: connect, and getConnection which do as their name suggests, and the close methods for the ResultSet, the Statement and Connection objects as well. The lastly mentioned classes are instantiated in the data access classes and used to execute actions upon the database.

#### Class AbstractDA

We can call this class as the “brain” of the application. It implements the generic operations for acting upon the database. Thus, methods for insert, delete, update and retrieve data are implemented.

In order to exemplify the generic implementation, let’s take the implementation of the findById method as an example.



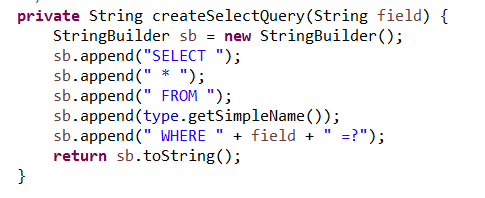
***Fig. 4.*** *The implementation of the method findById from class AbstractDA, package dataAccess*

As we can see, the method takes an integer parameter id, which corresponds to the id of the object to be retrieved. In the method, the Connection, Statement and ResultSet objects are created. These are used to create the connection with the database, to execute the query, and to get the result of the executed query respectively.

The line in the red bracket calls the function

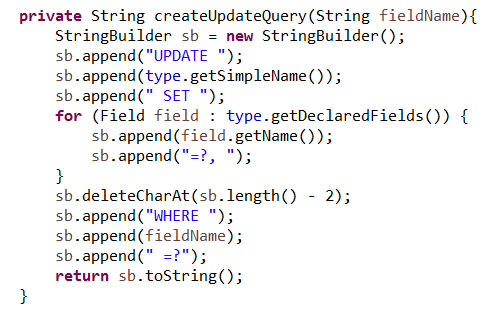
**private** String createUpdateQuery(String fieldName);

also implemented in this AbstractDA class. This creates a SQL query using the command SELECT, but taking as parameter the field based upon which the selection is made. It looks like the following:



***Fig. 5.*** *The implementation of the method createSelectQuery from class AbstractDA, package dataAccess*

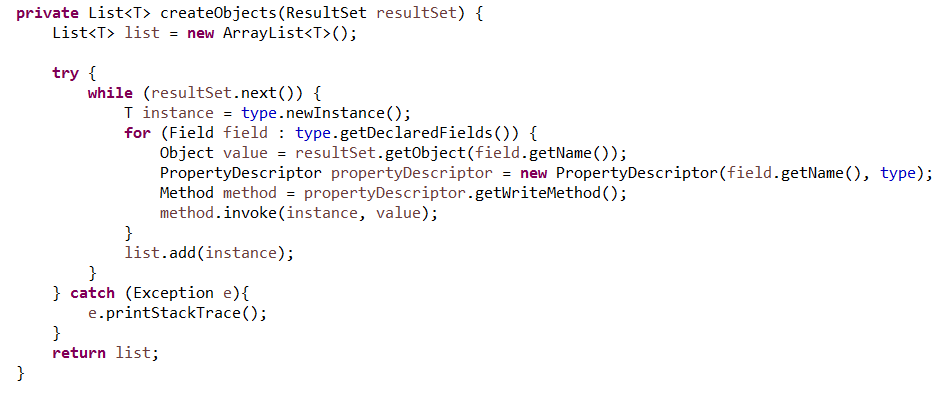
Similarly, all required methods to create the SQL statements are implemented, for example the one for the UPDATE query, in which reflection techniques are used to create a statement specific to the table with which we are working:



***Fig. 6.*** *The implementation of the method createUpdateQuery from class AbstractDA, package dataAccess*

The instructions in the bracket get the fields of the type with which we are working with, generically, thus allowing us to use this method for the various classes and models inheriting it.

Returning to *Fig. 4.* , the instructions in the try block execute the query upon the database, and while the result is stored in the respective ResultSet, the method mentioned in the line in the orange bracket (also implemented here), creates the objects based on this result set. For the findById method only the very first (and single) object of the resulted list is retrieved.

Method createObjects looks like the following: 

***Fig. 7.*** *The implementation of the method createObjects from class AbstractDA, package dataAccess*

It uses PropertyDescriptor objects to retrieve setter methods from the generic type T, and Method objects to invoke it.

The techniques presented above apply for the implementation of the remaining methods in this AbstractDA class, hence realizing a generic superclass whose methods can be used only by specifying the type of object we work with.

#### Classes CustomerDA, OrderDA, ProductDA

They extend class AbstractDA and specify their constructor the type of object they work with from the model.

### The Business Logic

#### Package businessLogic

Contains classes **CustomerAdministration**, **OrderProcessing**, and **WarehouseAdministration**, all based on the same principle: implementing the logic behind the user actions on the database. They are all implemented in a singleton manner, as we need only one instance of them in the system.

They contain instances of the data access classes relative to the type they work with, as well as instances of the user interface frames in which actions are performed. They contain inner classes implementing the logic actions regarding the dynamic elements from the user interface and define what happens next.

Let’s take as an example the action of deleting a customer based on their ID.

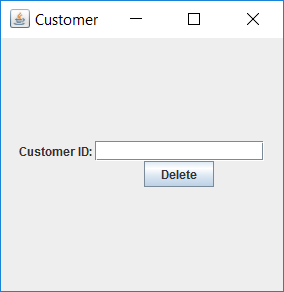
When the button for deleting a customer was pressed (the action response is controlled by the MainMenuController, described later), the static function from class CustomerAdministration corresponding to the deletion of a customer is called:

**public** **static** **void** deleteCustomer(DeleteCustomerFrame f) {

*dcf* = f;

*dcf*.addDeleteListener(*instance*.**new** AddNewDelete());

}

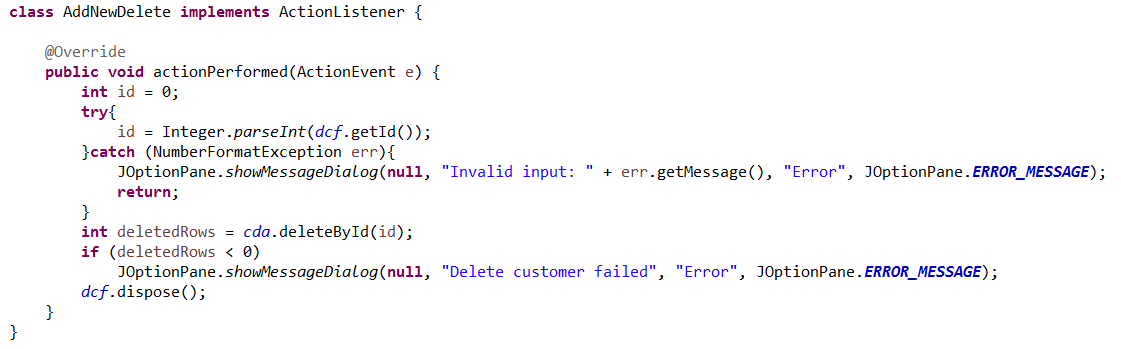
The parameter is the frame from the user interface which was created for the user (Fig. 8.), waiting for the input of the ID of the customer to be deleted. The listener for the button corresponds to the inner class AddNewDelete, implemented within this class.

***Fig. 8.*** *Frame for deletion of a customer*.

AddNewDelete implements the interface ActionListener and its method, actionPerformed. Within this method, user input data is read and parsed.

Throughout the whole application, whenever bad data is input, an pop-up error message dialog appears, and informs the user about the error.

If good input was obtained, as we can see from the line in brackets from the next figure, the deleteById method is called from the data access classes, where cda is an instance of CustomerDA class.



***Fig. 9.*** *The implementation of the inner class AddNewDelete from within class CustomerAdministration*

Following these principles and applying any necessary actions, the business logic of the system is implemented in this way.

#### Package validators

Contains the interface **Validator** and the two classes implementing it: **EmailValidator** and **TelephoneValidator**. These are instantiated in the CustomerAdministration business logic class and applied as constraints whenever new or modified inputs are to be inserted in the database.

### The Presentation

#### Package view

Contains 12 classes extending JFrame, each specific to a wintow in the user interface. Class MainMenu draws the initial window of the application, and, eventually if Administrator submenu was selected, the buttons for the administrative actions. Classes having as name Add/Edit/Delete/List + Customer/Product + Frame contain the implementation of windows specific to the actions.

All the classes within this packages contain constructors (eventually methods) for placing the graphical elements in the interface and if needed, action listener methods are implemented upon them.

##### Class OrderMenu

I would like to mention this class explicitly, as it contains a generic method for creating a JTable with the list of objects given to it as an argument. It uses reflection techniques and extract through reflection the object properties and then populates the table with the values of the elements from the list.

Its signature is the following:

**public** <T> **void** setList(Class<T> type, List<T> objectList)

where T is the object, Class<T> is the class of the object, needed to extract the fields, and List<T> is an ArrayList with the objects of type T. The method does not return a JTable, but creates and displays it within the method.

#### Package controller – Class MainMenuControl

Similar to the developed control classes for an MVC model, this class controls the main menu interface, and adds listeners to the buttons, redirects actions and implements in its inner classes the actions to be performed relative to the buttons pressed. It has an instance of itself, thus creating a singleton structure. It has a static start() method which starts the whole application actually. To be able to implement the Back button, this instance of the system is recreated when that button was pressed.

### Class Main

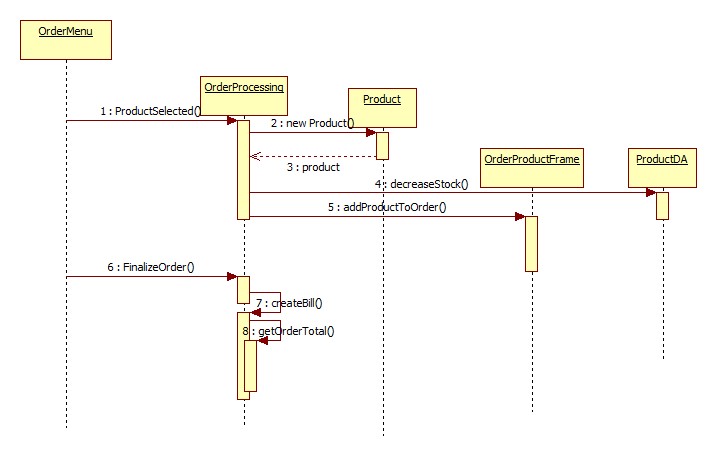
It is in the default package and contains the main method of the application, with a single line in it:

MainMenuControl.*start*();

Which instantiates the control and the user interface.

In the commented section of the main function there are some hardcoded inserts in the database, which are useful when the empty database is created or re-created and the application tester does not want to insert data one-by-one, so he just needs to decomment this section before running the application.

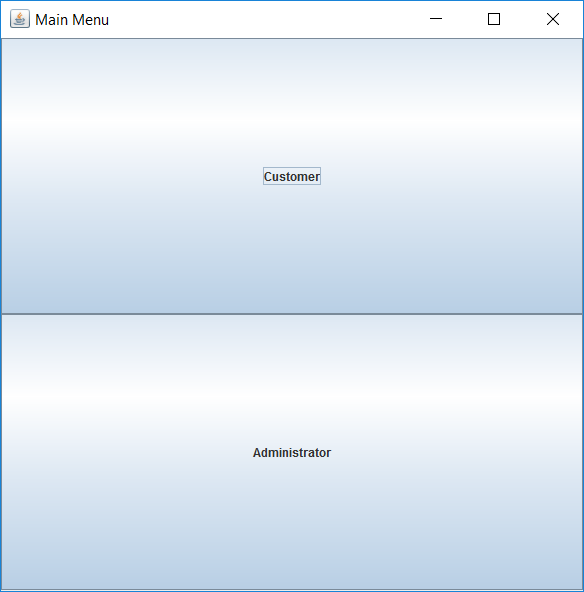
## Sequence diagram



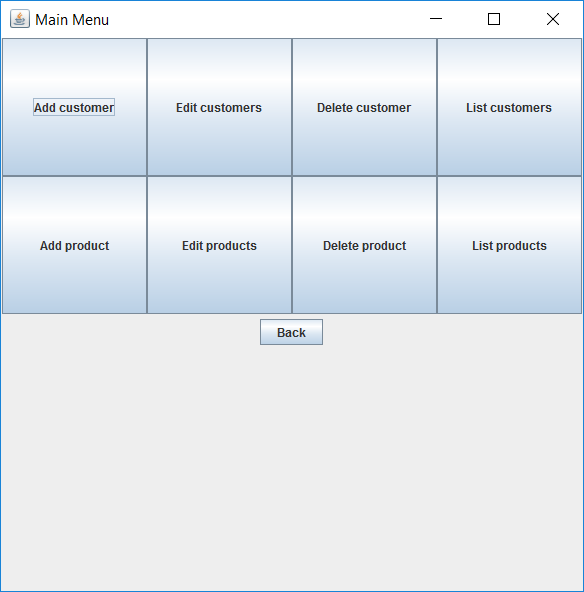
I’ve taken as example the sequence diagram for when an instance of OrderMenu was created, it creates a chain reaction and all the components of the system are instantiated accordingly.

## Graphical User Interface

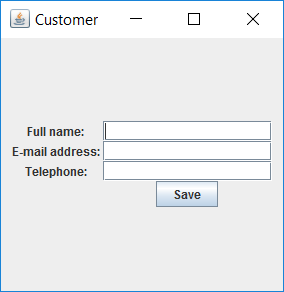
It is user friendly, the user is presented with intuitive actions to be performed. When first opened, the application shows the main menu:



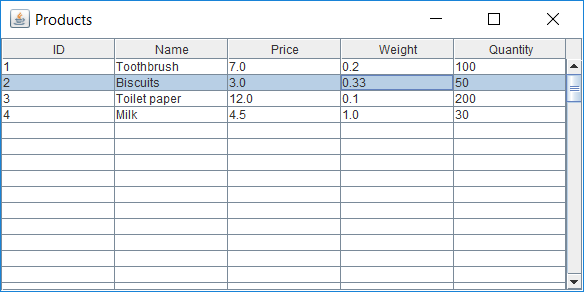
If the user chooses to perform administrative actions, when he presses the “Administrator” button, the following submenu appears:



Each of these buttons is functional and creates a separate action frame when pressed. This way, for example, pressing the “Add customer” button pops up the following window:

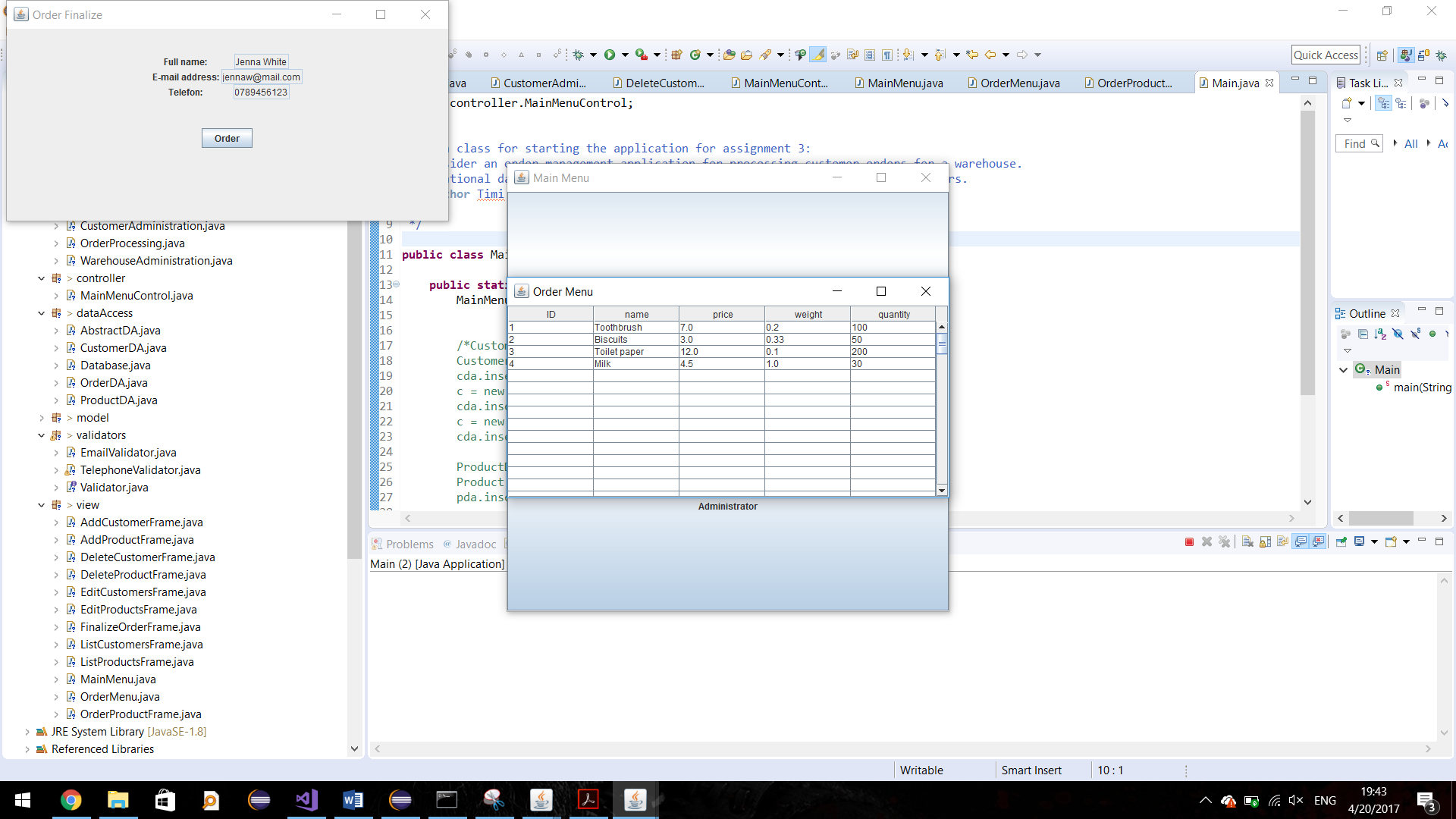


Or, pressing the “Edit products” button pops up a window with an editable JTable of the following structure:

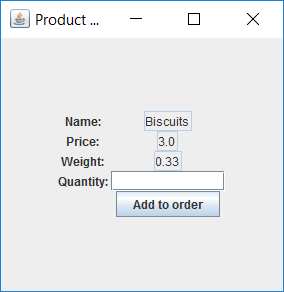


The listed products are the ones stored in the database, in the ‘warehouse’.

Pressing the “Back” button redirects to the main menu, where if the user chooses to press the Customer button, thus engaging in placing an order, a list with the registered customers pops up. There, he will have to click on one to start ordering for him. When the customer was selected, the following layout appears:



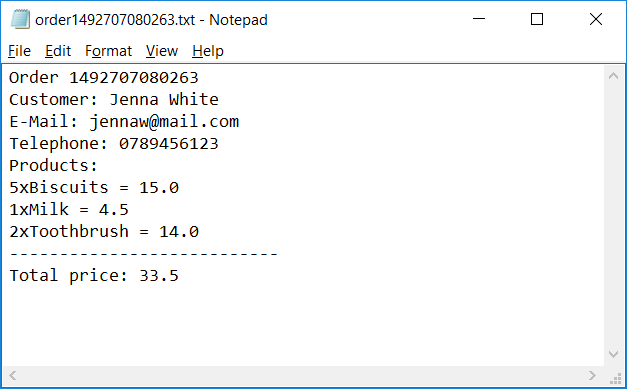
As it can be observed, in the top left corner of the screen, an ongoing order window popped up which is updated whenever a product is added to the order. To do so, the user has to select one from the list which appeared instead of the customers list, and then a window pops up with the product information and an input box for the desired quantity:



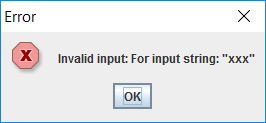
After desired quantity was introduced, the product is added to the order, stock is decremented and product list is updated. After all desired products were added to the order, on the ongoing order window, the “Order” button should be pressed to finalize it. Then the product list closes and this window gets the following layout:



In the same time, a bill according to this order is created in the logs folder of the project, which looks like the following:



Whenever an error occurs during the usage of the application, for example if bad data format was input, an error pop-up window appears which informs the user of the error. For example:



# Implementation and testing

Implementation started from modeling the problem and drawing an approximate of the class diagram, from where I first implemented the structure of the Model package. Alongside with this, I also created the database in MySQL to be able to verify my results, and to be able to build on it. The next implemented classes were Database, which realized the connection, and then the AbstractDA class, the center of the application. I can say that the application was developed layer by layer, because after this, the business logic was implemented. When it was done, I wrote the some of the user interface classes so I could build on it, display and verify my results, but along with that, the already implemented classes were constantly updated to make sure they work correctly.

During implementation I have verified the correctness of my code by analyzing the log events that I have displayed in the console. Whenever an unexpected error occurred, I have used Eclipse’s debugger to find the source of the problem and correct it.

Building the user interface took a lot of time and hard work, because all the actions had to be connected and implemented relatively to each other.

# Results

The development of this assignment resulted in a decent, user friendly application that illustrates in a professional manner the action which can be taken upon a database. The developed system of warehouse management shows in an effective way how data can be stored, updated, retrieved and deleted in a persistent manner. This means, that data is not recreated whenever the application is restarted. Also, the warehouse system is widely used nowadays, as web shopping grows, and the process of placing orders is widely known and recognized.

# Conclusions

## What have I learned?

With this assignment I have certainly learned the connection between applications and databases, how one works with the other. The techniques used in this project were also new to me, the layered architecture and the usage of generics and reflection techniques. I have also achieved knowledge about creating a more complex project, solving programming issues.

## Further developments

This application can be refined by defining each order as unique, as for now, the orders stored in the database are relative to a registered customer, not to time moment. Besides this, new facilities for couriers can be implemented for example, such as a report with the address of the customer and the package size and weight.

# Bibliography

1. For programming issues: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>
2. For more programming issues: <http://stackoverflow.com/>
3. Class diagram description: <https://en.wikipedia.org/wiki/Class_diagram>
4. Last semester’s OOP code on how to use Swing to create user interface