Technical University of Cluj-Napoca

Programming Techniques

Laboratory – Assignment 2

Database Management

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Laboratory Assistant:
Ing. Ciprian Stan

Student:
Enoiu Diana-Cristina

1. Assignment 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 (for example Order, Client, Product)
· 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.
a. Analyze the application domain, determine the structure and behavior of its classes and draw an extended UML class diagram.
b. Implement the application classes. Use javadoc for documenting classes.
c. Use reflection techniques to create a method that receives a list of objects and generates the header of the table by extracting through reflection the object properties and then populates the table with the values of the elements from the list.

2. Analysis

a) Problem analysis

This application should be able to fulfil all the requirements in order to display, modify, insert or delete clients and products and it should be able to place orders as well. The performing of these operations will be done with the help of a GUI. The data, about customers, items and placed orders, is stored in a relational MySQL database, along with the information about the users which have access to the system. This way, all the data is easier to retrieve and access from different computers.

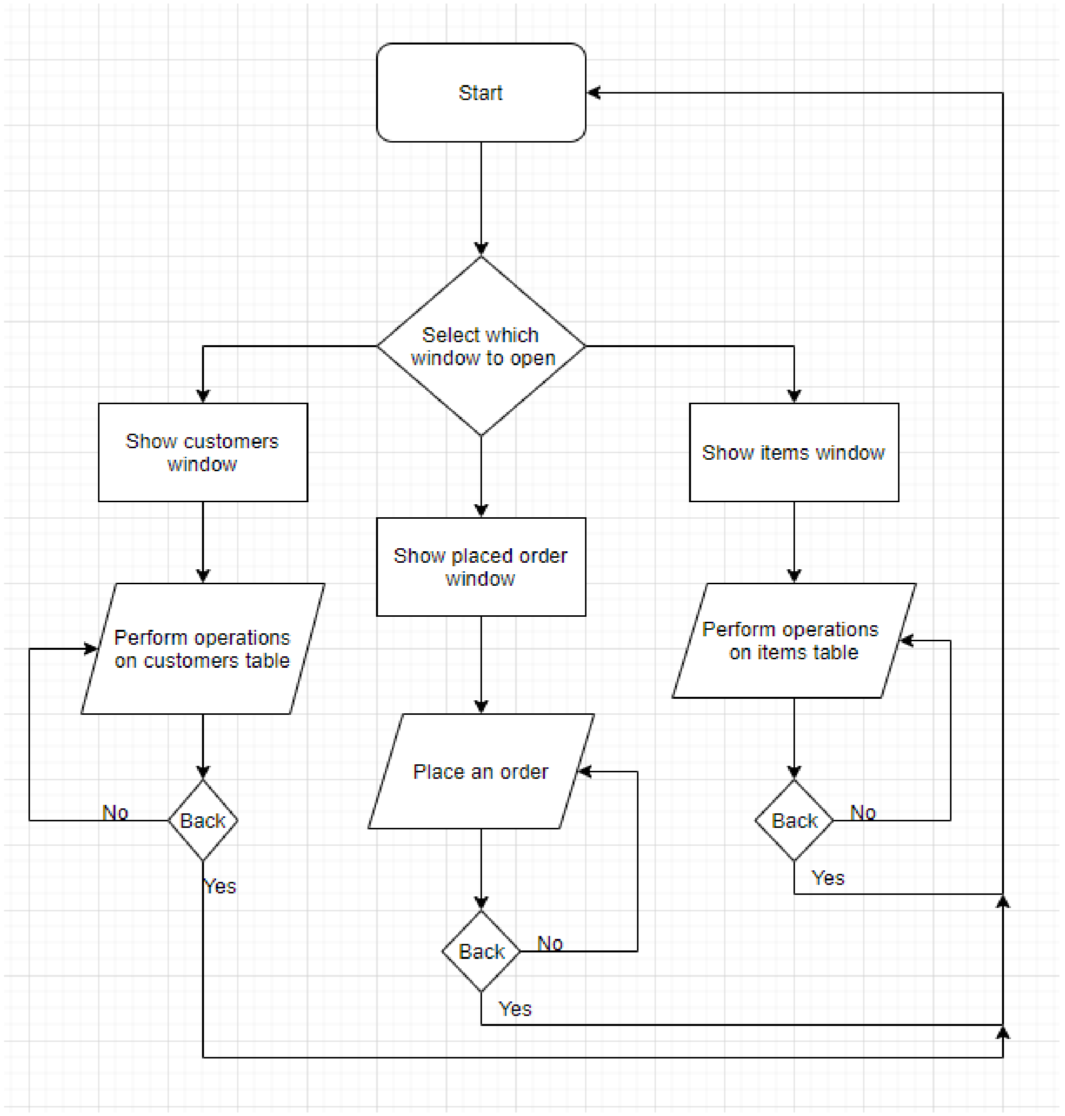
b) Modeling

All the data about the customers, the items and the placed orders will be placed in tables in a database.
The customers table will have a first name, last name, email address, phone number and an address to which the order will be delivered.
The items table will have a name, a price and a quantity that is available.
The placed orders table will have a data an customer id, an item id and a quantity that is ordered.
Further we can interact with the data from these tables by creating a connection to the database and then running specific queriers in order to modify, insert or delete data.
For example, for the Customer table the user can introduce an ID for the client (if no id is provided the application will generate an auto-incremented ID), the first name and last name of the client, the Email address, the phone number, and the address to deliver the order. All

fields have to obey the standard rules such as phone number has to have 10 digits (only digits) and the email address has to contain the @ and the . symbols .

c) *Different scenarios and use cases*

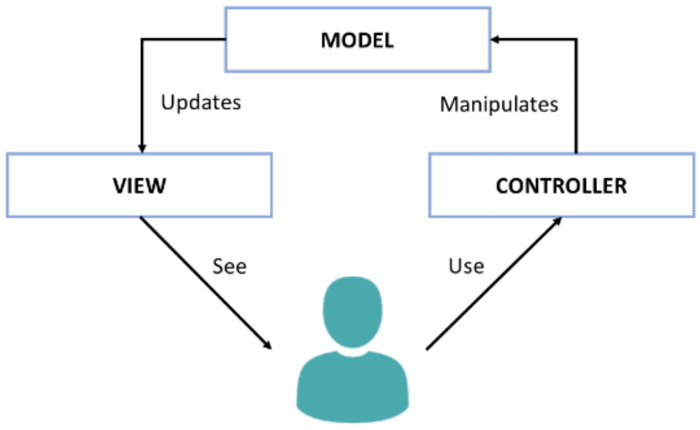
A use case is a methodology used in system analysis to identify, clarify, and organize system requirements. The use case is made up of a set of possible sequences of interactions between systems and users in a particular environment and related to a particular goal.
The use cases are corelated with the steps the user makes when using the application.
The application should have a user friendly interface in order for the user to understand faster what are the steps that are needed to be performed in order to achieve the desired result.
For this application, the flow-chart from below is describing the way the application is expected to behave, based on the user inputs.



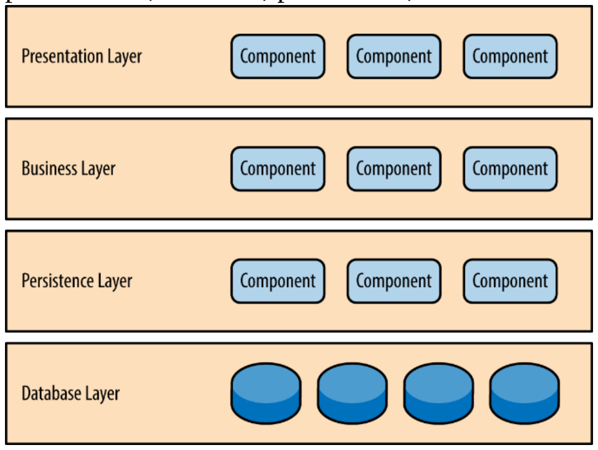
3. Design and Implementation

a) *Design decisions*

This application uses an object-oriented programming design. The application has a graphical user interface made in JavaFX, with the help of Scene Builder.
For the design of this application it was used, also, an architectural pattern, the MVC pattern, (Model-View-Controller pattern), which is used to separate application's concerns.
The Model–View–Controller is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements.

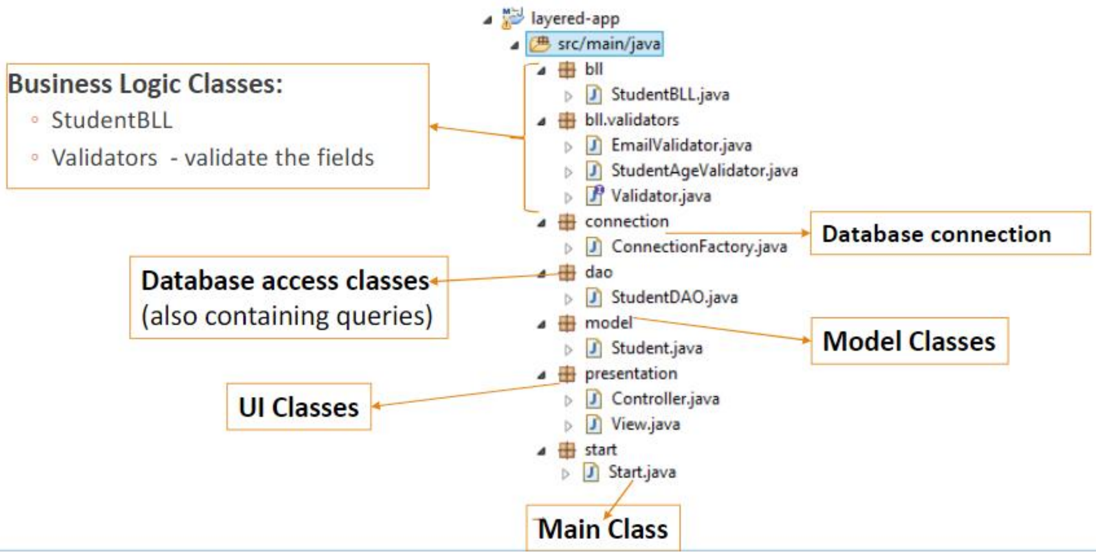


This application also uses the most common architecture pattern which is the layered architecture.
Components within the layered architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database.



Each layer of the layered architecture pattern has a specific role and responsibility within the application. For example, a presentation layer would be responsible for handling all user

interface and browser communication logic, whereas a business layer would be responsible for executing specific business rules associated with the request. Each layer in the architecture forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the presentation layer doesn't need to know or worry about how to get customer data; it only needs to display that information on a screen in particular format. Similarly, the business layer doesn't need to be concerned about how to format customer data for display on a screen or even where the customer data is coming from; it only needs to get the data from the persistence layer, perform business logic against the data (e.g., calculate values or aggregate data), and pass that information up to the presentation layer.
Example of a layered architecture:



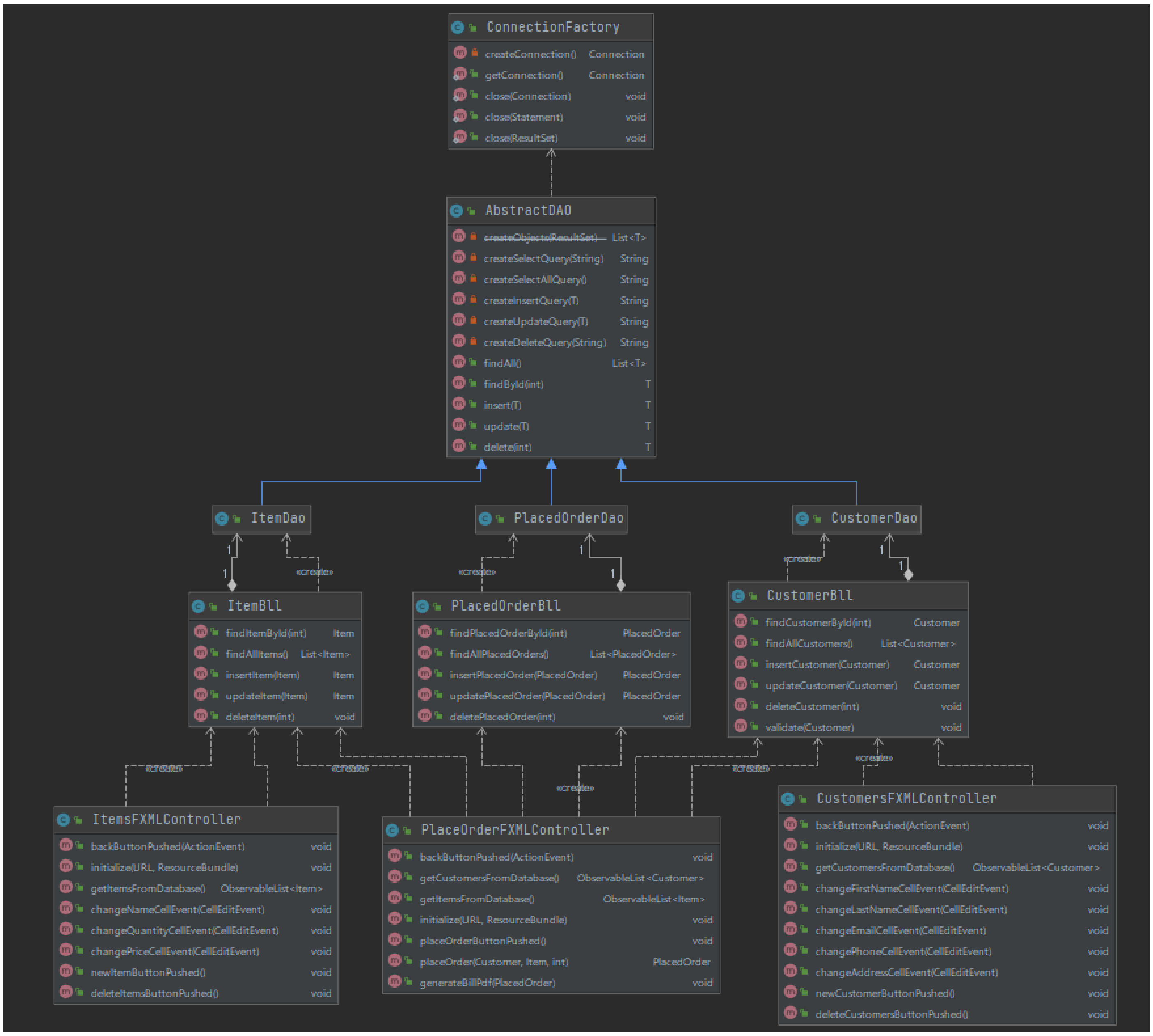
This application' classes have, thus, been split into seven main packages:

- The application package which contains the Main class;
- The model package which has the Customer, Item and PlacedOrder used in the modelling of the program;.
- The view package containing .fxml, .css files used for displaying the actual interface;
- The controller package, which manages the view and model packages, contains the HomeFXMLController class for the home.fxml, the CustomerFXMLController for the customer.fxml, the ItemFXMLController for the item.fxml and the PlacedOrderFXMLController for the placedOrder.fxml.
- The connection package responsible for creating a single instance to a connection to the database.
- The DAO (data access or persistence layer) package which will have a generic abstract class called AbstractDao which will create some generic queries and will implement the methods to execute them by establishing the connection to the database. This abstract class will be futher implemented by the CustomerDao, ItemDao and placedOrderDao.
- The Business Logic package which will encapsulate the application logic and has the CustomerBll, ItemBll and PlacedOrderBll classes that will basically call the required methods from the DAO, but without worrying about the database connection, or the specific queries that are used in order to receive the required data.

b) Data structures

For this application the data structures that were used are primitive data types (integers, doubles), non-primitive data types (Strings), and more complex data types such as ArrayLists, ObservableList(for the tableView display of data).

c) UML class diagram



ConnectionFactory
createConnection() Connection
getConnection() Connection
close(Connection) void
close(Statement) void
close(ResultSet) void
AbstractDAO
~~createObjects(ResultSet)~~ List<T>
createSelectQuery(String) String
createSelectAllQuery() String
createInsertQuery(T) String
createUpdateQuery(T) String
createDeleteQuery(String) String
findAll() List<T>
findById(int) T
insert(T) T
update(T) T
delete(int) T
ItemDao
PlacedOrderDao
CustomerDao
ItemBll
findItemById(int) Item
findAllItems() List<Item>
insertItem(Item) Item
updateItem(Item) Item
deleteItem(int) void
PlacedOrderBll
findPlacedOrderById(int) PlacedOrder
findAllPlacedOrders() List<PlacedOrder>
insertPlacedOrder(PlacedOrder) PlacedOrder
updatePlacedOrder(PlacedOrder) PlacedOrder
deletePlacedOrder(int) void
CustomerBll
findCustomerById(int) Customer
findAllCustomers() List<Customer>
insertCustomer(Customer) Customer
updateCustomer(Customer) Customer
deleteCustomer(int) void
validate(Customer) void
ItemsFXMLController
backButtonPushed(ActionEvent) void
initialize(URL, ResourceBundle) void
getItemsFromDatabase() ObservableList<Item>
changeNameCellEvent(CellEditEvent) void
changeQuantityCellEvent(CellEditEvent) void
changePriceCellEvent(CellEditEvent) void
newItemButtonPushed() void
deleteItemsButtonPushed() void
PlaceOrderFXMLController
backButtonPushed(ActionEvent) void
getCustomersFromDatabase() ObservableList<Customer>
getItemsFromDatabase() ObservableList<Item>
initialize(URL, ResourceBundle) void
placeOrderButtonPushed() void
placeOrder(Customer, Item, int) PlacedOrder
generateBillPdf(PlacedOrder) void
CustomersFXMLController
backButtonPushed(ActionEvent) void
initialize(URL, ResourceBundle) void
getCustomersFromDatabase() ObservableList<Customer>
changeFirstNameCellEvent(CellEditEvent) void
changeLastNameCellEvent(CellEditEvent) void
changeEmailCellEvent(CellEditEvent) void
changePhoneCellEvent(CellEditEvent) void
changeAddressCellEvent(CellEditEvent) void
newCustomerButtonPushed() void
deleteCustomersButtonPushed() void
«create»
1

d) *Class design*

Because of the MVC and Layered architecture, this program consists of 7 parts:

1) **The Model** – contains the logic of the application

Customer Class
Used to simulate the customer object. This class has the exact same fields as they are in the database.
Constructors:
- *public Customer(int id, String firstName, String lastName, String email, String phone, String address)* //the constructor that initializes the customer with its corresponding firstName, lastName, email, phone, address

Methods:
- *just some getters and setters.*

Item Class
Used to simulate the item object. This class has the exact same fields as they are in the database.
Constructor:
- *public Item(int id, String name, int price, int quantity) {*// the constructor that initializes the item with its corresponding name, quantity and price

Methods:
- *just some getters and setters.*

PlacedOrder Class
Used to simulate the placed order object. This class has the exact same fields as they are in the database.
Constructor:
- *public PlacedOrder(int id, Date date, int customerId, int itemId, int quantity, int totalPrice)* // the constructor that initializes the place order with its corresponding date, cutomerId, itemId, quantity and totalPrice

Methods:
- *just some getters and setters.*

2) **The View**– contains the graphical interface, the user interface.

This package contains the .fxml, .css files used in the displaying and the designing of the graphical user interface.
For this application, Scene Builder was used in order to be able to develop much faster an user-friendly interface, and with a nice-looking design without having advanced knowledge with .fxml files.
This package contains the home.fxml the scene where the user will select which window to display, and also the customer.fxml, item.fxml and placedOrder.fxml the scenes where the user can see the tables from the database and perform operations like insert, delete and modify.

3) **The Control**–manages the model and the view packages

This is a very important package because it acts on both model and view. It controls the data flow into model object and updates the view whenever data changes.

HomeFXMLController class – controls the home.fxml
The controller responsible for reacting when the user chooses an window and display the corresponding window.
Methods:
- *public void buttonPushed(ActionEvent event) throws IOException*

//receives an request from the user when the mouse click on theone of the three buttons corresponding to the three windows

CustomerFXMLController class – controls the customer.fxml
This controller is responsible for controlling the customer window (which holds the customer table from the database in a tableView) by receiving the users actions and act accordingly to them.

Methods:
- *public void initialize(URL url, ResourceBundle rb)*

//sets the tableView with the corresponding data
- *public ObservableList<Customer> getCustomersFromDatabase());*

// method to get all customers from the database
- *public void changeFirstNameCellEvent(CellEditEvent editedCell);*
- *public void changeLastNameCellEvent(CellEditEvent editedCell);*
- *public void changeEmailCellEvent(CellEditEvent editedCell);*
- *public void changePhoneCellEvent(CellEditEvent editedCell);*
- *public void changeAddressCellEvent(CellEditEvent editedCell)*

//the methods from above modify the tableView cell with the new data entered by the user, sometimes they have to use validators to check if the input is correct, otherwise it will display an incorrect window popup
- *public void newCustomerButtonPushed();*

// This method will insert a new customer in the database, when the add new customer button is pressed
- *public void deleteCustomerButtonPushed();*

// This method will delete a customer or customers in the database, when the delete customer button is pressed
- *public void backButtonPushed(ActionEvent event) throws IOException*

// method to come back to home window

ItemFXMLController class – controls the item.fxml
This controller is responsible for controlling the item window (which holds the item table from the database in a tableView) by receiving the users actions and act accordingly to them.

Methods:
- *public void initialize(URL url, ResourceBundle rb)*
//sets the tableView with the corresponding data
- *public ObservableList*< Item > *getItemFromDatabase());*
// method to get all items from the database
- *public void changeNameCellEvent(CellEditEvent editedCell);*
- *public void changeQuantityCellEvent(CellEditEvent editedCell);*
- *public void changePriceCellEvent(CellEditEvent editedCell);*
//the methods from above modify the tableView cell with the new data entered by the user
- *public void newItemButtonPushed();*
// This method will insert a new item in the database, when the add new item button is pressed
- *public void deleteItemButtonPushed();*
// This method will delete a item or items in the database, when the delete item button is pressed
- *public void backButtonPushed(ActionEvent event) throws IOException*
// method to come back to home window

PlacedOrderFXMLController class – controls the placedOrder.fxml
This controller is responsible for controlling the placedOrder window (which holds the placedOrder table from the database in a tableView) by receiving the users actions and act accordingly to them.
Methods:

- *public void backButtonPushed(ActionEvent event) throws IOException*
// method to come back to home window
- *public ObservableList<Customer> getCustomersFromDatabase());*
// method to get all customers from the database
- *public ObservableList*< Item > *getItemFromDatabase());*
// method to get all items from the database
- *public void initialize(URL url, ResourceBundle rb)*
//sets the choice boxes with the corresponding data
- *public void placeOrderButtonPushed()*
//This method is called when place order button is pushed
-*public PlacedOrder placeOrder(Customer selectedCustomer, Item selectedItem, int insertedQuantity)*
//This method will place the order in the database
-*public void generateBillPdf(PlacedOrder placedOrder) throws FileNotFoundException, DocumentException*
//This method will generate a bill in a pdf format

4) **The Application** – contains the Main class which runs the main() method in order to "turn on" the application, and also set the necessary stage and scene for the Graphical-User Interface.

5) **The Data Access Layer -** contains the classes containing the queries and the database connection

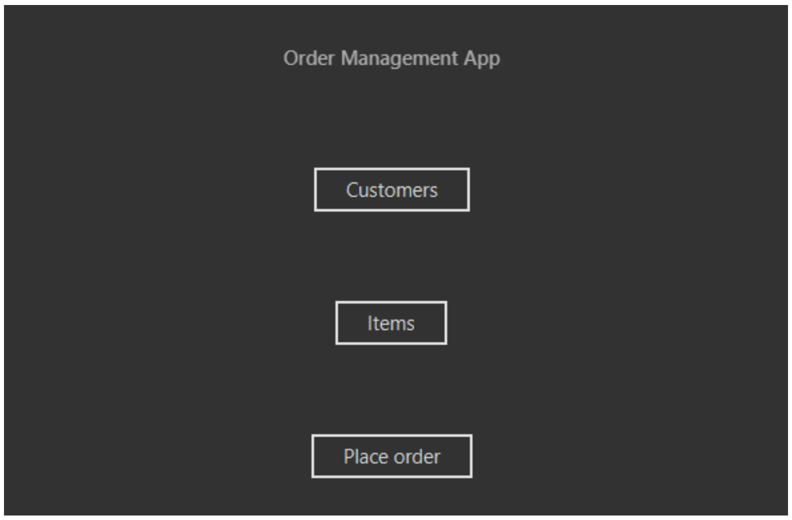
   AbstractDao class – generic abstract class responsible for generating the queries and executing them with methods like: insert, update, delete. The CustomerDao, ItemDao and PlacedOrderDao just implement the class.

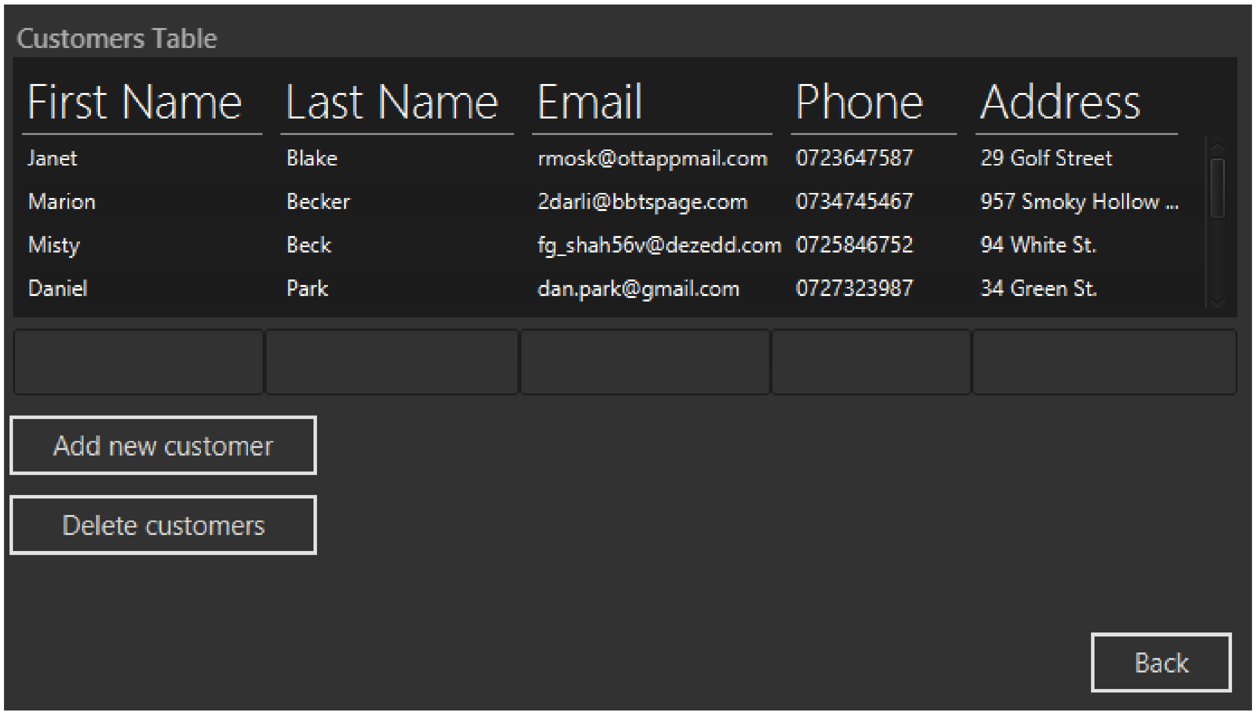
6) **The Business Logic Layer**
   The CustomerBll, ItemBll and PlacedOrderBll classes are used to implement the methods the required to communicate to the database (using the dao classes), methods like: findAllCustomers, insertItem, deletePlacedOrder etc.

7) **The Connection**
   The Connection Factory class – for generating a single instance of a connection to the database

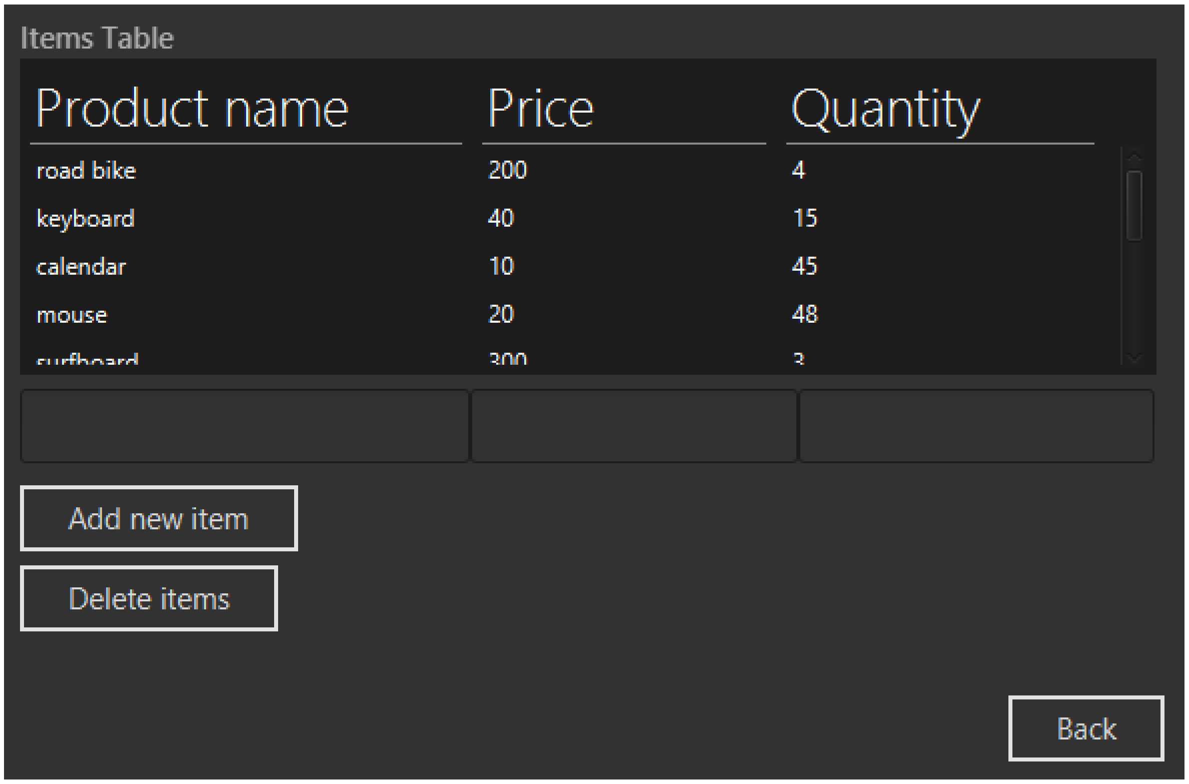
e) Packages:
App
Bll
Connection
DAO
Model
View
Controller

f) User interface

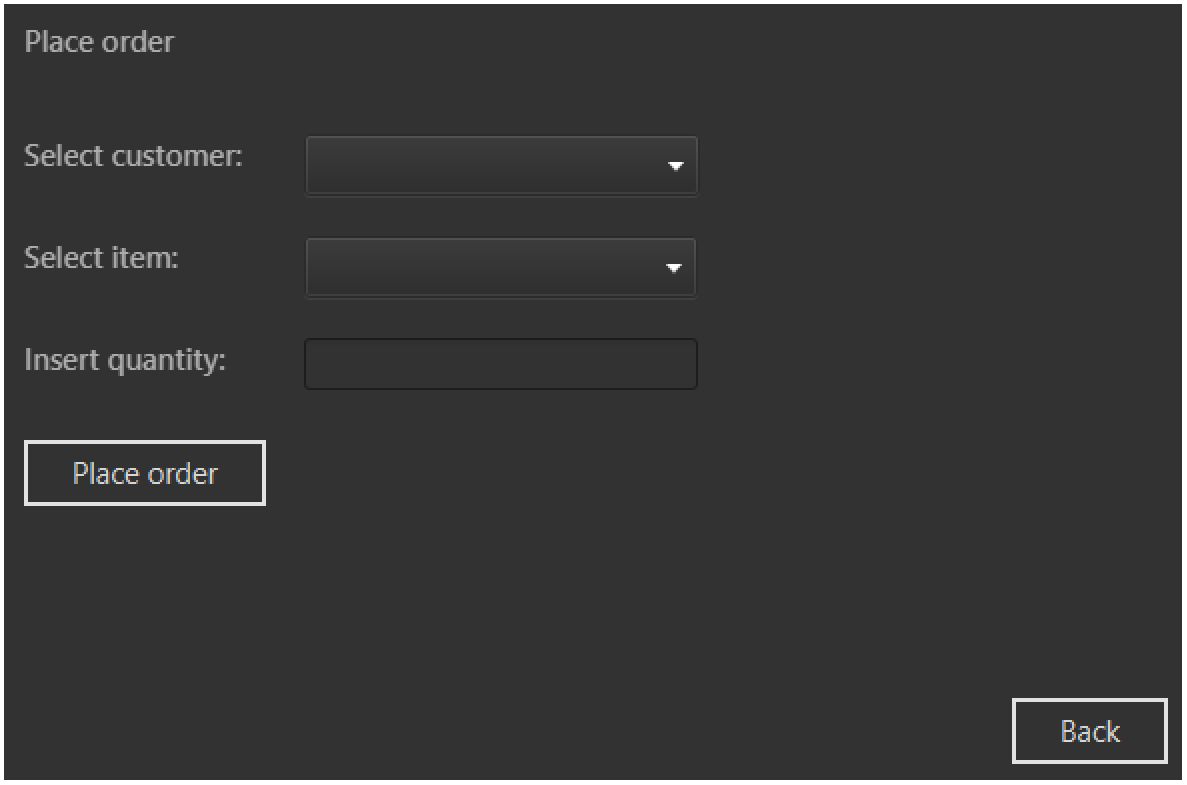




First Name	Last Name	Email	Phone	Address
Janet	Blake	rmosk@ottappmail.com	0723647587	29 Golf Street
Marion	Becker	2darli@bbtspage.com	0734745467	957 Smoky Hollow ...
Misty	Beck	fg_shah56v@dezedd.com	0725846752	94 White St.
Daniel	Park	dan.park@gmail.com	0727323987	34 Green St.



Product name	Price	Quantity
road bike	200	4
keyboard	40	15
calendar	10	45
mouse	20	48
surfboard	300	3



4. Conclusions

This assignment helped me understand better how an connection between an application and a database is created and how we can use this to communicate back and forth between those two. It also helped me familiarize with the use of generic classes and Reflection Techniques so that we can apply the same code for multiple different concrete situations.
This assignment was an introduction in building a project with Gradle, and it helped me familiarize with the building of a user interface with JavaFX.
The using of the MVC architectural was also important for me to understand because it is the basic structure which most web applications, mobile apps and desktop programs are built on.
The layered architecture was also used together with the MVC pattern because this will allow us to add new features, or change the current features more easily. Adding a new use case to the system, or extend the business rules on a particular domain object would have been much harder if the process or business logic is spread throughout the code.

5. Bibliography

http://stackoverflow.com
https://openjfx.io/openjfx-docs/#gradle
https://ro.wikipedia.org/wiki/Model-view-controller
https://app.diagrams.net/
https://en.wikipedia.org/wiki/Strategy_pattern
Programming Techniques – Lectures of prof. Ioan SALOMIE