*Technical University of Cluj-Napoca*

*Faculty of Automation and Computers*

*Department of Computers*

*2nd Year, 2nd Semester 2015-2016*

Programming Techniques

Homework 1

Student: Robert Văcăreanu

Group: 30424

CONTENTS

1. Project specification
2. Problem analysis, model, testing and usage
3. Design
4. Implementation and testing
5. Results
6. Conclusions
7. Bibliography
8. Project Specification

Propose, design and implement a system for processing orders in a warehouse. Consider implementing it using a relational database. Organize your code according to the “Layers” architectural pattern.

The problem specification gives us flexibility in terms of: data structures used, design patterns, structure, functionality. The approach we’ve took in designing the application is described in the following paragraph.

We followed the requirements of using a relation database and the 4 layers architectural design pattern. This design pattern allows us, if necessary, to easily modify or replace its modules without, or with minimal changes across the remaining ones.

1. Problem Analysis, Model, Testing and Usage
2. Problem Analysis

In order to fulfill the requirements, we had to choose a database as well as a management system. We opt for MySQL, a well-known open-source relational database management system. To follow the requirements, we had to organize our code into four main modules, namely: Business, Data Layer, Model and Presentation. We further divided into some more sub-modules.

1. Model

Since our task is to model a warehouse and process orders, it is clear that, among other entities, there should be: customer, order and product.

We further added the classes: abstract user and admin for a better model. Both admin and customer extend abstract user. However, it is not directly represented in the database since, while having some fields in common, they will never have, or rarely, some data in common.

For a clear identification in the database, AbstractUser have two fields, namely: **name** and **password**. Further, we included in the Customer class a field named : **balance**. We represent them in the database by using their name as the primary key. Hence, there are not allowed two customers or two admins having the same name. For admins, this is simple since we will only have one: **admin**.

The class Product is represented as having three fields: an **id**, used in database for identification, a **price** which represents, as the name suggests, the amount of money the customer have to pay to buy it. There is also a field named : **name** which is defines the name of the product. There is a constructor that takes as argument a list of strings because, in the eventuality of using reflection, is much easier to work with the class this way. For example, to build an object from the database using reflection we would do: read as long as there are columns and add them in a list. Next, by using a hash map that links between the table name and the class name, or by choosing the table name the same with the class name, we can invoke Class.forName(String name) method then get the specific constructor and pass it the list as the argument. Everything else is taken care of by the constructor.

The class Order is represented as having 4 fields: a static id, used for creation of orders as well as dummy orders. Is it static since we will increment it when a new order is created. It is useful for keeping track of the numbers of objects created. This matters when adding to database because, while we can take advantage of the auto increment feature provided by the MySQL we choose this approach in order to be able to display to the user the id of the order. The second field is the actual **id** of the object. It is attributed as either the genericId, or the argument. The third field is an object, namely a Customer object. It is needed since every order have a customer. The fourth and final field of this class is a hash map that links a product to a integer. This is because we made the product to be unaware of the quantity since it doesn’t make sense for it to know about it. An order is capable of having multiple items as well as multiple same product. Hence, we need it a way of storing this in an order and a hash map suffice our needs.

1. Testing

The correct functionality of the application was proven by multiple tests that covers most of the situation a consumer will encounter.

1. Usage

The application comes with a fairly simple to use GUI. There are no overwhelming buttons, settings or other things like that. Everything is kept as simple and minimalistic as possible.

Exit



Admin mode. If clicked, there is not possible of clicking other buttons from this frame.

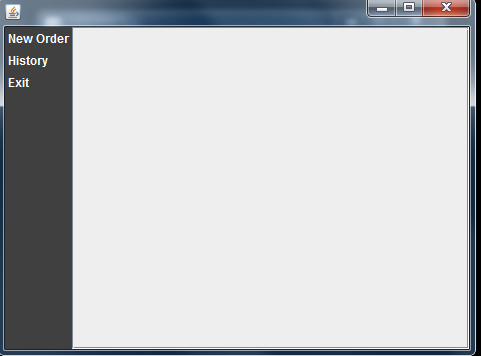
It opens a new frame corresponding to admin view

User mode. If clicked, there is not possible of clicking other buttons from this frame.

It opens a new frame corresponding to user view

A history of orders of the user

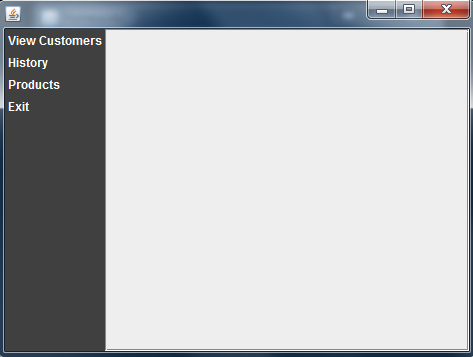
New Order view. Provides the user with the list of possible products and their prices



Exit button

**Admin View**

View a list with all customers. They can be modified, created, deleted. However, the integrity of the database must be preserved



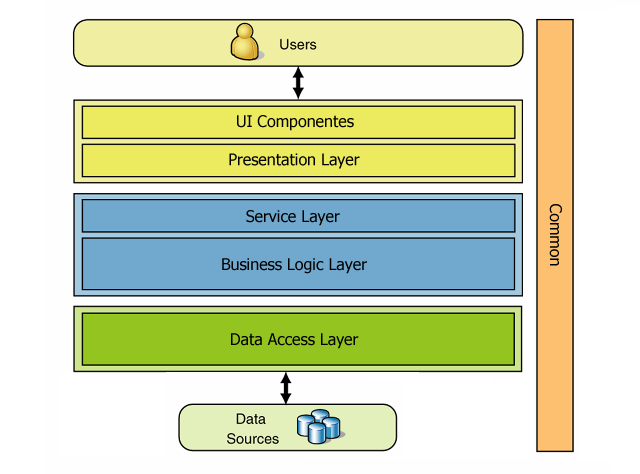
A list with all the products. They can be modified, created, deleted. However, the integrity of the database must be preserved

A list with the orders of all customers. They can be deleted, added. However, the integrity of the database must be preserved

1. Design

As mentioned above, we will split the implementation into four packages, following a **Layered Architecture**

approach.



This architectural design pattern provides a linear access through layers, unlike Model View Controler which provides circular access. This provides increased security and higher modularity.

The **Package business**:

Contains the following classes: **EntityDTO, Main, UserManager**

**EntityDTO**

This class is used to pass around the layers the needed informations fetched from the database. It is the only class that should communicate with the class **EntityDAO**. It contains specific following fields:

private List<Customer> users;

private List<Admin> admins;

private List<Order> orders;

private HashMap<Product, Integer> products;

private EntityDAO entitydao;

The first four fields, namely users, admins, orders and products are the collections used to store the data fetched from the database. The data is received by using an entityDAO object: entitydao.

It contains 17 methods that are used to: get, update, add objects to / from database.

**UserManager**

This class is the superviser of the actions done in the GUI. It authenticates and makes sure that the actions done by the users doesn’t do any harm to the app and to the datbase.

It is a singleton class. Since it is not a multi-threading app, the class is constructed in the classical way, no double locking or enums for singleton.

This class keeps the current user.

The classes from the presentation layers must use this class in order to access the database. According to the structural design pattern we are following, as well as any linear structural design pattern, we are not allowed to skip a level while accessing another. In order to access the bottom from the top, we must pass through all the intermediate layers. While this may seem like more trouble, it is actually very useful. Consider the case: get some user input that must be stored into database. Pass it to some business handler. There will be done checking, input validator etc. Next, pass it to Data layer: there will be put into the database, or not if it breaks the rules. If it is not, send feedback to businesslayer that will send feedback to the presentation layer.

**Package: DataLayer**

This packagae is at the lowest level in the schema. It is responsible with all the database interactions. It contains: another package, named dao from **D**ata **A**ccess **O**bject, a class Constants, a class Database Manager and a class Entity DAO. The inner package dao contains: Customer Data Access Object, Order Data Access bject and Product Data Access Object.

In the package dao is done all the queries to database. An instance to those classes is contained in the class EntityDAO. This provides like an abstraction to the actual layer between datalayer and database. All queries are hardcoded. This is because the queries wrote by the user are prone to mistakes. But if needed, this funtionality can be easily added . The class DatabaseManager provides to the other classes in the same package the connection to the database. This separation provides great flexibility for switching between different relational databases.

Every class from the dao package obtains its connection from the entity dao. After that, they perform their operations: create, read, update, delete. However, for order dao it was needed to be already fetched data provided by customer dao and product dao. This is because I chose to implement it like this:

An order has a customer and multiple products. A customer have multiple orders. The objects contained in the order objects are not created but added from the existing objects. This means that once a product or a customer is erased from the database all its orders or products from that orders are also deleted. However, it doesn’t make much sense to delete records from a database. The reason I chose to implement the fetching this way is due to efficiency. If we query the customers and the products and create the corresponding projects, why should we create the objects from the orders as well? This can lead to a lot of memory over used for convenience. In fact, the methods for orders querying are below 100 lines of code. The drawback is that this makes very hard to get the data from the database using reflection. This is because first: we need a way of knowing that we are querying that data and this kind of breaks the purpose of using reflection. This is because, when you use reflection you want to not care or care very very little of what objects you work with. By using a flag we actually care about what objects are we manipulating. Another set back in using reflaction for getting the orders is the fact that an order can have many products. This could, however, be over comed by considering the following scenario: when we query we do operations on that result set while there are rows. Then we create the order corresponding to the first row, ignoring the fact that the informations is not YET complete. Then while iterating through the result set we check to see if there are things to add to already objects. This means that we will need another classes, namely order data transfer object, product data transfer object and customer data transfer object. All of them should implement the interface DTO that provides some methods like: add, delete, update, get. In this methods, we make sure that the data from the reflection process is put togeher in the right way. Let’s further analyze the process/flow of getting the orders from the table by using reflection. We said above that we will need a class named order data transfer object ( orderDTO ). This is because when we extract an order from a row we will add it to its DTO. In the case of order, when we add a new oder we check to see if there is any order with the same id. If there is, we do the appropiate things like: add the products, increase the quantities of other objects etc.

All this pretty complicated methods are done due to the fact that the database is normalized up to 3.5 NF ( BCNF ) at least. There appear a multivalued dependency in the order table but this is not very important ( for now ).

The package **Model** was presented above. As a short recap: there are 5 classes, namely: AbstractUser, Admin, Customer, Order and Product. Only the last four have a corresponding table into the database.

The package **Presentation** contains another package: adminview and classes: AbstractWindow, AdminWindow, UserWindow and Window.

The inner package **adminview** contains CustomerView, OrderView, ProductView.

This inner package was created due to the fact that the code inside the admin view grows very fast and very large and adding new functionality would become hard and this is not what we want from an application.

The class **Window** is the GUI for the jframe that separates the user view from the admin view.

The class **AbstractWindow** is the class from which the classes admin window ad user window inherits. It is responsible with the log in. The log in uses a text field and a pass field. The pass field is used as recommended in the documentation because underlying sends an array of char. An array of char is preffered because an object of type string is immutable and this means that there will remain in memory until gargage collection comes into action. By remaining in memory means that it is easily accessible. This is not what you want from a field responsible with passwords

Both classes **AdminWindow** and **UserWindow** inherits from AbstractWindow. They come into play after the log in. The GUI for those is similar: there is on the left a panel used for navigation. It is static, meaning that it doesn’t modify according to the user action. On the right there is a panel controlled by the menu reprsented by the left panel. According to what button the user clicks, it displays the desired informations.

The left panel containts panel that were specifically crafted to look like labels. The reason buttons were chosen despite the posibility of using labels is the versatility of its action listeners.

The right panel is dynamically modified by the user actions on the left panel. Except one action, all the others leads to some JTable presented. The column names were generated using reflection. The method is very simple:

a for loop. Some complexity is added because we are only interested in the non-static fields, therefore we must discard the static ones. This is done inside the check: Modifier.isStatic(someField).

While designing the GUI, all the action listeners wrere written as annonymous inner classes. This leads to longer code but provides greater lizibility.

The left panel of the admin view contains the following options:

View Customers

History

Products

Exit

View Customers:

Provides the admin with the list of customers. The output is provided inside a jtable. The j table headers are generated using reflection. There are 3 actions available on this set of data: Save, Add Customer, Delete.

Save: used to save the modifications done upon existing data.

Add Customer: prompts the admin with a dialog and asks him to insert the desired attributes of the newly created customer

Delete: deletes the selected rows from the jtable as well as from all orders that the customer is part of

History:

Provides the admin with the list of orders of all customers. The output is provided inside a jtable. The j table headers are generated using reflection. There are 3 actions available on this set of data: Add, Delete.

Add: : prompts the admin with a dialog and asks him to insert the desired attributes of the newly created order

Delete: deletes the selected rows from the jtable

Products: Provides the admin with the list of customers. The output is provided inside a jtable. The j table headers are generated using reflection. There are 3 actions available on this set of data: Save, Add, Delete.

Save: saves the modification done

Add: prompts the admin with a dialog and asks him to insert the desired attributes of the newly created product

Delete: delete the products from the jtable as well as from all the orders that this product is parf of

Exit:

Exits the view

The user view

The left panel have 3 options: New Order, History, Exit

New Order: provides the user with the view of: all products, their price and a jtextfield for putting the desired quantity. The user cannot type a quantitty greater than the quantity in the warehouse. To the right of the Order button there is a J Label where the total cost will be put. To the left of Order button there is the current balance of the user.

When the user press the order button, the order is automatically and instantly processed.

History: provides the customer with the list with all of its orders in a jtable created using reflection.

The user cannot modify it, as it doesn’t make any sense because, what if the user will buy 1 product and will modify it to be 10?

The Exit : exits the view

From the above explanations, it is clear that the admin view is more complex and bigger than the user view. Hence the need of breaking it into more classes:

Customer View: This class takes the responsibility of providing a clean interface with all the details and functionality available for customers: create, read, update, delete.

Order View: This class takes the responsibility of providing a clean interface with all the details and functionality available for orders: create, read, delete.

Product View: This class takes the responsibility of providing a clean interface with all the details and functionality available for products: create, read, update, delete.

On all of those four classes there are some exception thrown, catched. This is because of the database errors. We want to make the user aware of its mikstakes, hence we dsplay a option dialog.

1. Conclusions

While the difficulty depends on the model and the way the programmer chose to implement it, I evaluate the difficulty as being somewhere between easy-medium. Firstly, because database management can get hard. This is not the case, since we modeled only 4 small tables. But take for example the case of hundreds of tables in gigabytes of data

I learnt from implementing this project how difficult can be to verify and modify the user input while giving him the freedom of putting the input however seems clear to him while also making sense to others.