page1image33832960

Software Architecture Coursework

WEIGUANG RAN

40412989@live.napier.ac.uk SET10101

14/October/2019

**1. Abstract**

A nationwide retailing corporation (such as Homebase http://www.homebase.co.uk/ or B&Q http://www.diy.com/) is planning to develop a new distributed store management system for their retail branches to provide better coordination of their business. It has named the proposed system DE-Store. Your company want to pitch for the software development contract and plan to do this by developing a software prototype of an architecture that you believe would show that you could meet the needs of the project.

DE-Store is NOT an online shopping system; instead it is a DISTRIBUTED business management system. Please note that DE-Store is a DISTRIBUTED system. You are supposed to use the appropriate architecture styles and technologies you’ve learnt to develop an effective solution, such as client/server, peer to peer, service-oriented, RMI, three-tiered, etc. DE-Store is expected to be an expandable and adaptive system to accommodate changing business requirements in the future.

DE-Store aims to have a suite of store management functionalities such as price control, inventory control, delivery charge, approval of financial support, and performance analysis.

• Price Control: DE-Store allows the store manager to set the price of the products and to select products on a variety of sale offers, which include 3 for 2, buy one get one free, free delivery charges.

• Inventory Control: stock is monitored all the time by uploading data from the warehouse database. Items out of stock will be ordered from the central inventory system at the headquarters. DE-Store generates warning messages for items in low stock automatically and also sends them to the mobile message box of the store manager.

• Loyalty Card: the store can make further special offers to customers who regularly use their branches.

• Finance Approval: DE-Store offers its customer the opportunity to buy now and pay later using an online finance system, Enabling, which is linked to DE-Store via a portal.

• Reports and Analysis: DE-Store tracks the purchase activities of customers from the accounting database and generates reports on how the store is performing.

DE-store is expected to be an expandable and adaptive system to accommodate changing business requirements in the future.

**Key words: software architecture, distributed system, design**

# Architecture Recommendations

Just as the requirement mentioned that this is the distributed business management system, and DE-Store is expected to be an expandable and adaptive system to accommodate changing business requirements in the future. Therefore, it is very important to upgrade the software without affecting the overall architecture of the software. The two Architecture Systems that have been chosen are **Server-Oriented** and **three layer**.

## Server-Oriented

When looking at larger systems, the relationship between requirements and components of a system are complex. The basic principles of service-oriented architecture are independent of vendors, products and technologies. This means that any changes to the requirements that need to be made may result in changing multiple components. However, in server-oriented-architecture, the service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently. The requirement may be implemented by numerous components and vice versa.

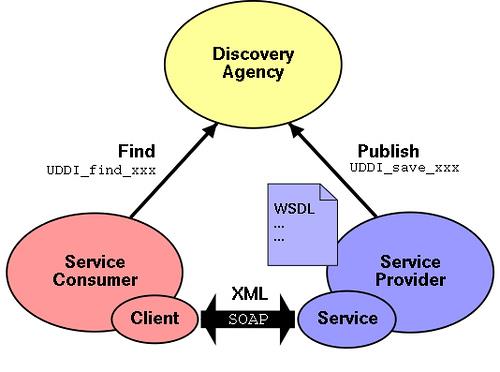


Figure1: Server-Oriented

### What is it?

**Service-Oriented Architecture (SOA)** is a component model in which different functional units (called services) of an application are linked through well-defined interfaces and contracts between these services. The interface is defined in a neutral manner and should be independent of the hardware platform, operating system and programming language that implement the service. This allows services in a variety of such systems to interact in a unified and versatile manner.

This feature of a neutral interface definition (without forcing binding to a particular implementation) is called loose replacement between services. The advantages of the loose conversion system are twofold, one is its replacement, and the other is that it can continue to exist when the internal structure and implementation of each service of the entire application is changed step by step. Concatenation, which means that the interface functions and structures between the different components of the application are closely linked, so they are vulnerable to some form of change to some or all of the application.

（Services exist as physically independent software programs with distinct design characteristics that support the attainment of the strategic goals associated with service- oriented computing. Each service is assigned its own distinct functional context and is comprised of a set of capabilities related to this context. Those capabilities suitable for invocation by external consumer programs are commonly expressed via a published service contract.）

（erl.2007）

### Why use it?

One of the main benefits of using SOA systems is the system more maintainable. When the requirements change, there is no need to modify the interface that provides the service. Only the business service process or the modification operation needs to be adjusted, and the entire application system is more easily maintained.

### Advantages

### 1. The service communicates through a simple, precisely defined interface, without involving the underlying programming interface and communication model.

### 2.Coarse-grained: The coarse-grained service provides a specific business function. The advantage of using the coarse-grained service interface is that there is no need to perform multiple reciprocations between the user and the service layer. One reciprocation is sufficient.

### 3. Loose coupling: Loose coupling requires that a loosely coupled relationship between different services in the SOA architecture should be maintained, that is, a relatively independent and independent relationship should be maintained.

### 4. Location Transparency: Location transparency requires that all services in an SOA system are location-transparent to their callers, that is, the caller of each service only needs to know which service to invoke, but It is not necessary to know where the physical location of the called service is.

### 5. Protocol independence: Protocol independence requires that each service can be called through a different protocol.

### Disadvantages

1. Reduced system performance
2. Increased profile costs in the transition to excessive standardization
3. A lot of file type information that doesn't make much sense

**4.** Highly demanding plans for business processes

## Three-tier

In software engineering, multitier architecture or multi-tier architecture is a client–server architecture in which presentation, application processing, and data management functions are physically separated. The most widespread use of multitier architecture is the three-tier architecture. And tree layer is based on browse/server architecture system.

### What is it?

A multi-tier architecture is a coping strategy based on isolation control that developers face in the development process in the face of complex and volatile requirements. Each layer can be deployed separately. The whole project is divided into bottom-up:

UI (presentation) layer,

Logic (business) layer,

Data persistence (data access) layer.

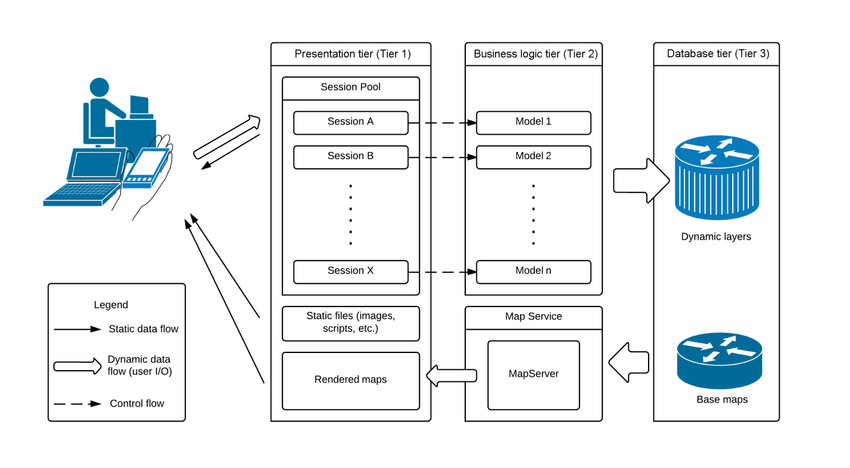


Figure2: three-layer architecture (From <https://www.researchgate.net/figure/Three-tier-architecture-of-HoLSAT_fig1_273140450>, Yang yang)

Data access layer: A data storage device responsible for persisting data responses, such as Database, Txt, Excel, etc.

Business logic layer: Responsible for processing a series of logic and services customized to meet software requirements. For example, after the user places an order at the front end, the entire business flow may involve obtaining user information, obtaining product information, obtaining shopping cart information, and verifying. Whether the quantity that can be purchased can satisfy this purchase, and generate different preferential strategies for the user identity. At the same time, it will verify the validity of the data generated by the cookie, Session, etc., and finally the order will be generated, and the order will generate warehousing and logistics. The series of Erp system services, all of which belong to the business logic of the "order" requirement.

Presentation layer: The interface responsible for interacting with the user.

### Why use it?

One of the grounds for using a three-tier architecture for software is change-oriented. When stratification, the boundaries of the hierarchy should be determined for different reasons of change. It is forbidden to interfere with each other at the level, or at least minimize the impact of changes on each layer. For example, the modification of the database structure will naturally affect the data model of the infrastructure layer and the domain model of the domain layer, but when we only need to modify the implementation logic of database access in the infrastructure layer, it should not affect the domain layer. In other words, tiered applications are most suitable for commercial applications that are constantly changing requirements, and this is entirely satisfied the purpose of this project.

### Advantages:

1. No need to develop client software, easy to maintain and upgrade.

2. Can be operated across platforms.

3. It has good openness and expandability.

4. Easy to transplant the database.

5. Good security.

6. Resource reuse is good.

### Disadvantages:

1. Sometimes it leads to cascading changes. This modification is especially reflected in the top-down direction. If you need to add a function in the presentation layer, in order to ensure that its design conforms to the hierarchical structure, you may need to add corresponding code in the corresponding business logic layer and data access layer.
2. Compared with the non-hierarchical programming method, the application of the three-layer or multi-layer architecture is inefficient, the code is large, and the difficulty is increased.

## Final recommendation:

### Decision:

Finally, I decide to use three-layer architecture system style. The most important reason is three-layer is very good at decoupling in architecture design. DE-STORE is an online store management system, so the data will update very often, and it is important if the software engineering can develop and change coding separately, the changing of each layer would not influence other layers.

### Three-tier layers architecture

Three-tier layers architecture is very good for digital business companies or big online companies. The first reason is the software developer can only just focus on one layer for the whole structure. For instance, the front-end developer can just work on “UI” layer, this can greatly improve the development efficiency, but also facilitate the software’s later maintenance. The second reason is that this architecture style can reduce layer-to-layer dependencies, which means if the platform needs to be upgraded or the data is updated, it can be done without affecting the user login operation, which is very important for the e-commerce platform.

### Object oriented programming

The Model-View-Controller pattern is being used to develop an interactive system by splitting them into three components:

* Model: Contains the core functionality of the system and the Data
* View: Contains the GUI information
* Controller: Handles all the user input and logical processes.

This pattern uses encapsulation, as the components of the system only have access to what they need to, and the rest is hidden, which also allows code to be reused. Polymorphism is also being used to reuse code, as instead of creating separate classes for each patient, one patient class can be created, and then used as a basis to create an object from that class.

### Three layer architecture

Three-layer architecture will also use in this project. However, it is different from Three-tier architecture style. First of all, tier-to-tier is simply the physical separation of the app components, but layer-to-layer act as more logical separators that exist to separate and organize the actual code. For example, people often hear terms like “business logic layer”, “presentation layer” etc. These are simply always to organize all of the code within the application.

If there is a web app that contains the data access and business logic which is running on the same machine/server, then it can be called 3-layered app in 1-Tier. And if the data access is hosted on different machine/server, and the business is also hosted in different machine/server, then it can be called 3-layerd app in 3-tier. In this project, I will use 3-tier- layer architecture style.

# Design and choose modelling

## Analysis

The program is a basic database administration application. It allows the user (product manager) to manage the clients (add, update, delete), the products (add, update, delete, sale offers etc.) and the Inventory Control (can create new orders). Everything is stored in a database and the data is available even after the program is restarted. The data is displayed neatly in tables and it can be easily accessed and modified. The application can be useful for warehouse managers and administrators.

## Modelling

The approach I have chosen for modelling this problem involves the creation and use of a layered architecture. This involves the use of three-tier layers: the data access layer, which works directly with the database, the bussiness logic layer, which is concerned with manipulating the data from the database and from the user, and the presentation layer, which is used for communicating with the user. Additionally, all the classes in these layers use some classes that represent the entities from the database, classes which are used to simplify the process of retrieving data and updating the database. Just as the figure shows below:

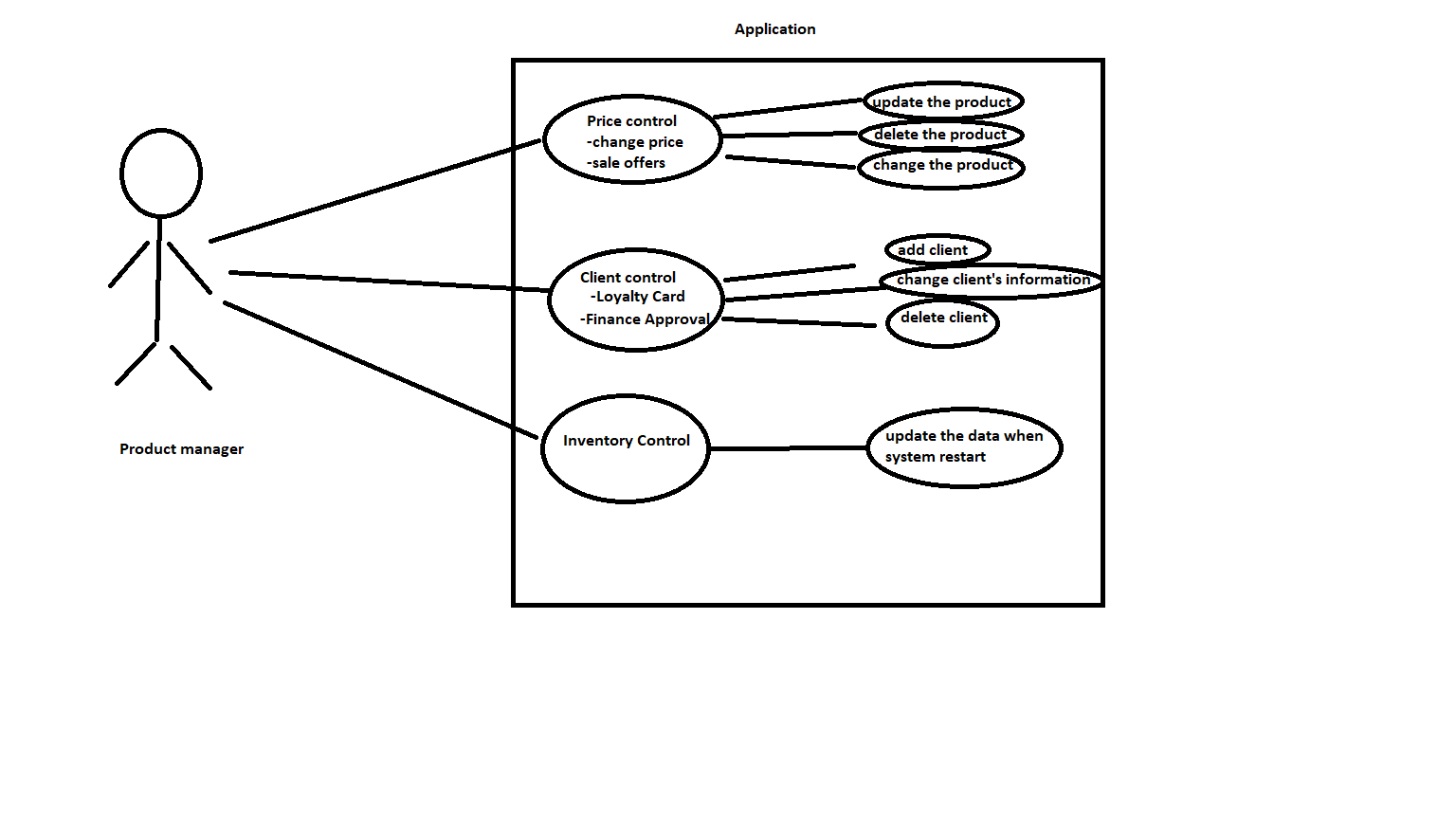


Figure3: application modelling

## Design

### Design decisions

The application uses both the Model-View-Controller architecture and the 3-tier layered architecture. Thus, from the MVC point of view, there is a separation between the problem Model (the classes concerned with solving the problem in a correct manner), the View (the classes concerned with the GUI: the parts which must be displayed on the screen, the interface between the user and the applications: text fields, text areas, buttons) and the Controller (the class which make possible for the View classes to send messages to the Model classes and by this allow the user to control the application). From the layered architecture point of view, the Controller and the View are part of the top layer, while the Model actually contains the other two layers: business logic and data access and also the actual model of the database.

### UML class diagram

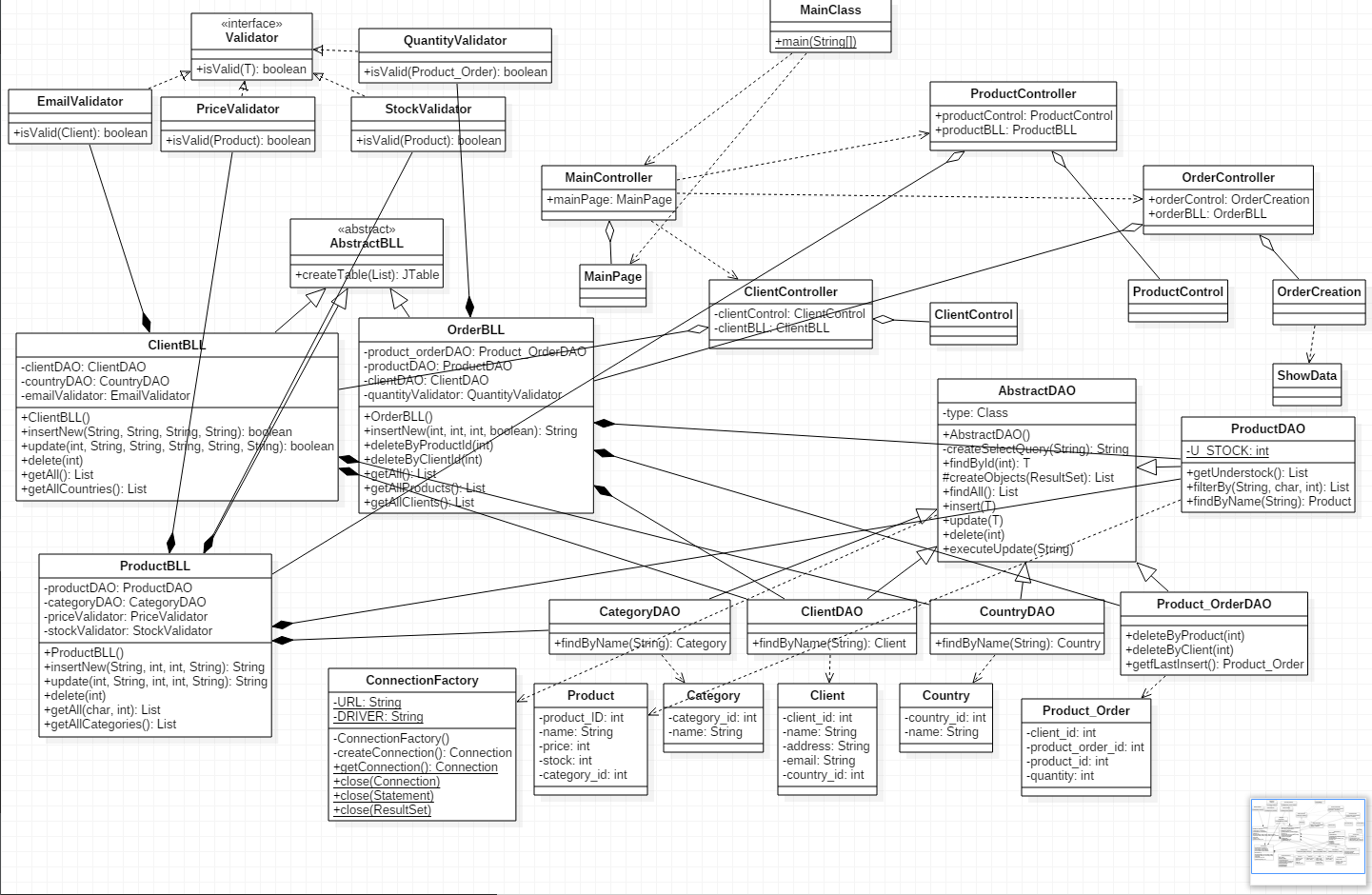


Figure4: UML class diagram

### Data Structure

In this coursework, there are two main data structure: *arrays* and *lists*:

***Lists*** and namely the *Vector* and *ArrayList* implementation of the interface are used for arrays of variable (and often changing) length, because they are easier to use and have already-implemented methods for modifying the size and the required memory, thus being more efficient. They are used to model the lists of objects returned from the database by various queries.

### Class Design

There are 8 different classes in this project:

* **AbstractDAO:** this abstract class is concerned with accessing the database. Through its method it can perform data retrievals, updates, insertions and deletions. Its methods work for any type of data because it uses reflection techniques. This class is extended by classes specific for each table, which implement other methods which require database access, like search for a specific name. These classes are: CategoryDAO, ClientDAO, CountryDAO, Product\_OrderDAO, ProductDAO.
* **AbstractBLL:** this abstract class is used as a base for the business logic level classes specific for each operation. It contains a common method that creates a JTable starting from a list of object by using reflection techniques. This class is extended by ClientBLL, ProductBLL, OrderBLL, which all have specific methods for dealing with their tables in the database. These subclasses work with the DAO classes, having methods to insert, update, delete and retrieve data.
* **ConnectionFactory:** this class is built in accordance with the *Singleton design pattern*. What is Singleton design pattern? The singleton design pattern is only one object of this class can exist at any moment, and the constructor is private so the object can only be instantiated by calling static methods of this class. This class is used to open and close the database connections and return them to be used for accessing the database.
* **View** classes**:** these classes represent the graphical interface between the user and the application. The classes are MainPage (the starting window), ClientControl (the client administration window), ProductControl (the product administration window), OrderCreation (the window used to create new orders) and ShowData (a window used to show a table on screen).
* **Controller** classes**:** these classes are part of the MVC architecture. Through the use of inner classes, they provide the graphic interface with their functionalities, by matching the ActionListeners with methods from the BLL classes. These classes are: MainController (controls MainPage), ClientController (controls ClientControl with ClientBLL), ProductController (controls ProductControl with ProductBLL) and OrderController (controls OrderCreation with OrderBLL).
* **Validator** classes**:** objects of this class are used to validate the data before it is inserted in the database. They implement the **Validator** interface. Each BLL class has some Validator classes associated with it.
* **Category, Client, Country, Product\_Order, Product:** these classes match perfectly the entities from the database. They are used to facilitate data manipulation: objects of this classes are returned by methods that query the database, and also objects of these classes are used by the methods which update the database. These classes do not offer other functionalities apart from the part of the Model-View-Controller architecture, this class is used to create the Queue objects, assign a QueueStatistics object to each one, create the ClientProducer object, assign a ShopStatistics object to it, start all the threads and after their execution finishes, send the resulting information to be printed.
* **MainClass:** this class is used to start the application through its main() method.

### Interfaces

* **Validator:** this interface is used to validate the objects to be inserted in the database so that they do not create database errors.

### Relationships

There are a lot of relationships between the classes. The following section presents the most important ones:

* (Main/Proudct/Order/Client)Controller (aggregation) MainPage/ ProductControl/OrderCreation/ClientControl, /ProductBLL/OrderBLL/ ClientBLL: The ()Controller receives its View and its BLL in the constructor: the latter is used to implement the action listeners for the former
* BLL (composition) DAO: Objects of the BLL classes have DAO objects as their instance variables but the objects are created inside the constructor, not outside it. The DAO objects are used to implement the BLL methods.
* AbstractDAO (dependency) ConnectionFactory: The AbstractDAO class uses methods from the ConnectionFactory class, but not objects.
* BLL (composition) Validator: The BLL classes have Validator objects as their instance variables but the objects are created inside the constructor, not outside it.

## Implementation:

Please go to Appendix2

## Result:

### Conclusion:

After building this project I have found that chosen the right architecture style is the first step for the project. It is very important because it will waste a lot of time if we choose the wrong architecture style. Therefore, familiar with mastering several different software architecture styles are very important for being a good architect in the future.

### Lesson learned:

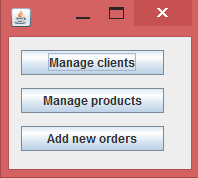
From this coursework, I learned some different software architecture style (SOA, client-server, peer to peer, service-oriented, RMI, three-tiered etc.) and knew how to use them in different problems and situations. Also, I have learned how to work with reflection techniques and database connection and access, and how to implement the Singleton design pattern.

### Improvement:

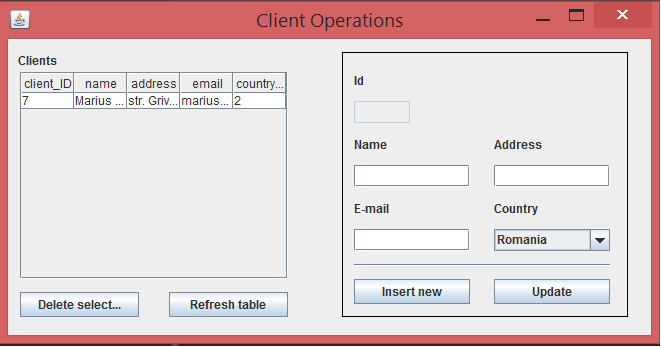
1. Build the simple three-tier architecture structure with Java.
2. Enhance the graphic user interface in order to show the database evolution in a graphical manner.
3. Design the suitable architecture in specific situation.
4. Improve my MVC architectural pattern skill.

# Appendix

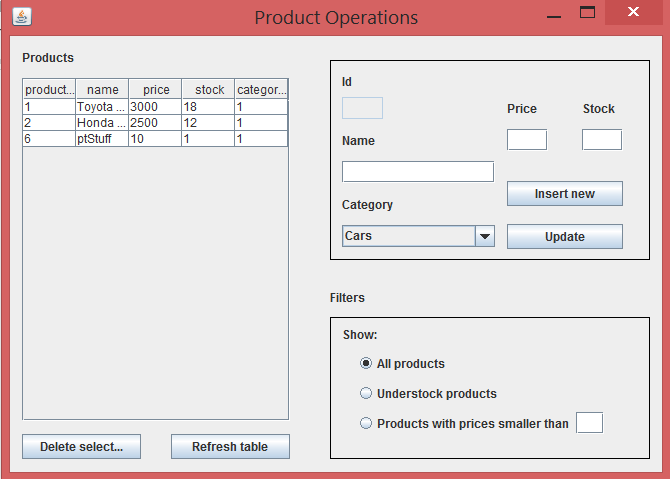
## Appendix1: User Interface



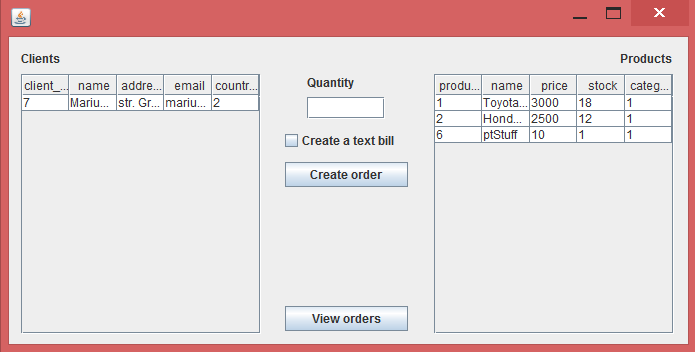
Main window



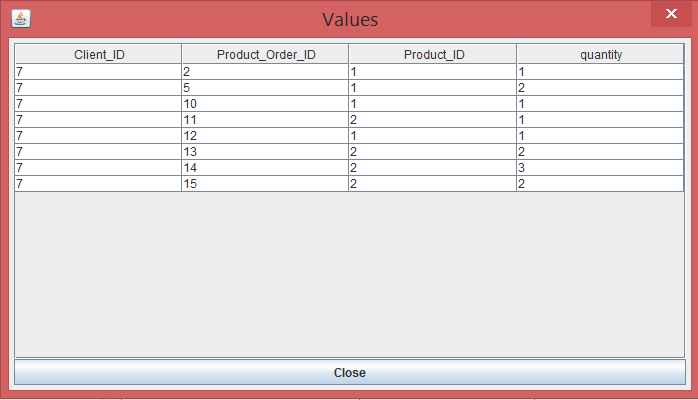
Client manage window



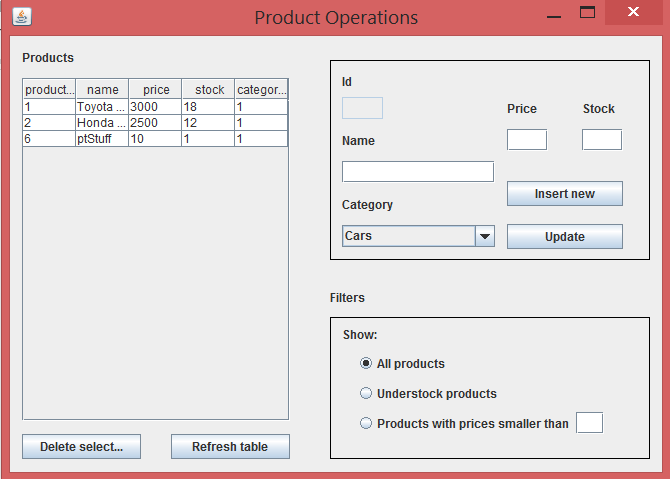
Product manage window



The Order insertion window, together with two tables showing the contents of the database.



The window which shows the Orders.



The Product management window, together with the table showing the contents of the database.

## Appendix2: Implementation

* The **AbstractDAO<T>** abstract class: has a type (Class<T>) as instance variable. Methods:
  + Constructor: assigns to the type a Class coresponding to the generic class used
  + createSelectQuery: has a String as parametera, it creates a query to be used on the database based on the String given as parameter which correesponds to a field in the database, if the input parameter is null, it creates a query to retrieve everything from the table;
  + findByID: has an int as parameter, it returns the object corresponding to the value from the database with the given input id;

**public** **T** **findByID**(**int** id) **throws** **Exception**{

Connection **con**=**null**;

PreparedStatement **pS**=**null**;

ResultSet **rS**=**null**;

**String** **query**=createSelectQuery(type.getSimpleName()+"\_ID");

con=**ConnectionFactory**.*getConnection*();

**try** {

pS = con.prepareStatement(query);

pS.setInt(1, id);

rS = pS.executeQuery();

**return** createObjects(rS).get(0);

} **catch** (**Exception** **e**) {

**throw** e;

} **finally**{

**ConnectionFactory**.*close*(con);

**ConnectionFactory**.*close*(pS);

**ConnectionFactory**.*close*(rS);

}

}

* + createObjects: has a ResultSet as parameter, it uses reflection techniques to create a List of objects based on the type instace variable and on the ResultSet;

**protected** List<**T**> **createObjects**(ResultSet rS) **throws** **Exception**{

List<**T**> **list**=**new** ArrayList<**T**>();

**try** {

**while**(rS.next()){

**T** **instance**=type.newInstance();

**for**(**Field** **field**:type.getDeclaredFields()){

**Object** **value**=rS.getObject(field.getName());

**PropertyDescriptor** **pD**=**new** PropertyDescriptor(field.getName(), type);

**Method** **method**=pD.getWriteMethod();

method.invoke(instance, value);

}

list.add(instance);

}

**return** list;

} **catch** (**Exception** **e**) {

**throw** e;

}

}

* + findAll: it returns a List of objects corresponding to all the rows from a table from the database;
  + insert: has a object T as parameters, it inserts the given object in the database through the use of reflection techniques;
  + update: has a object T as parameters, it updates the given object in the database through the use of reflection techniques;

**public** **void** **insert**(**T** t) **throws** **Exception**{

**String** **insertQuery**="Insert into "+type.getSimpleName()+" values (";

**try** {

**Object** **value**=**null**;

**for** (**Field** **field** : type.getDeclaredFields()) {

**if**(field.getName().equalsIgnoreCase(type.getSimpleName()+"\_ID")){

**continue**;

}

**PropertyDescriptor** **pD** = **new** PropertyDescriptor(field.getName(), type);

**Method** **method** = pD.getReadMethod();

value=method.invoke(t);

insertQuery += "\'" + value + "\',";

}

insertQuery = insertQuery.substring(0, insertQuery.length() - 1);

insertQuery += ")";

executeUpdate(insertQuery);

} **catch** (**Exception** **e**) {

**throw** e;

}

}

* + delete: has an id as parameter (int), it deletes from the database the entry with the given id;
  + executeUpdate: has a String as parameter, it executes the query given as parameter on the database (insert/update/delete)
* The **DAO** classes: extend the **AbstractDAO** class by adding specific methods:

**public** **Country** **findByName**(**String** name) **throws** **Exception**{

**String** **query**="Select \* from Country where name=\'"+name+"\'";

Connection **con**=**null**;

PreparedStatement **pS**=**null**;

ResultSet **rS**=**null**;

con=**ConnectionFactory**.*getConnection*();

**try** {

pS = con.prepareStatement(query);

rS = pS.executeQuery();

**return** createObjects(rS).get(0);

} **catch** (**Exception** **e**) {

**throw** e;

} **finally**{

**ConnectionFactory**.*close*(con);

**ConnectionFactory**.*close*(pS);

**ConnectionFactory**.*close*(rS);

}

}

* + findByName: has a String as parameter, it returns the entry in the database with the given name;
  + getUnderstock: it returns a list of Product objects that have their stock instance variable lower than the static field of the class;
* The Model classes **Client, Category, Country, Product, Product\_Order**: have different fields corresponding to the database fields and mehods for accessing and mutating these fields. Also have both a parameterless constructor and one with parameters.
* The **AbstractBLL** abstract class: contains a single method:
  + createJTable: has a List of objects as parameter, it returns a Jtable containing the given objects by iterating through their type using reflection techniques.

**public** **JTable** **createTable**(List<Object> objects){

**Class**<?> **type**=objects.get(0).getClass();

**Vector**<Object> **columnNames**=**new** Vector<Object>();

**for**(**Field** **field**:type.getDeclaredFields()){

columnNames.add(field.getName());

}

**Vector**<**Vector**<Object>> **data**=**new** Vector<**Vector**<Object>>();

**for**(**Object** **entry**:objects){

**Vector**<Object> **row**= **new** Vector<Object>();

**for**(**Field** **field**:type.getDeclaredFields()){

**try** {

**PropertyDescriptor** **pD**=**new** PropertyDescriptor(field.getName(), type);

**Method** **method**=pD.getReadMethod();

**Object** **value**=method.invoke(entry);

row.add(value);

} **catch** (**Exception** **e**) {

e.printStackTrace();

}

}

data.add(row);

}

**return** **new** JTable(data, columnNames);

}

* The **BLL** classes extend the **AbstractBLL** class by adding business specific methods. They have various DAO objects as instance variables and also Validator objects. Methods include:
  + Constructor: parameterless, each instance variable receives a new object created inside this method;
  + insertNew: has various parameters depending on the specifics of the BLL class. It is used to insert new objects in the database. The object is created and validated inside the method. If the object is not valid, the method returns an error message to be sent to the user. For foreign keys, this method does not use the foreign table id, but the name of the row, obtaining the name by using the findByName methods in the DAO objects;
  + update: has various parameters depending on the specifics of the BLL class. It is used to update objects in the database. The object is created and validated inside the method. If the object is not valid, the method returns an error message to be sent to the user;

**public** **boolean** **update**(**int** id, **String** name, **String** address, **String** email, **String** country) **throws** **Exception**{

countryDAO=**new** CountryDAO();

**try** {

**Country** **countryObj**=countryDAO.findByName(country);

**Client** **client**=**new** Client(id, name, address, email, countryObj.getCountry\_id());

**if**(!emailValidator.isValid(client)){

**return** **false**;

}

clientDAO.update(client);

} **catch** (**Exception** **e**) {

// **TODO** Auto-generated catch block

e.printStackTrace();

**throw** e;

}

**return** **true**;

}

* + delete: has an integer id as parameter, it is used to delete an entry from the database;
  + getAll: returns a List of all the entries from a table in the database;
* The **ConnectionFactory** class: Built in accordance with the Singleton design pattern: only one object of this class can exist at any given moment. Methods:
  + Constructor: is private, it opens the database connection;
  + createConnection: is private, it returns an object of type Connection;
  + getConnection: static, it returns the Connection created by createConnection;
  + close: can have a Connection, a ResultSet or a Statement as parameter, it closes the given object;
* The **MainPage, ClientControl, OrderCreation, ProductControl, ShowData** GUI classes were built with **WindowBuilder**, but they were heavily modified. The text fields, buttons and combo boxes were transformed in instance variables so that their values can be accessed from outside. Also, additional method were added:
  + Add()ActionListener: adds an action listener to a button;
  + Getters for the instance variables;
  + showData: has a String as parameter, it outputs a message dialog containing the String;
  + setTable: has a JTable as input parameter, it sets the table on the window to the input table, also adding a MouseAdapter to the table.

**public** **void** **setTable**(**JTable** newTable){

table=newTable;

**JScrollPane** **scrollPane** = **new** JScrollPane();

scrollPane.setBounds(12, 33, 268, 207);

contentPane.add(scrollPane);

scrollPane.setViewportView(table);

table.addMouseListener(**new** MouseAdapter(){

***@Override***

**public** **void** **mouseClicked**(**MouseEvent** evt) {

**int** **row** = table.rowAtPoint(evt.getPoint());

**if** (row >= 0) {

textField\_3.setText(table.getValueAt(row, 0).toString());

textField.setText((**String**)table.getValueAt(row, 1));

textField\_1.setText((**String**)table.getValueAt(row, 2));

textField\_2.setText((**String**)table.getValueAt(row, 3));

comboBox.setSelectedIndex((**Integer**)table.getValueAt(row, 4)-1);

}

}

});

}

* The **Controller** classes: have a BLL and a GUI object as instance parameters. Methods:
  + Constructor: sets the corresponding fields.
  + **ActionListener**(s): inner classes which implements the ActionListener interface. Methods:

**class** **UpdateActionListener** **implements** ActionListener{

**public** **void** **actionPerformed**(**ActionEvent** arg0) {

**try** {

**if**(!clientBLL.update(**Integer**.*parseInt*(clientControl.getTextField\_3().getText()),

clientControl.getTextField().getText(), clientControl.getTextField\_1().getText(),

clientControl.getTextField\_2().getText(), (**String**)clientControl.getComboBox().getSelectedItem())){

clientControl.showMessage("Invalid email address format!");

}

} **catch** (**Exception** **e**) {

clientControl.showMessage("Could not update, a database error ocurred!");

}

}

}

* + - actionPerformed: has an ActionEvent as parameter. The method retrieves data from the GUI and calls the methods from the BLL objects whenever the user pushes a button.
* The **Validator** classes: implement the *Validator* interface, are used to validate objects before they are inserted in the database. Method:
  + isValid: has an object as parameter, returns true if the object is valid and false otherwise.