Technical University of Cluj-Napoca

Faculty of Automation and Computer Science

Computer Science Department

Year 2015-2016 Semester II

Programming Techniques

*2nd Project*

Student: Dan Toderici

Group: 30422

Contents

[Project specification 3](#_Toc446960610)

[Problem Analysis 3](#_Toc446960611)

[Modeling 4](#_Toc446960612)

[Use Case Diagram 6](#_Toc446960613)

[Design 7](#_Toc446960614)

[Relational Diagram 7](#_Toc446960615)

[Class Structure 9](#_Toc446960616)

[Implementation 10](#_Toc446960617)

[Model 10](#_Toc446960618)

[Persistence 11](#_Toc446960619)

[Controllers and Views 11](#_Toc446960620)

[Data Layer 12](#_Toc446960621)

[Business Layer 12](#_Toc446960622)

[User Layer 13](#_Toc446960623)

[BillMaker 13](#_Toc446960624)

[Conclusion and further developments 14](#_Toc446960625)

[References 14](#_Toc446960626)

# **Project specification**

Propose is to design and implement a system that simulates an order management application for processing customer orders. It is going to be just like a shop database implementation with its specific functionalities and data.

The application, that resulted from design such system, can perform 11 main operations:

1) Insertion of a new client

2) Editing of client’s details

3) Deletion of client’s info

4) Insertion of a new product

5) Editing of product’s details

6) Deletion of product’s info

7) Creating a .txt or .pdf with order details for each user

8) Display products in a JTabel format

9) Display order details for each client

10) Display client details in a JTable format

11) Display messages (ex. Understock) when trying to insert data

However, all these operations must be executed based on user input. Therefore, a graphical user interface is needed. The UI must provide the user methods to insert orders and to choose which operations he or she desires to be execute. There will be many views and the users group will be divided into 2, admins and basic user. Adaptive view was not required, so you can generate separate views for both types of users.

# Problem Analysis

Splitting the problem into smaller parts is the first main problem. Due to the need of persistence and further requirements, a package which operates on a database is needed. Not to forget about designing a proper database, this will help us much more than an xml or a basic file! Moreover, in order to execute all the commands received from the user, a business logic package will handle fetching and executing the commands. Fetching refers to gathering data from the database, asses the correctness of the user input and only then executes the command or warn the user upon the usability and integrity of its inputted data.

Operations on the database, as expected must be in conformity with the Data Access Object pattern (DAO); this will be explained later when reaching the implementation and considerations chapters.

Views, due to my previous experience regarding Layered Architecture and Model View Controller designs, will follow the same basic rule. Views are the dumbest, but the most used. They do one thing only but perfectly, which is to show the user information and provide resources for him to pass his or her desired command.

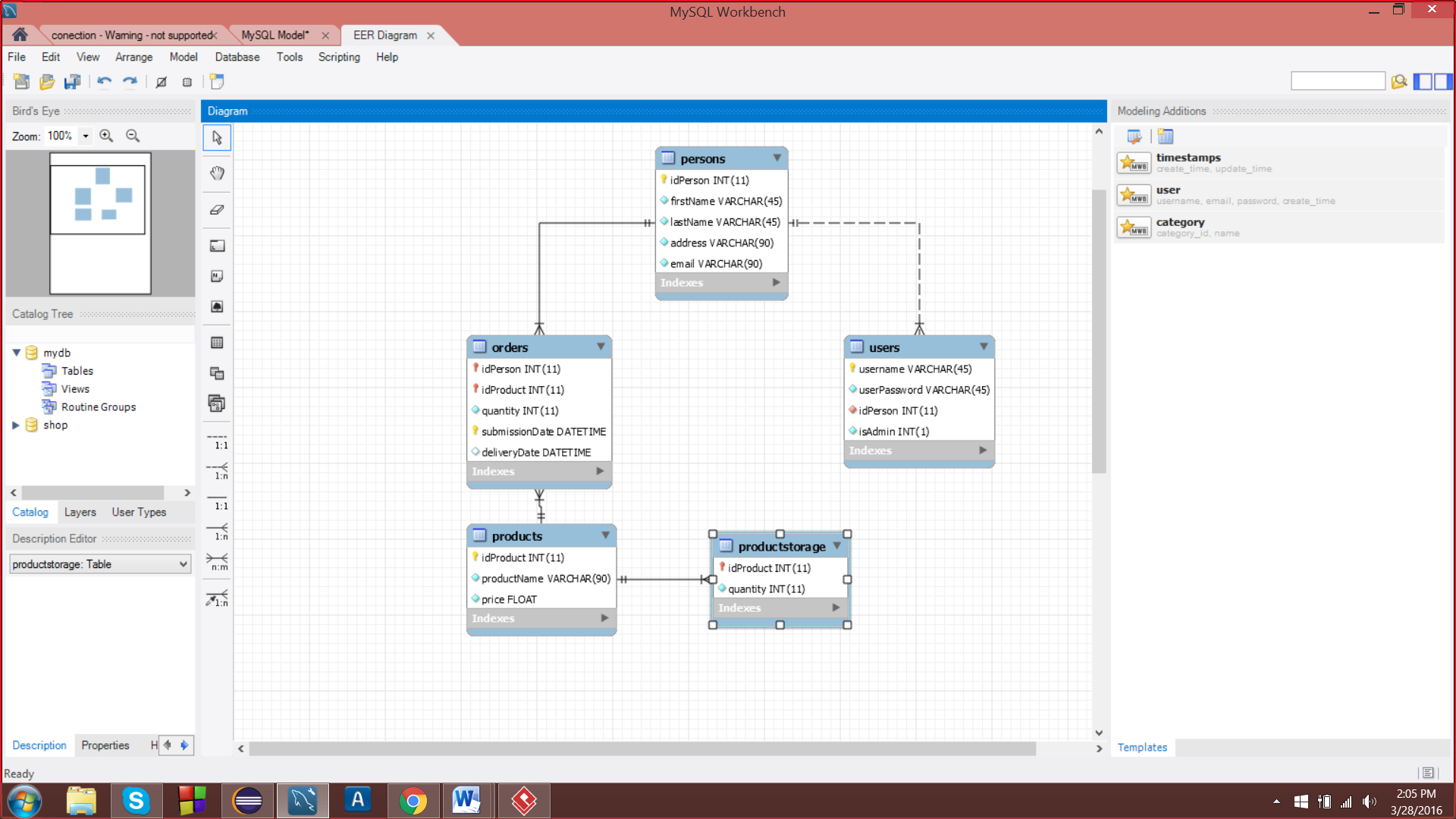
Model, however are only recipients of database table information, this makes them POJOs . Plain Old Java Object concept will be explained as well in the implementation chapter, which brings us to Controllers.

I might consider Controllers my bread and butter, therefore this project cannot have weak linking and classes have high cohesion. Data must come from the database and be placed in the model classes, from there the Controllers provide them to views and last, but not least views make connection with the user.

# Modeling

In order to keeps a basic, but strong structure for the data fetching operations part. An interface is needed that contains the basic methods, which will be implemented: getAll(), insert(), update(), delete().

Almost forgot about the design of the database:



Using Reverse Engineering tool in MySql Workbench, the afore presented diagram was obtained. The yellow key mark primary keys and the red one mark foreign keys. In database relational modeling and implementation, a unique key is a set of zero, one, or more attributes. The value(s) of these attributes are required to be unique for each tuple (row) in a relation. The value or combination of values, of unique key attributes for any tuple should not be repeated for any other tuple in that relation.

When more than one column is combined to form a unique key, their combined value is used to access each row and maintain uniqueness. These keys are referred to as aggregate or compound keys. Values are not combined; they are compared using their data types.

The key that is selected is called the primary key. Only one key within an entity is selected to be the primary key. This is the key that is allowed to migrate to other entities to define the relationships that exist among the entities. When the data model is instantiated into a physical database, it is the key that the system uses the most when accessing the table, or joining the tables together when selecting data. Therefore, unique key that has migrated to another entity is called a foreign key. Sometimes, in order to better order the data in the table we must define index.

An index is one way of providing quicker access to data. Indices can be created on any combination of attributes on a relation. Queries that filter using those attribute can find matching tuples randomly using the index, without having to check each tuple in turn. This is analogous to using the index of a book to go directly to the page on which the information you are looking for is found, so that you do not have to read the entire book to find what you are looking for. Relational databases typically supply multiple indexing techniques, each of which is optimal for some combination of data distribution, relation size, and typical access pattern. Indices are usually implemented via B- trees, R-trees, and bitmaps. Indices are usually not considered part of the database, as they are considered an implementation detail, though indices are usually maintained by the same group that maintains the other parts of the database. It should be noted that use of efficient indexes on both primary and foreign keys can dramatically improve query performance. This is because B-tree indexes result in query times proportional to log(n) where n is the number of rows in a table and hash indexes result in constant time queries (no size dependency as long as the relevant part of the index fits into memory).

Also, the constraints are important when updating, inserting and deleting data. To shorten things up, when having reference(s) to a primary key in another table(s) as foreign key and you want to delete a row in the first table, we must make sure we have deleted the rows in the other tables as well. Because otherwise we will have a MariaDB exception due to the restriction done by the foreign key existence and to the integrity of existing data. Is like having a bridge over a lake, if you cut a part out, for example the ropes, the bridge will fall, however if you make sure you cut wood steps one by one and rebuild it afterwards, you might still have a up and running bridge.

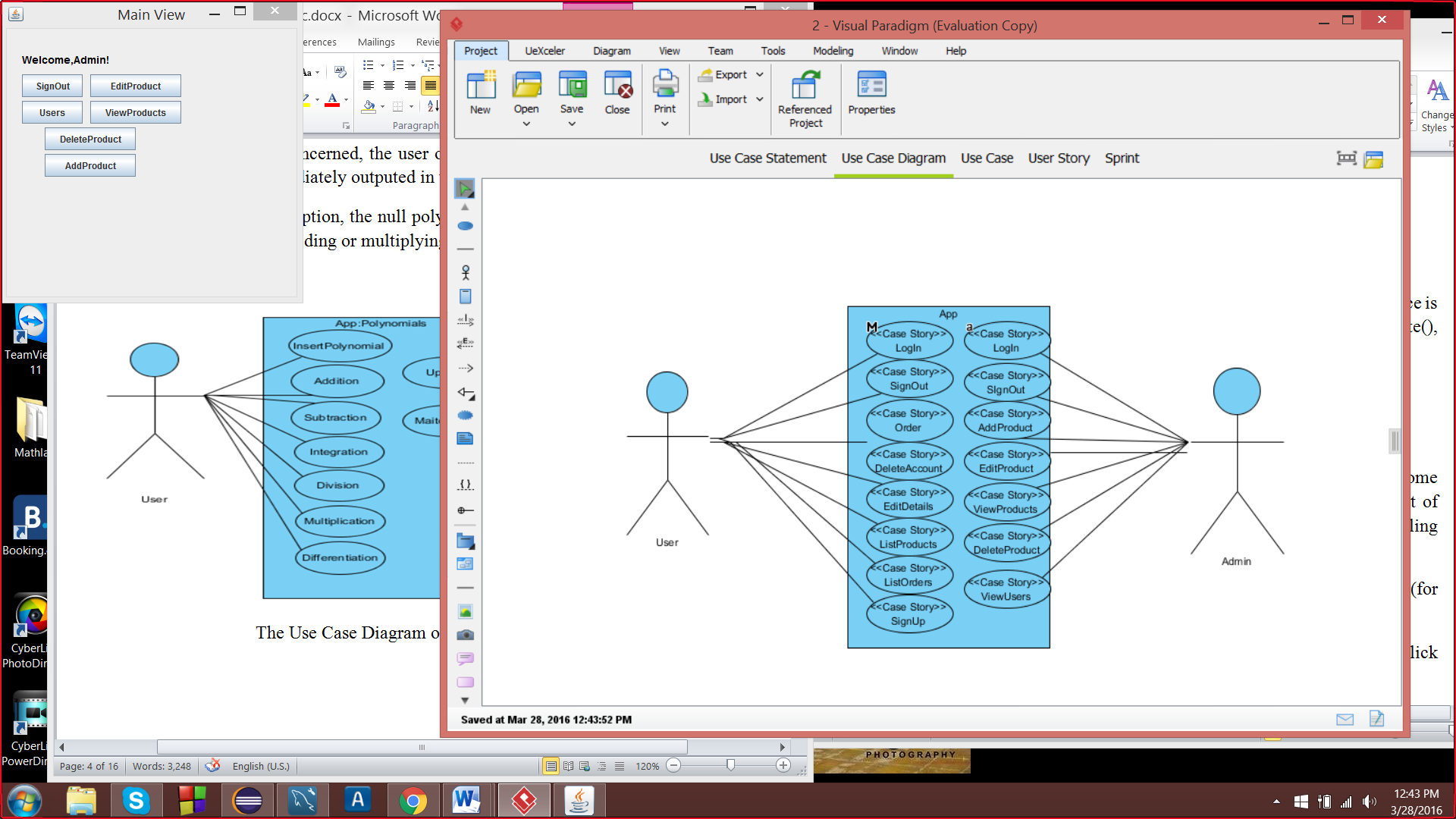
In our case we have the 5 tables presented in the diagram, which will be also POJO classes in our application, in order to have an easier way to store and manage data. They are not dependent of one another, but the BussinessLogic classes take in consideration the foreign keys constraints and the index contraints when operating with those classes.

## Use Case Diagram

This type of diagrams have an important role in showing how an user may interact with some features of a certain application. In software and systems engineering, a use case is a list of actions or steps, that define the interaction between a role and a sytem( in the Unified Modeling Language the role is also known as an actor) in order to achieve a goal.

In this case, there are many use case, this chapter will only display and explain some of them (for example placing and order, singing up and deleting account).

As far as the operation are concerned the, the user has to choose which respective button to click on and the result is immediately outputed in a textfield or a new View will be displayed.



The Use Case Diagram of the application

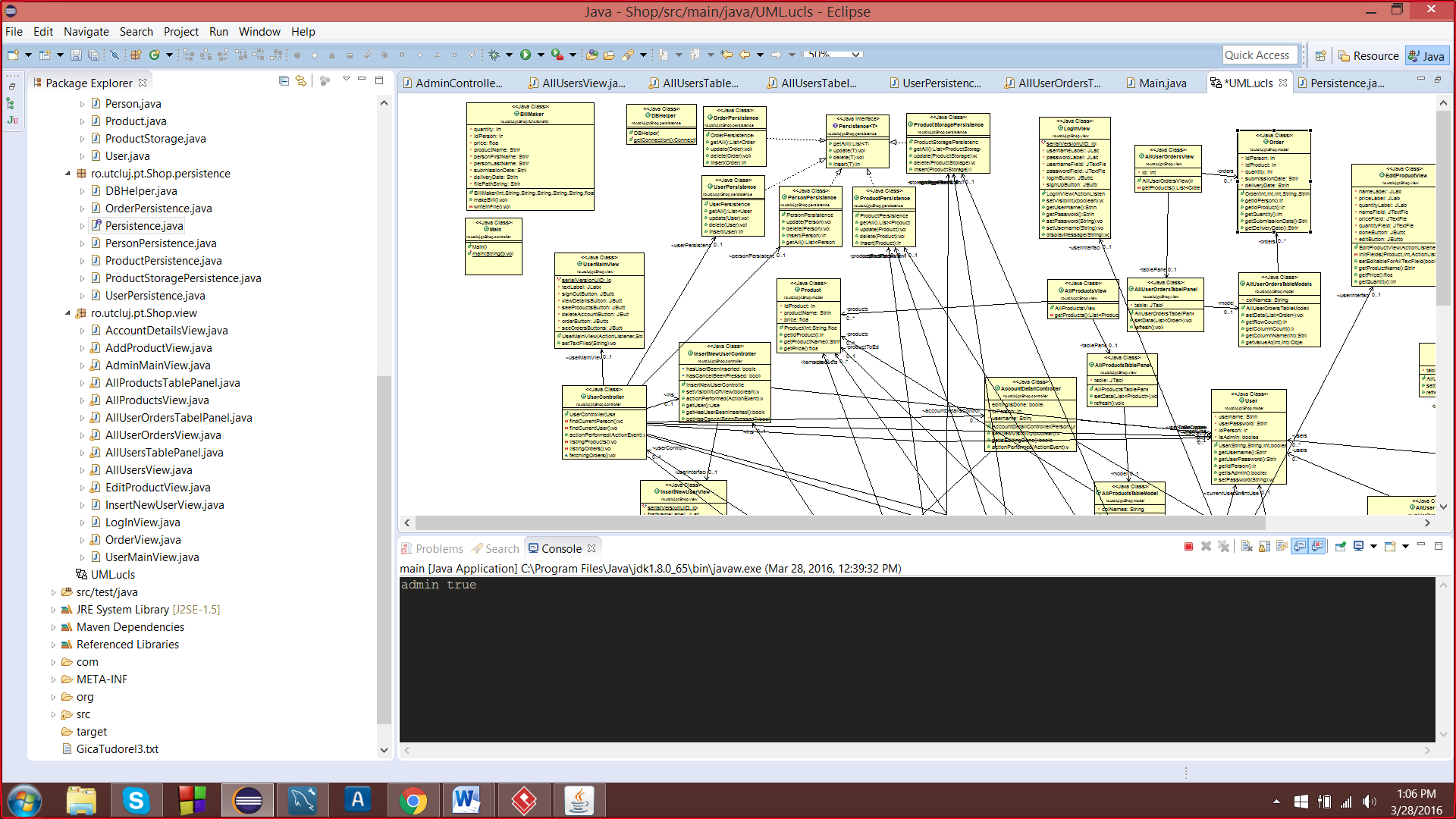
# Design

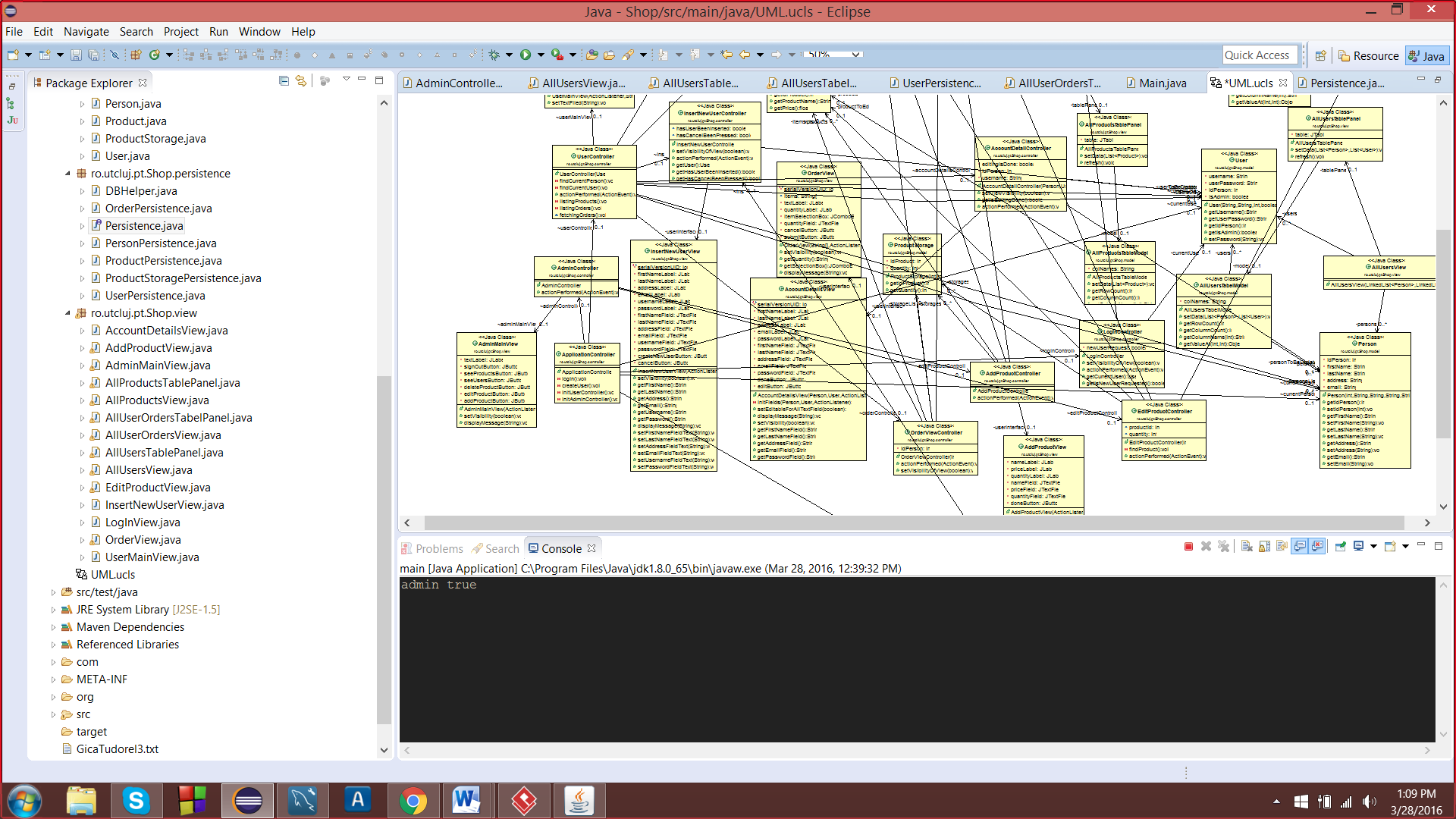
## Relational Diagram

The entity-relational model is a data model that describes aspects of a business domain or its process requirements, in such an abstract way that it can lead to being implemented in a database such as a relational database. The main components of ER models are entities and the relationships that can exist between them.

In most cases the ER design is the first step in solving any OOP problem. I have chosen to implement each operation into a separate class and also to add a class for testing. However, testing is needed only for verifying if the persistence classes work properly, because of the dimensions of the project and the number of operations done on the database, we must make sure that those classes are perfect. Also controllers, can have or not a hierarchic structures.

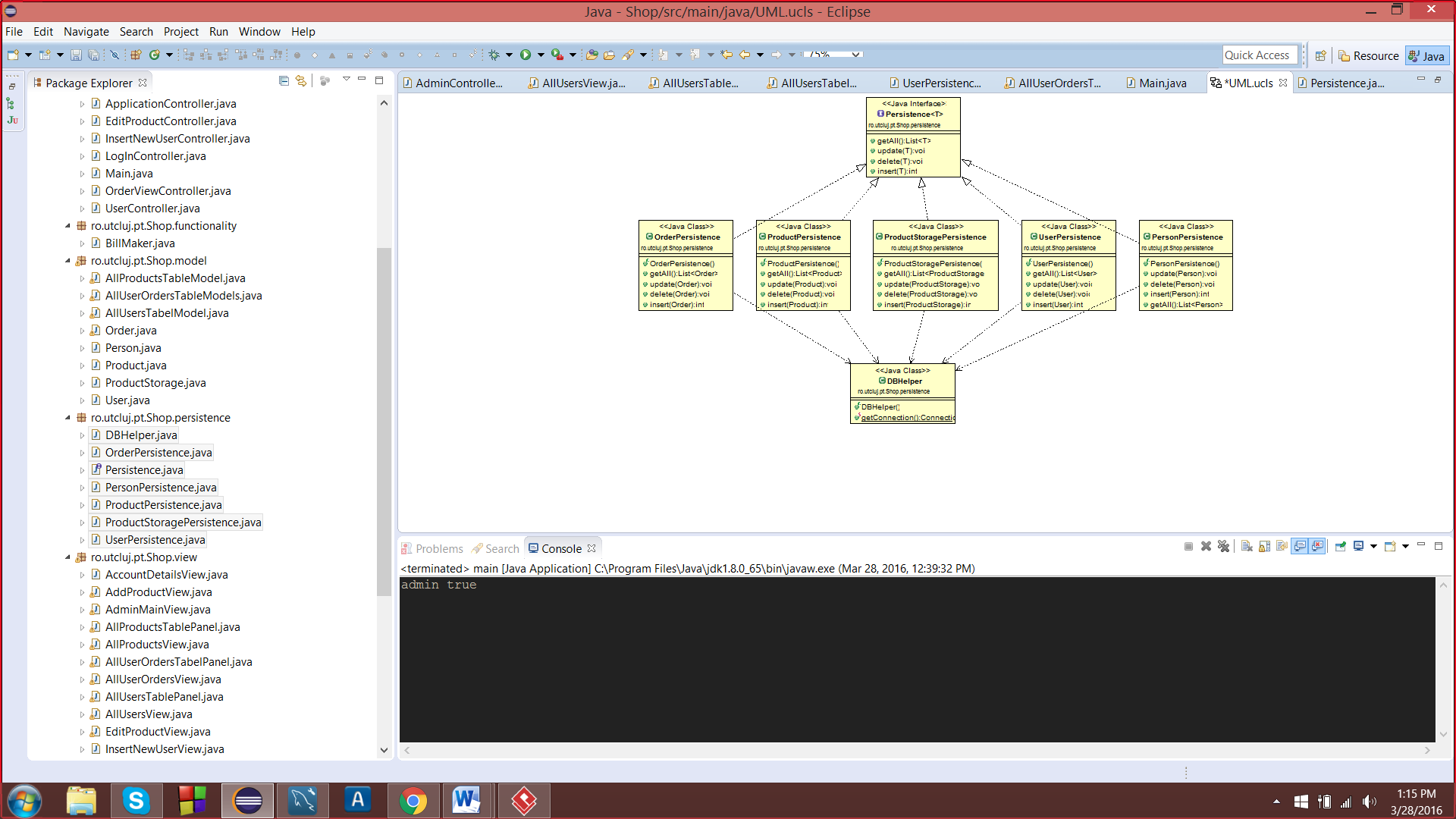
This is the relationship diagram containing associations and dependencies type relations:





Above are some screenshots of the UML diagram,however due to the number of classes it cannot be displayed properly.

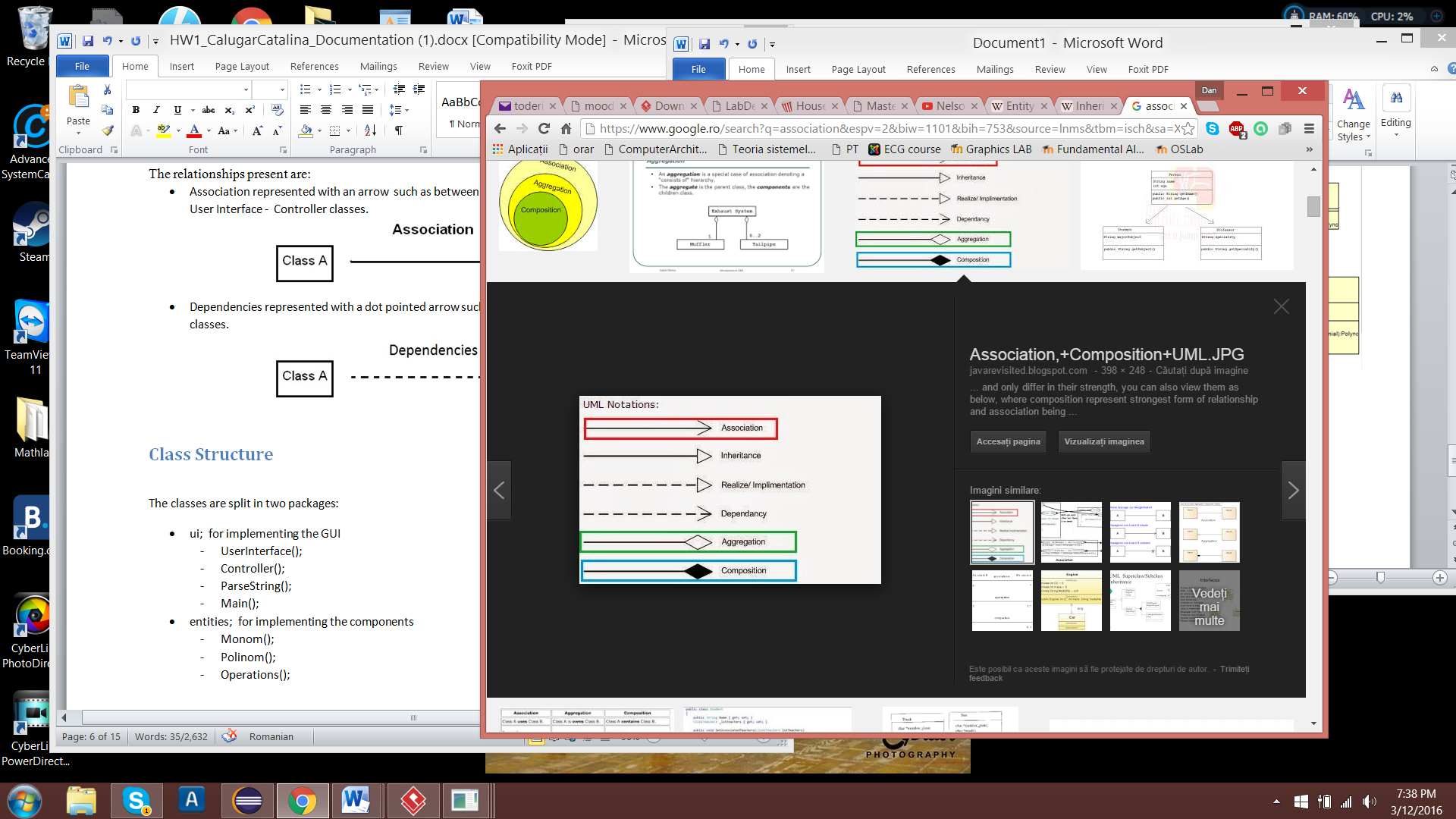
In addition, in this part of the diagram there is presented a special type of association that is called implementation and dependencies:



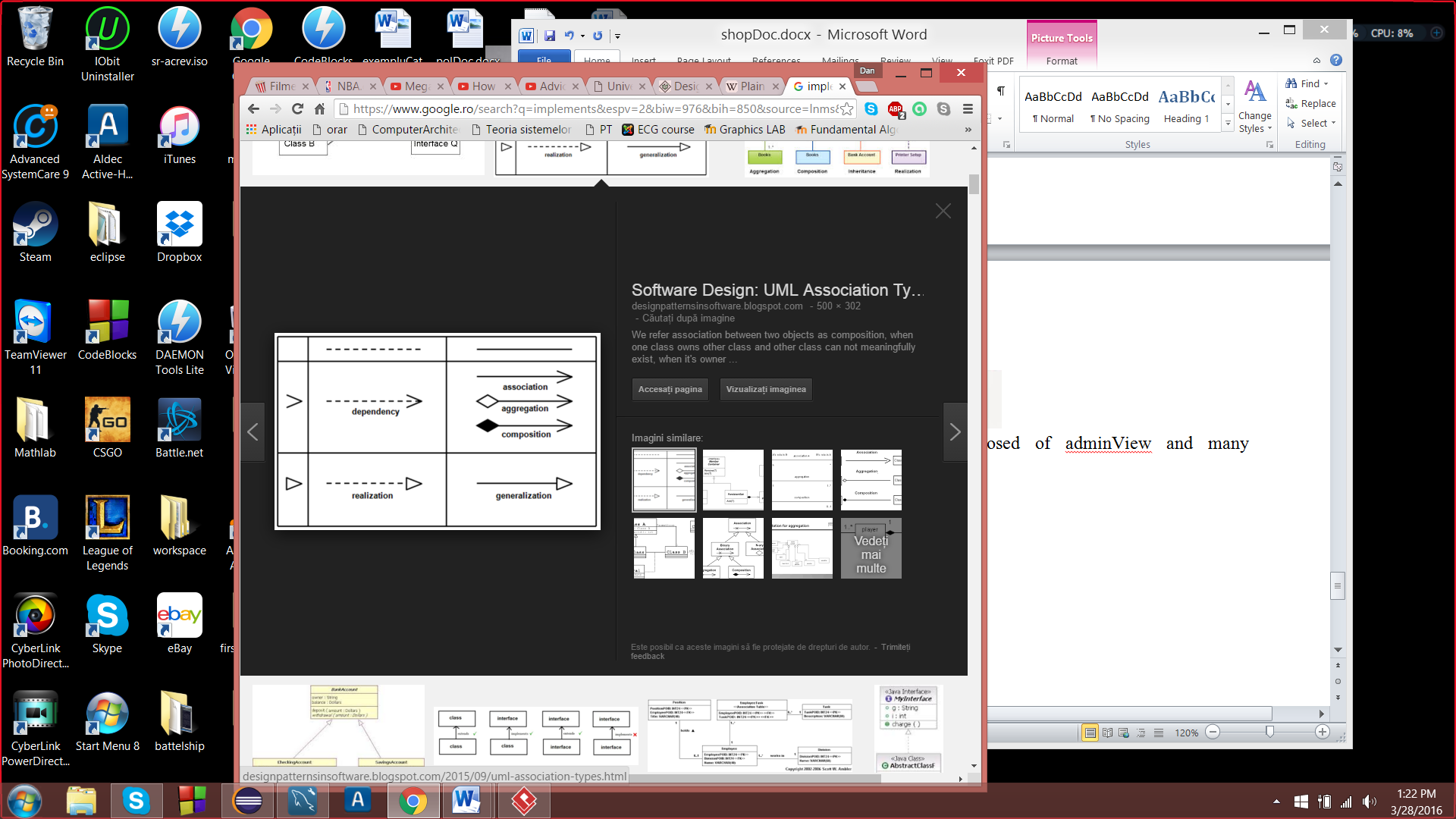
All persistence classes, implement Persistence<T> interface and they use a DBHelper class object to obtain a connection to the database and execute the SQL query.

The relationships that are shown in the diagrams are:

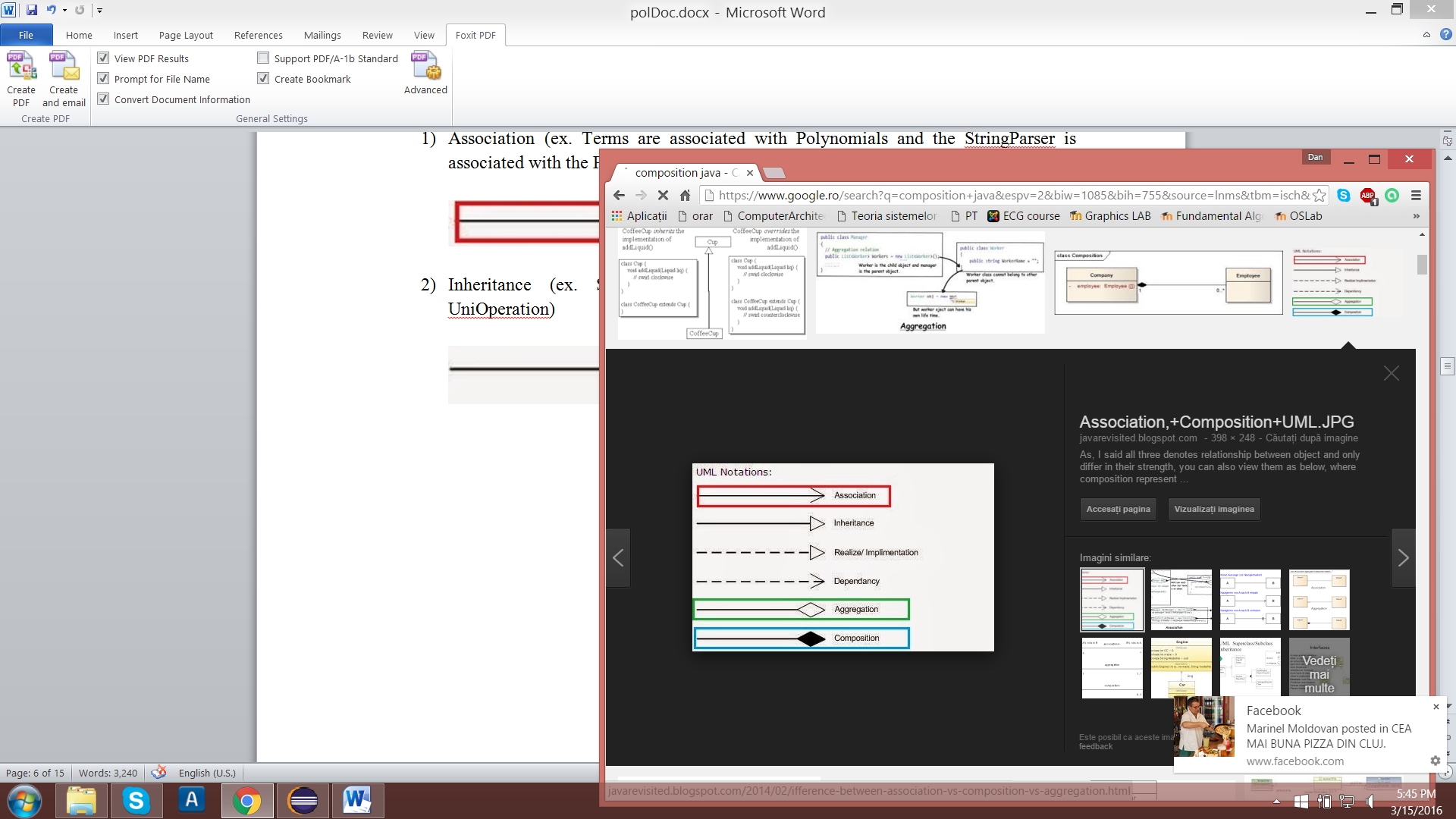
1. Association (ex. BillMaker is associated with the OrderController).



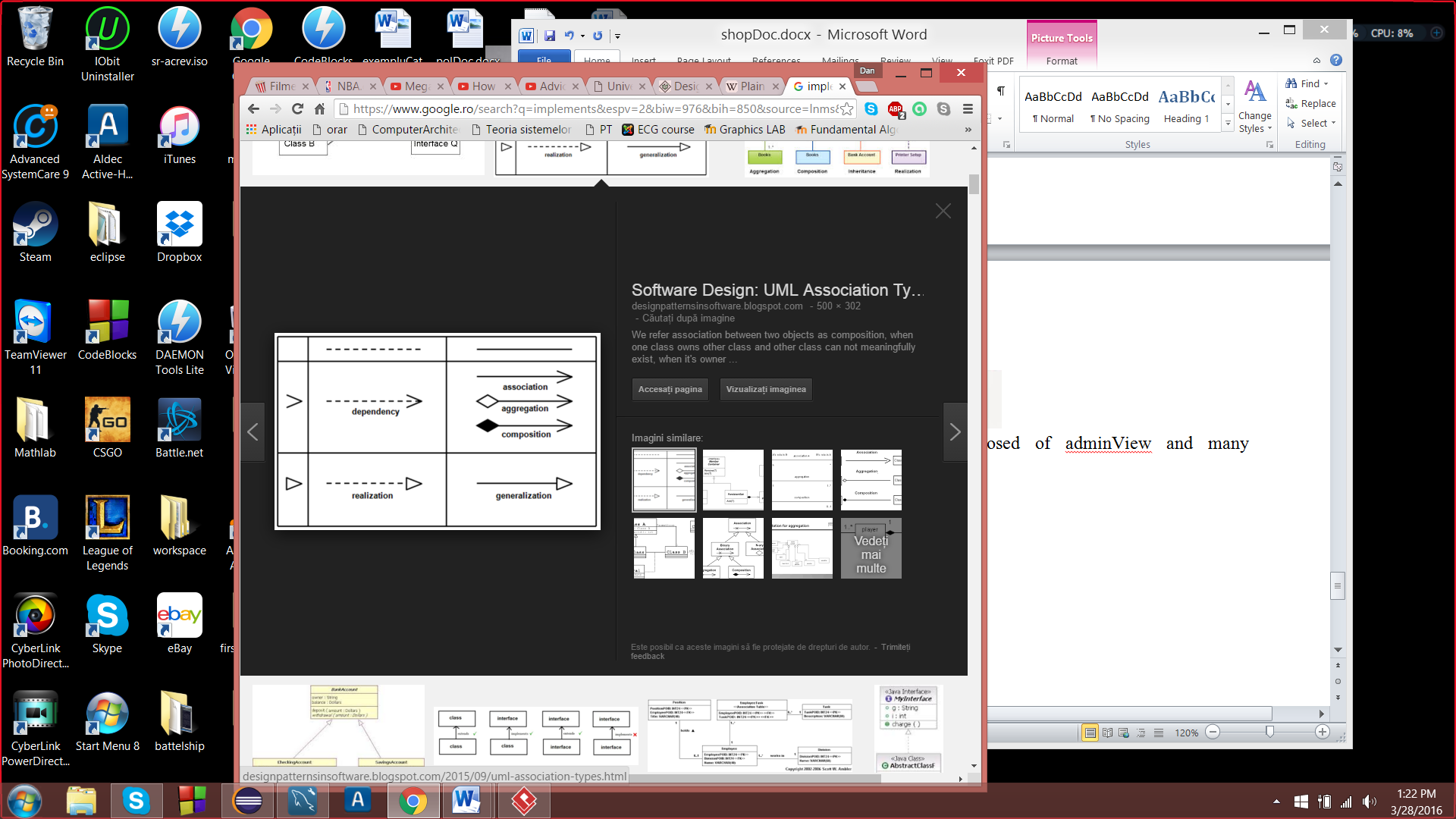
1. Implements (ex. ProductStoragePersitence and PersonPersistence implements Persistence)



1. Composition (ex. AdminController is composed of adminView and many controllers).



1. Dependency (ex. Persistence classes depend on the DBHelper)



## Class Structure

The classes are split into numerous packages:

1. ro.utcluj.pt.Shop.model (POJOs):

-Order

-Person

-Product

-Product Storage

-User

-All Products Table Model

-All User Orders Table Model

-All Users Table Model

1. ro.utcluj.pt.Shop.functionality:

-Bill Maker

-DB Helper

1. ro.utcluj.pt.Shop.persistence:

-Order Persistence

-Persistence

-Person Persistence

-Product Persistence

-Product Storage Persistence

-User Persistence

4) ro.utcluj.pt.Shop.view:

-Account Details View

-Add Product View

-Admin Main View

-All Products Table Panel

-All Products View

-All User Orders Table Panel

-All Users Orders View

-All Users View

-Edit Product View

-Insert New User View

-Log In View

-Order View

-User Main View

5) ro.utcluj.pt.Shop.controller

-Account Detail Controller

-Add Product Controller

-Admin Controller

-Application Controller

-Edit Product Controller

-Insert New Use Controller

-LogIn Controller

-Main

-Order View Controller

-User Controller

# Implementation

Firstly, let’s start by explaining what the packages main function is and how this packages work together to form this application.

## Model

A POJO is a Java object not bound by any restriction other than those forced by the Java Language Specification, but this is purely ideally speaking. POJO stands for Plain Old Java Object, and would be used to describe the same things as a "Normal Class" whereas a JavaBean follows a set of rules. Most commonly Beans use getters and setters to protect their member variables, which are typically set to private and have a no-argument constructor.

All our classes only have setters and getters and a simple constructor, nothing less and nothing more. Their properties are identifiable from the database EER Diagram that was presenting in modeling chapter.

## Persistence

Reiterating the purpose of a set of DAO classes, we must take in account what is this pattern all about. Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

**Data Access Object Interface** - This interface defines the standard operations to be performed on a model object(s).

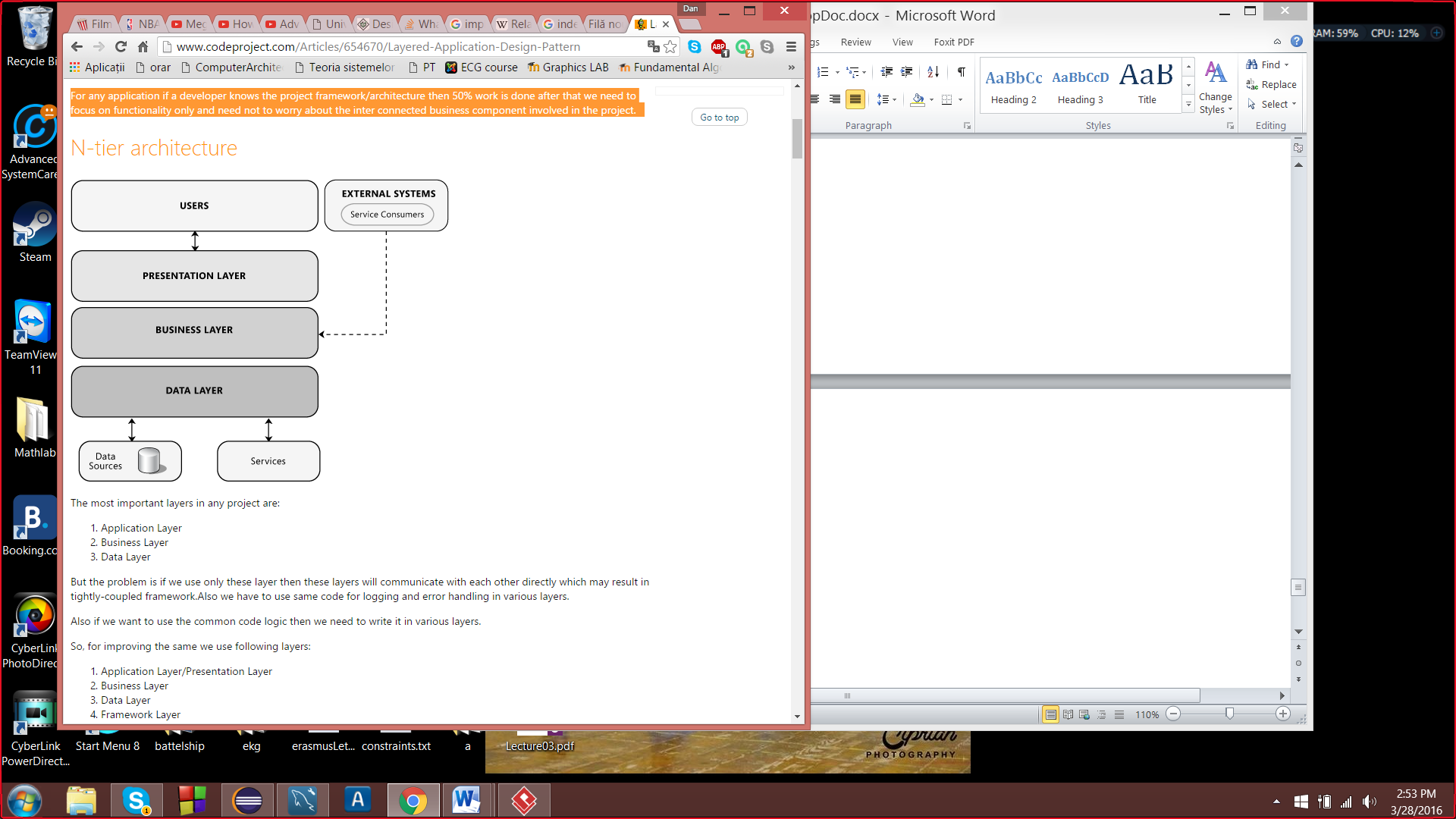
**Data Access Object concrete class** - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.

**Model Object or Value Object** - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

In this project, DAO pattern was the neck and bone of the modeling design process. The interface is called Persistence. Some implementations are PersonPersistence and ProductPersistence, which both fetch database data and transpose them into products or persons or take a person object or a product object and insert, update or deletes them from the database.

## Controllers and Views

In critical application development, the most important thing is project architecture or project framework which comprises of various modules involved in request processing or business handling. If the framework is clear and robust, and completely designed with proper planning then it’s very easy to do the code and develop the particular requirement for business.   For any application if a developer knows the project framework/architecture then 50% work is done after that we need to focus on functionality only and need not to worry about the inter connected business component involved in the project.



The most important layers in any project are:

1) User Layer (Views)

2) Business Layer (Controllers)

3) Data Layer (Persistence)

### Data Layer

This layer will contain model entities, Database connectivity and data access classes. Data access components in this data layer are responsible for exposing the data stored in these databases to the business layer.

### Business Layer

This layer contains the business processing logic.All the CRUD operations calling go here. First of all added DataLayer reference to Business Layer.In this Layer we are validating the credentials provided by the end user with the values in our tables.For performing any operation with database we use LINQ (Language Integrated Query) and for accessing the table entities we need to make an object of the connection entities(DBHelper).

Business Layer is like the messenger and chief between presentation and the data layer. It passes data and gives commands based on the business processing logic.The multitude of controllers that are presented in the controllers package form the business layer.

The controller is also its ActionListener, which means that when a button is pushed this component handles the event.Handling an event, means executing a set of instruction if the event occurs.

That brings us to a very known and used design pattern called Model – View – Controller (MVC).This is a software architectural pattern mostly (but not exclusively) for implementing user interfaces on computers. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. The three main components are:

1. The model manages the data, logic and rules of the application. It captures the behavior of the app in terms of its problem domain, independent of the user interface
2. The view only displays to the user based on changes in the model.
3. The controller accepts input and converts it to commands for the model or view.

### User Layer

This layer contains the presentation logic. The application consists of a series of frames with which the user interacts. Each form contains a number of fields that display output from lower layers and collect user input. The views form the User Layer.

Mostly the view implementations were quite simple; however there was a requirement which stated that JTable has to be used for displaying products, user details, orders, etc. That was the most difficult part, and after hours of research and tries a solution was found.

In order to have a good JTableView design, you must first implement 2 classes: the model for the JTableView and the specific panel.

The model must extend AbstractTabelModel and have an array of Strings that contain the headers of the table. Also the must implement methods specify, the data setup, number of rows, number of columns and returning the value at the corresponding index

The panel has nothing special, but must contain the model and the JTable object. Other than that, there is nothing more to it.

Finally, the View extends JFrame and therefore must setup the dimensions and has to have a reference to the date that will be put in the model.

## BillMaker

It is the only class that is put in the fuctionality besides DBHelper, because of its purpose. It receives a person object and an order object and has to create or append to a file having the name formed from the user name plus his or hes id. Moreover, having one simple method except to his constructor, it enables the writting of bills in that specific file.

# Conclusion and further developments

Arriving to the last point of this presentation, I personally considered this project a good exercise for many things, such as layered pattern design implementation, MVC design and DAO classes creation and usage. Throughout the process of developing this application, more than enough problems appeared regarding UI due to the number of views that communicate with the main controller.Splitting the main controller into UserMainController and AdminMainController was a good idea in managing specific controllers and views for each type of user, due to the fact that they had different options. Though, I firstly tried to implement a setVisibility() method that would help me in switching between views, however when putting it in Controller nothing seemed to work. I left my thoughts behind and I have fallen in the disgrace of opening each view and setting setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE).

Having more than one problem in hand, the persistence classes came as a relief and helped me a lot in implementing my functionalities. In addition, I could have done many more functions based on such architecture, but maybe in the near future or in another project.

Some improvements and future development plans would be:

1. Check the formats of the inputted data if they have the right format, using regex statements or simple methods.
2. Creating a server, for more than one user to use this application at a time and for real time data to be transmitted
3. Have a multi-thread design for fetching and storing data
4. Design a better user interface with much more functionalities and prettier view
5. More notifications for the user, based upon his or her actions.

# References

<http://www.codeproject.com/Articles/654670/Layered-Application-Design-Pattern>

<http://stackoverflow.com/questions/12517905/what-is-java-pojo-class-java-bean-normal-class>

<https://en.wikipedia.org/wiki/Plain_Old_Java_Object>

<https://en.wikipedia.org/wiki/Relational_database#Constraints>

<https://en.wikipedia.org/wiki/Unique_key>

http://www.tutorialspoint.com/design\_pattern/data\_access\_object\_pattern.htm