Documentation

Order Management application

Homework number: 2

Due date: week 6

Contents

[1. Purpose 1](#_Toc447539487)

[2. Problem analysis 2](#_Toc447539488)

[a. Modelling 2](#_Toc447539489)

[b. Scenarios and Usage 2](#_Toc447539490)

[3. Design 3](#_Toc447539491)

[a. User interface 3](#_Toc447539492)

[b. Data structures 4](#_Toc447539493)

[c. Layers and packages 4](#_Toc447539494)

[d. Class design 5](#_Toc447539495)

[Presentation layer 6](#_Toc447539496)

[Business Logic layer 8](#_Toc447539497)

[Data Access layer 8](#_Toc447539498)

[Other utility classes 9](#_Toc447539499)

[4. Implementation and testing 9](#_Toc447539500)

[5. Results 10](#_Toc447539501)

[6. Conclusions 10](#_Toc447539502)

[7. Bibliography 10](#_Toc447539503)

# Purpose

The purpose of this assignment is to analyze a real-life problem and provide a solution to it by efficiently designing and implementing an application in object-oriented programming style. We are required to implement an application for handling order management.

# Problem analysis

<Modelling, scenarios, usage>

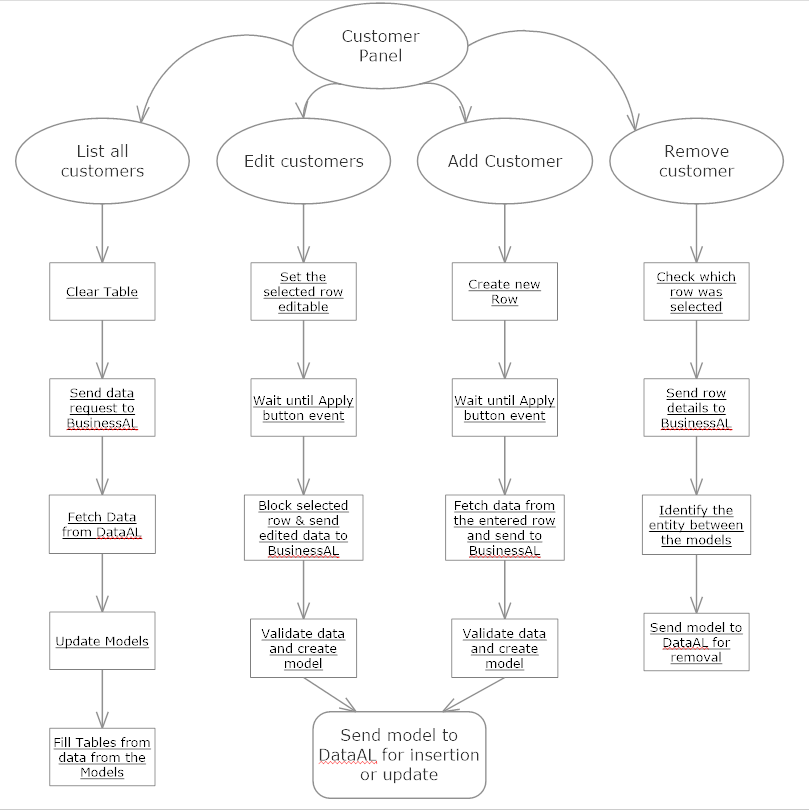
## Modelling

The application is designed for companies which sell products to different clients. We have to store a large variety of products, including their name, description and location. We also need to know the quantity of a given product. The list of products has to be easily editable. We also have to store all the necessary details of the customers: an ID for each customer, address, name and other required fields. Customers also have to be editable and we have to make sure that a customer has only one account. We also have to visualize all this data, thus the minimum requirements for the application are storing and visualizing customers and products.

To make the company more productive, we have to allow the processing of orders. For each customer we should be able to place orders, consisting of an order ID and a list of products (including the quantity). A customer shall have many orders, provided that each order has different order ID.

## Scenarios and Usage

Supposing that we already have graphical support to display the stored customers, products and orders, we have to be able to manipulate the given data. The Use Case diagram of manipulating the content of users (customers) is presented below:



We have four possible buttons: List all customers, Edit customer, Add customer and Remove customer. The first button will completely update the table with contents from the database, however the other buttons require further user interaction: the user has to select and/or edit a row which will be manipulated only after the Apply button is pressed. Editing Products and Orders is also similar, so I won’t discuss it in detail.

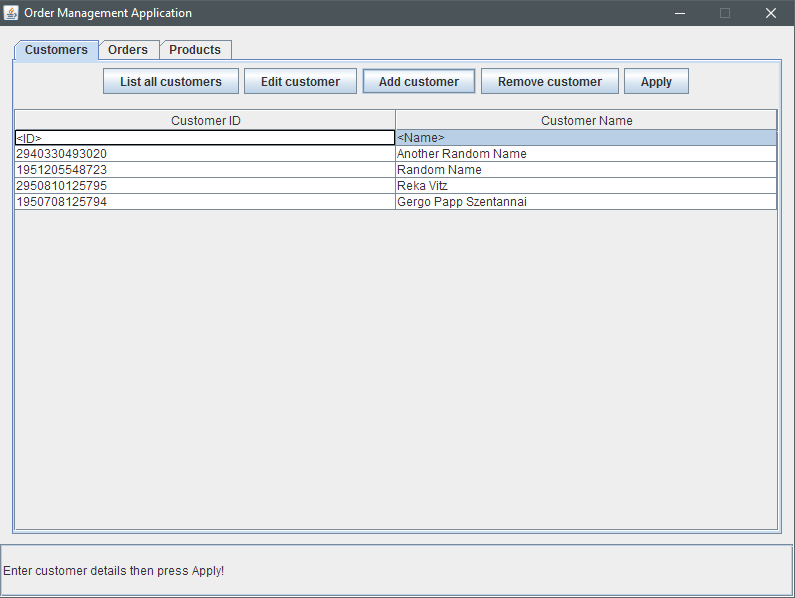
# Design

<UML diagrams, data structures, class design, interfaces, relationships, packages, algorithms, user interface>

## User interface

The user interface consists of a Main Frame, which contains a Panel, called Main Panel, having as layout manager the *Borderlayout* class. The center of the panel is a *TabbedPanel*, called *InternalPanel*, having three tabs. Each tab contains buttons and a *JTable* for displaying and editing information about customers, orders or products. At the bottom we can notice a text field, which serves as a status bar – it displays important messages to the user.

A screenshot of the GUI is presented below (the current tab shows the customers; a customer is currently being inserted):



## Data structures

To store the required data structures, I instantiated a Model class, which stores all the non-volatile information (customers, products, etc., which have to be manipulated and saved for later re-use). These entities are stored in Sets – the reason for this is that we restrict internally the apparition of multiple elements.

Each entity has its own class – these classes are: *Customer*, *Order*, *Product* and other classes that link them, together. When the application starts all the Sets are initialized but they have zero elements. When creating such entities, we have to keep in mind that we will need to save them. For this reason, each of the model classes are serializable and they also contain annotations which define tables and keys in a database.

There are two methods to save and retrieve data: the first one is local, by means of Serialization. Each model entity implements the Serializable Interface and defines a unique ID. These classes can be retrieved at the beginning and we can also save them at the end of a session. The other method is more complex and stores all the required information in a MySQL database (other database types are also supported). Each model class represent a Table in the database. This method has a much higher usability but requires a connection to a database (local server or internet connection).

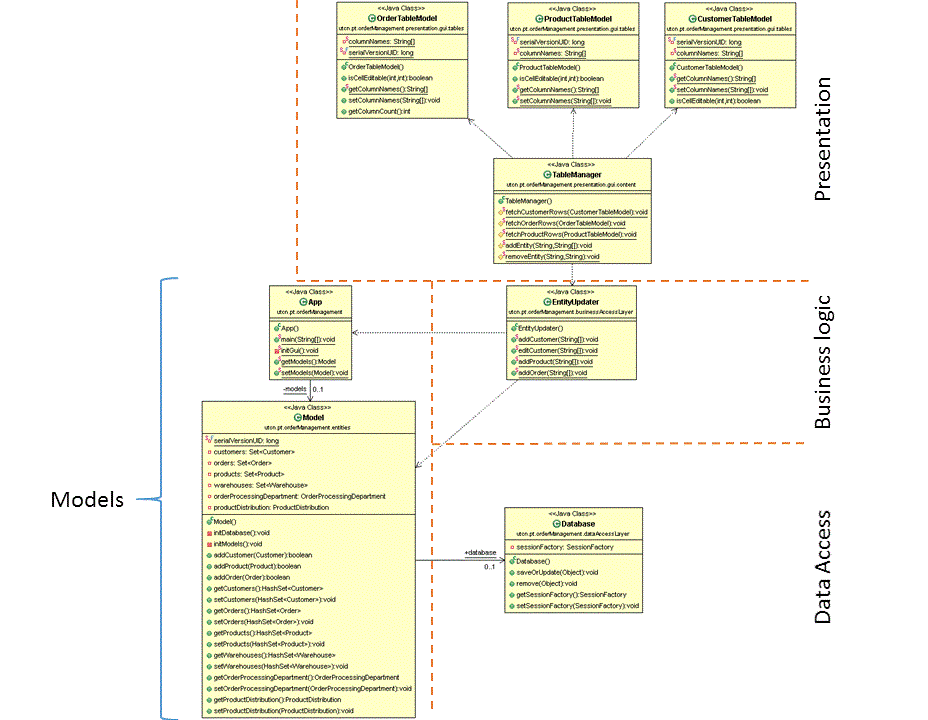
## Layers and packages

The packages are split in two categories:

* Layers
* Entities (models)

There are three vertical layers, the highest one is the Presentation (or GUI), the middle is the Business Logic Layer, and at the bottom we have the Data Access Layer.

The structure and interconnection of the layers, as well as the usage of the models can be seen in the following scheme (Please note, that this is only a sketch – only a few classes are presented, just to show the interconnection of the layers!):



As you can notice, the GUI layer cannot directly access the Data access layer (and vice-versa). To update a table or enter a new entity, first we have to access the Business logic layer. This layer then manipulates the entities and sends the required commands to the Data access layer, which manipulates the storage area. In my implementation I used the Hibernate environment to handle all the necessary SQL operations. All hibernate-related operations are done within the Database class, which manages the connection to the actual database and acts as an interface for Hibernate. A more detailed presentation of each layer is presented in the next part.

## Class design

When designing the classes, I kept in mind that each layer should be like a stand-alone application – layers should have as few references with each other as possible. Each layer has a well-defined region of interest.

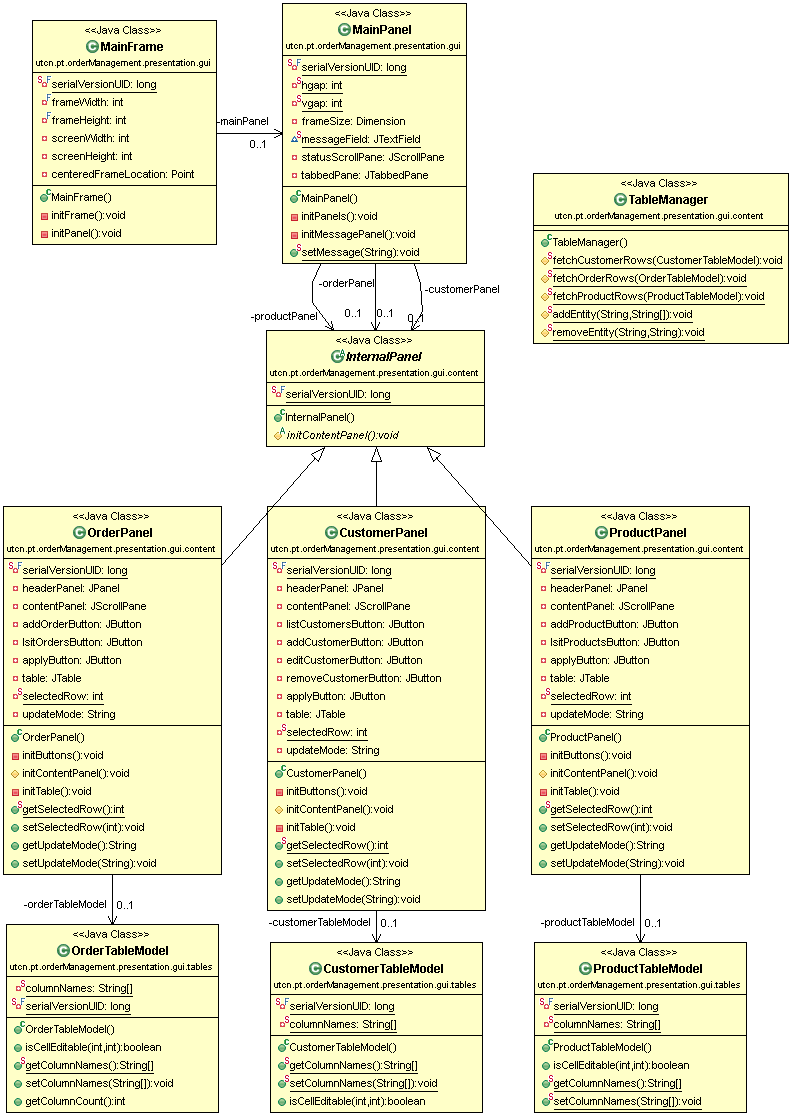
For example, the Presentation handles everything related to the GUI, including the event-handling of the buttons and managing the tables. However, the content of the tables is never determined in the Presentation layer. The only exception is when we insert or edit a new row – in this case the Presentation temporarily stores the field values. When choosing what to show in the tables, I make a call to the business layer and “fetch” all the necessary fields.

### Presentation layer

There is a well-defined logical hierarchy in the design of GUI classes. The first instantiated class is the MainFrame which has a MainPanel. Everything is displayed within the MainPanel, including severel other sub-panels, like a text field for user-related messages and a tabbed panel for displaying an object called InternalPanel. The InternalPanel (which is an interface) can be instantiated as CustomerPanel, OrderPanel or ProductPanel. Each of these panels is of type JTable and has a model called CustomerTableModel, OrderTableModel and ProductTableModel respectively.

None of these classes communicate with the business level. I made a wrapper class, called Tablemanager, which handles all the communication, data fetching and data update. All contents of the tables are handled as String[][] Objects. Because of this the implementation, the Business and Presentation layers do not depend on each other and each layer can be easily modified without having to make large modifications on the other part.

The class diagram of the Presentation layer is presented on the next page.



### Business Logic layer

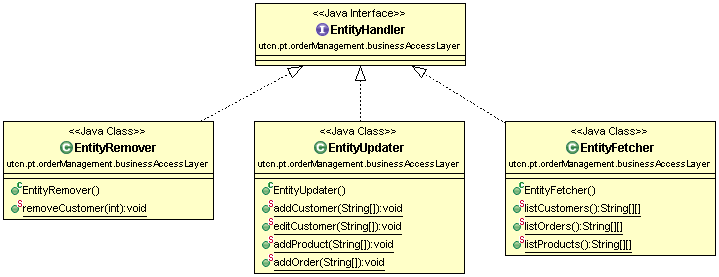
The Business Logic Layer does not provide many classes, however these are one of the most important ones: they handle passing data between layers and they read and write to the local storage of the entities.

The contents of this layer are:

* EntityFetcher
* EntityRemover
* EntityUpdater

Each of these classes implement the interface EntityHandler, which is an empty, marker interface. The handler classes do exactly what their name specifies. They usually take parameters from the Presentation layer as Strings (representing the rows in the JTables) and create a model based on these parameters, which then has to be manipulated (saved, edited, sent to the Data Access layer, etc.).

The class diagram of this layer is presented below:



### Data Access layer

The Data Access Layer has only one class, because it I only an “interface” for the physical database which is entirely managed by the Hibernate environment.

When instantiating the Database class, a connection to the database is created. In a configuration file, all the tables of the database are mapped to the entity classes of the application. The connection-related information is also provided in this configuration file (eg. database url, user name, password, type of the database, etc.).

The methods of the Database class are:

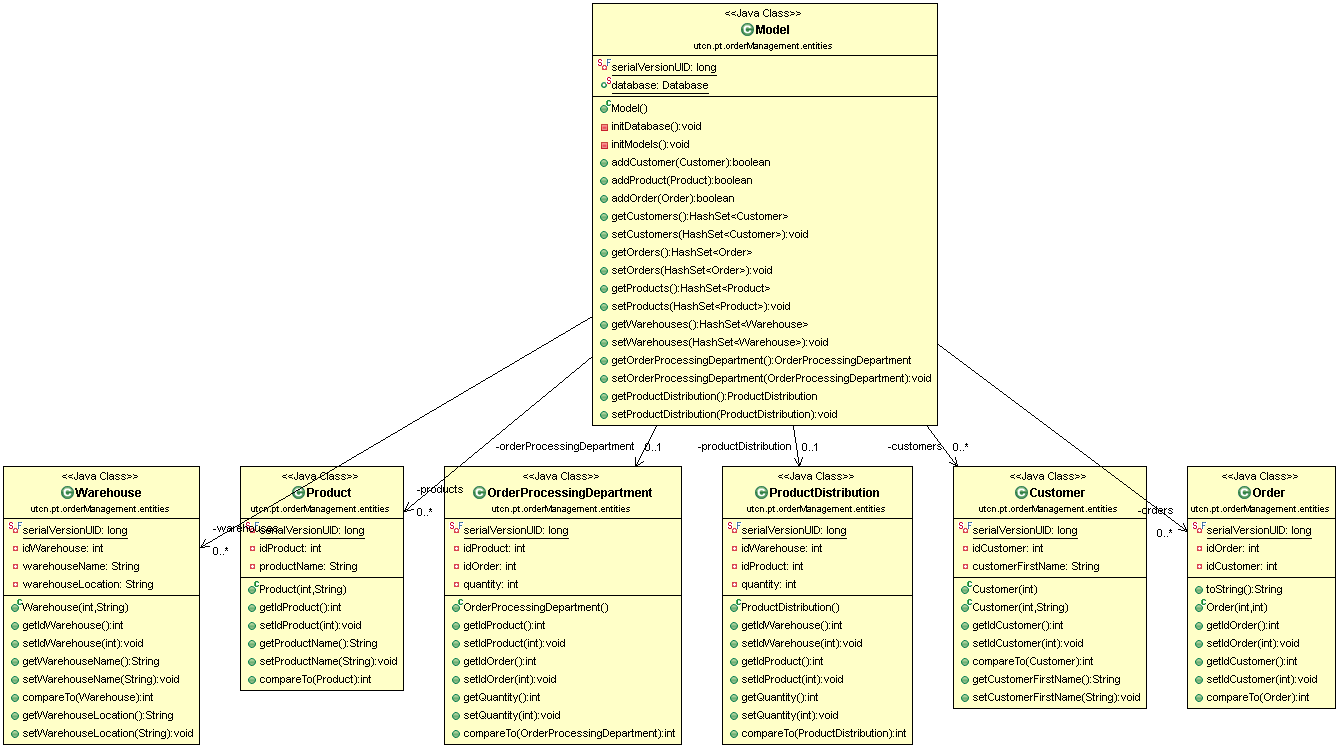
* *listCustomers()* – returns a List with all the customers;
* *listOrders()* – returns a List with all the orders;
* *listproducts()* – returns a List with all the products;
* *saveOrUpdate(Object)* – saves (or updates if it already exists) an entity object.

When using any of these methods, a new transaction is being created and then the values are inserted (or acquired). After the operation is completed, the transaction is closed.

### Other utility classes

The entry point of the application is the App class, which initializes the GUI and the Model class. The Model stores all the entities in separate HashSets. The reason for using HashSets is that we implicitly restrict duplicate elements (which is illegal in a database).

The class diagram of the models is shown below:



Some additional classes were also created (Warehouse, ProductDistribution, etc). These are used in later stages of the development to implement the one-to-many and many-to-many relationships.

# Implementation and testing

The implementation is as simple as the complexity if the program allows it. Each class has a well-defined purpose and one method does only one task at a time. This way the program remains modular and the coupling between these modules remains as low as possible. The implementation depended mostly on the data path of the information. Usually an event triggered at the Presentation layer had to modify data in the underlying levels. We can easily observe how the data is being pushed from one layer to the other. In most cases the representation of this data/information is dynamic and needs to be changed from level to level. For example, a customer is represented as a vector of Strings in the Presentation, as a class in the Business Level, and then it is saved as an entity in a database’s table.

As a first step, I implemented the GUI, to be able to represent the stored values and to be have a working, testable module. Then I implemented the models and made sure they are serializable and can be saved in a database. The third step was to make the connection to the database. This seems the hardest step because I used the Hibernate environment, which was brand new for me – although I did not have to waste time designing the database. Then, step-by-step I implemented the connection between the layers and realized each operation at a time.

In this way I could test the functionality of the program at each small modification.

# Results

As a result, I have made an application, which is does not offer a complete solution for the problem, however it is very well designed and can be anytime further extended. The basic functionality is still supported – one can connect to the database and enter/update/delete customers, orders or products.

# Conclusions

<What have I learned, further improvements & development>

I have learned that designing an application is at least as important as the implementation, however it has to be done as early as possible. Otherwise there is not enough time for the implementation.

I believe that my program obeys the main OOP rules and is highly modular and extensible from any point of view. It is also true that it lacks some parts of the implementation.

# Bibliography

* <http://www.tutorialspoint.com/hibernate/hibernate_environment.htm>
* <https://www.youtube.com/playlist?list=PL4AFF701184976B25>
* Other internet sources, like StackOvreflow