|  |
| --- |
| Order Management |
| Tatar MaraProgramming TechniquesApril 9, 2016 |

# Abstract

Consider an application OrderManagement for processing customer orders. The application uses (minimally) the following classes: Order, OPDept (Order Processing Department), Customer, Product, and Warehouse. The classes OPDept and Warehouse use a BinarySearchTree for storing orders.

* Analyze the application domain, determine the structure and behavior of its classes, identify use cases.
* Generate use case diagrams, an extended UML class diagram, two sequence diagrams and an activity diagram.
* . Implement and test the application classes. Use javadoc for documenting the classes.
* . Design, write and test a Java program for order management using the classes designed at question c). The program should include a set of utility operations such as under-stock, over-stock, totals, filters, etc.

# Problem analysis

The application was truly more demanding for me as compared with the first one,mainly because I had to use Serialization and Deserialization.Also,it required two perspectives:the common user or customer and the administrator of the site.When I was a kid I used to imagine that I had a clothing store,so this was my chance to implement this.A real clothing store would however demand a database,but this one follows the behavior of a real store quiet well.

From the user point of view,the scenarios are the most common ones.The user has to register with a name (aka username) and a password.Once registered,he will login with that name and the corresponding password.The range of products he can choose frome is relatively small,because there is no link to a database and hard coding of the items cand be rather inefficient.Also,hard-coding makes the code harder to follow.That is why I have chosen to serialize and deserialize the data.After the user decides on a product,he can add it to cart .After multiple products were selected he can finally place the order,which will be serialized in a file in the working directory.

From the manager point of view,the application has to provide some complex features.The administrator has to login in ,after he has selected the “admin” propriety.After this,he will be prompted in a window where he can add products to the stock.The only condition is that the ID of the product added should be unique.He will add the item,the size,the color ,the price and the stock.By stock we understand how many items of that type do we have in the warehouse.He has also the ability to update the stock of a certain item,to delete an item of hiss will.All data will be ,like I have previously mentioned,saved in specific files. The administrator can switch back and forth between the view of warehose and the view of orders.He can finish an order,whose correspondent in real life would be that of forwarding the order to delivery.

In order for all the features that are described above to be implemented correctly, a clear and intuitive GUI would come in great help for both the user and the manager to use the application in the most efficient and useful way.

# Modelling

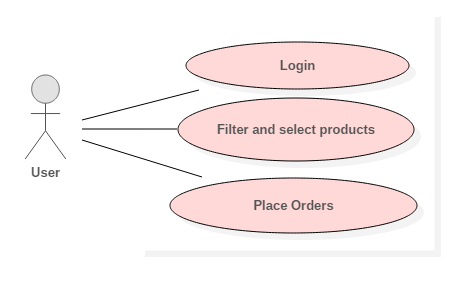
The main models we deal with in the application are the user( who can be either a cutomer or an admin), the products which are stored in a warehouse (represented by a binary search tree) and the orders ,organized in a Order Processing Departament(also a binary search tree).

# Scenarios ,Use cases

There are two main scenarios in which this application can be used (taking into account the user privileges – we can see our user as a normal customer or, otherwise, as a manager of our store. The manager, obviously, has many more abilities in the application compared to the client.

## Use case diagrams

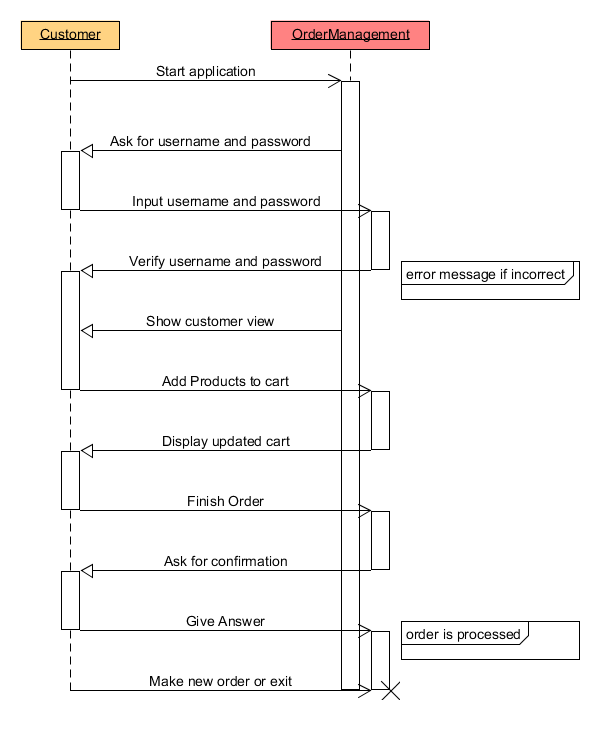
### User Use case

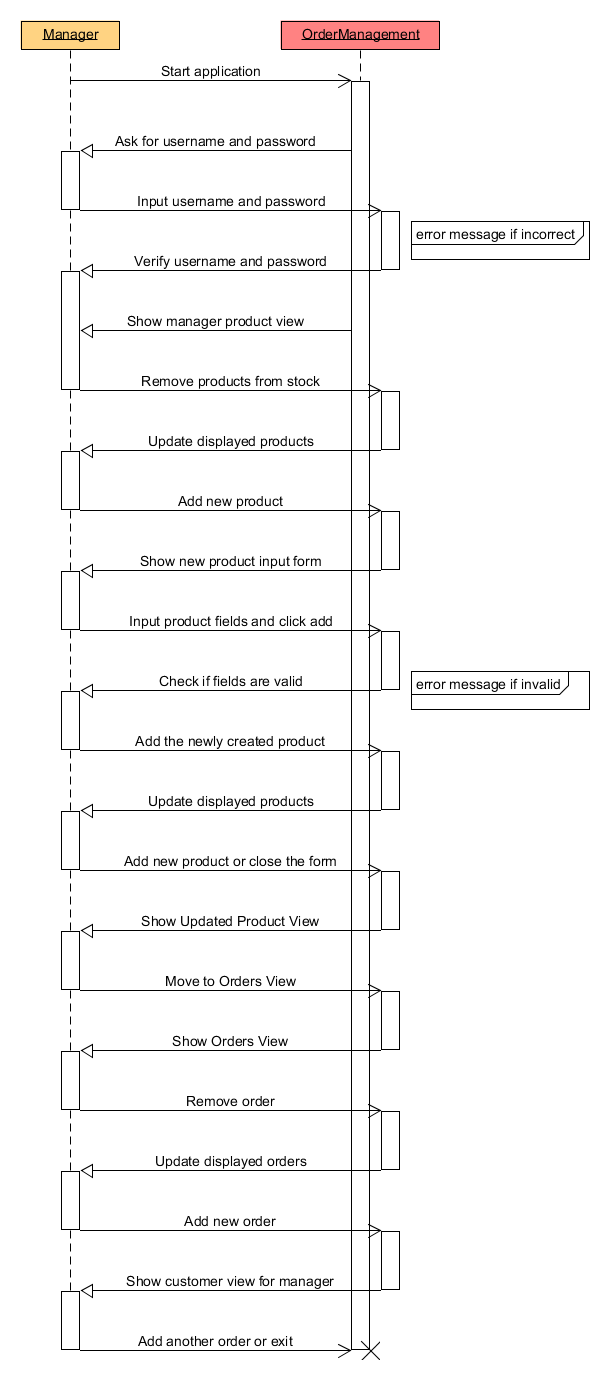


### Administrator Use Case

## C:\Users\marat\AppData\Local\Microsoft\Windows\INetCache\Content.Word\UseCaseDiagram2.png

### Sequence diagrams

****

****

## Description of scenarios

### The Customer

The Customer has to register first,if he has not previously registered.If he has already registered,he can proceed to loggin in.If however he tries to log in before registering,a message will be prompted saying that he has to register first.

After successfully logging in,the customer will select an item of his interest,the color and size and the image of the item will be displayed in the left side of the frame.On the left are the drop down filters that helps the user narrow his choices.

If he decides to purchase the item,he will press add to cart.If the item of his will is not on stock,a message will prompt notifying him.

After finishing shopping,the user can view the total sum at the end of the application and press the button place order,which will immediately register his order.

He is prompted to a window that says “Thank you for ordering”.

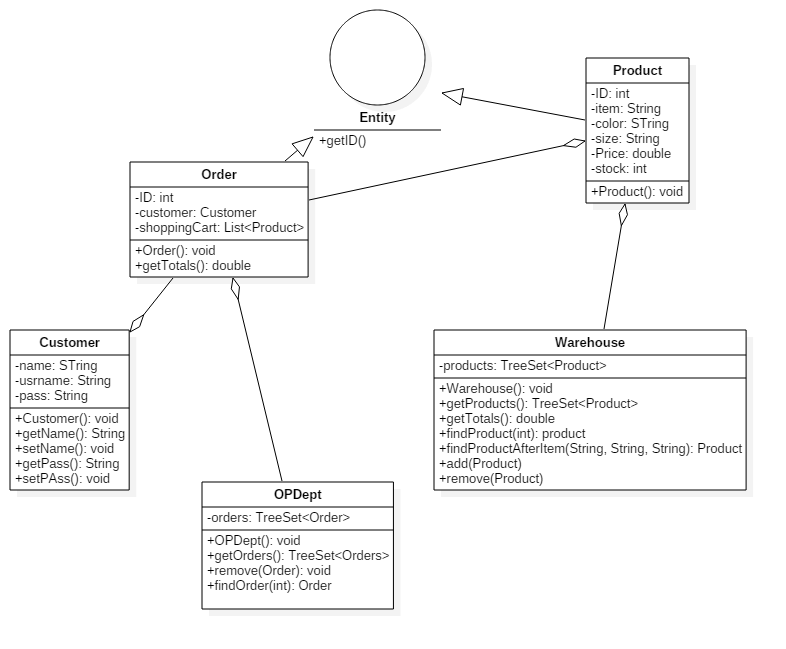
### The Manager

The manager has to log in as well(using admin username and admin password).If the username and password are correct,he will see the products that are currently in the warehouse.If he decides to see the orders,he will press the Ordrs button.He can switch back anytime to see the products in the sock.

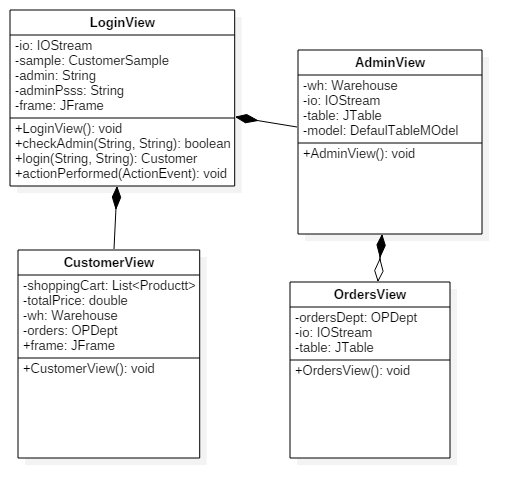
1. The Products View:the manager can see all the available products in the table .For each type of product,the ID,the type,the color and the size are displayed in the table.No two items with same ID can be introduced in the table.The manager can add a product of his will or delete a product from the table,and implicitly from the stock.He can also modify the stock of the items by selecting the row he wants to modify,modifying in the corresponding text area and then pressing the Update button.
2. The Orders View:the manager can browse through all orders and see the order ID,the customer that made the purchase and the item he bought .For each item a new row in the Orders table is displayed.The administrator can finish the order,meaning sending it to the delivery department .He can also switch back to the products view.

# Design

## The UML Diagram for the model



## The UML Diagram for the view



## Sequence diagrams

### Communication between the customer and the application

## Activity diagram

# The Packages

The project is designed according to the Model-ViewControl design pattern. That’s why we have three packages:

- models – this package contains the entities, on a logical level, with which our application works. It directly manages the data, logic and rules of the program. It includes the abstract classes User, Node, Binary Search Tree,

- views – this contains the GUI, the graphical interface that the user sees and uses to input data and retrieve information from the application. This is completely separated from the model (a new kind of GUI could be implemented on the same model) but it is merged together with the “control” part – which ties together the model and the view. It instantiates objects from the model and adjust the view according to them – this was made concurrently with the „view” part

- sample – in this package I added the hardcoded data with which the program starts. This facilitates the simulation of a real world store – it is in fact a series of factories of objects that populate the application – Sample Data – which creates Customers.

- services– in this package I added the Serialization and Deserialization services. In orer to keep the data constantly updated and make it easier to process orders and customers,I have used the java interface Serializable.This package was named services mainly for further development ,in which we coul add more services to our aplication.

-comparators- the comparators package contains the comparator used by the Tree Set in order to insert the products and orders in the Tree Set by their ID’s.The IDComparator ovverrides the method ComparTo in order to rightfully complete the tree.

-views- contains the classes related to the interface of our application. The user will face the Logger View and the User View,and the administrator will interact with three views,the Logger View,the Admin View and the Orders View. I have organized this in another package to follow the Model –View-Control design pattern.

# The classes and main structures used

## Models

The **Entity** abstract class contains a single attribute ID ,and setter and getter methods for that particular ID. The reason od existence of this class is that any Product and Order must have an ID.Also,if we implement this class, we will only need one comparator class to help place properly the elements in the Tree Set. This is the main motivation behind using this class ,however it has proven to be more efficient as an interface,as it does not display properly the IDs when it is an abstract class.

The **Product** class models the products and implements the interface Serializable.It also extends the Entity abstract class and therefore inherits the private attribute ID and the methods related to it,getID and setID ,respectively.The rest of the attributes are all private and it has to provide the follow characteristics of the product:the type of the clothing ,represented by the attribute “item”,the size of the product(could be either S,M or L) ,the color of the product ,the price per item and the stock ,or in other words the number of items of that particular type.For each attribute, the class provides getters and setters to retrieve the values from other classes.These attributes are saved in files in order to be always kept updated and in order to avoid hardcoding the data.

The **Order** class models the order made by a customer – hence, it has as attributes a customer, a set of Products that were added to the Shopping Cart and the date. There are methods to add a Product to the Shopping cart, to get the total amount of money that must be payed for all the products in the Shopping cart, a method to remove from cart, a method to get the whole Shopping Cart, getter and setter for the customer and the date, and also a overriden toString method to display the content of the Shopping Cart (the product’s brand, name and price) to the customer. This class also implements Serializable and this assures that all the data is saved in a file and kept up to date.

The **Warehouse** class is used to store all the products in a Binary Search Tree. I have provided it with a private Tree Set as an attribute.The Tree Set uses Product as a generic and adds the products in the tree by overrding the CompareTo method in the IDComparator class with a method that places the items by the order of Ids.I have added methods to add and remove a product to the tree,using the methods provided by the Tree Set Collection.Two additional methods ,findProduct and findProductAfterItem help us find a certain product in the tree,either by searching by the ID or by the item,size and color ,as a group.In both methods,the ID and the prev mentioned group of attributes identifies an unique product in tree.

The **OPDept** class is used to store all the Orders in a Tree Set. I have provided it with a private Tree Set as an attribute.The Tree Set uses Order as a generic and adds the orders in the tree by overrding the CompareTo method in the IDComparator class with a method that places the items by the order of Ids.I have added methods to add and remove a order to the tree,using the methods provided by the Tree Set Collection.The fidOrder method finds a particular Order by searching by its ID.

The **Customer** class also implements Serializable in order to keep its data up to date.It contains tree instance attributes ,all of the private, a name ,an username and a password.For simplicity,I have assumed that username and name are the same,however this feature can be further developed in order tomake the application more user friendly (for eg,once the user logs in with his username,a welcome message containg his name could be displayed) and to assure the security of data.It also contains getters and setters for each attribute.

The **Runner** has the purpose of a main class.1 1 1 1 1 1 1 1 11 1 1 1 1 11 1 11 1 1 1 11 1 1 11 1 1

## Comparators

The IDComparator class implemets both Comparator restricted to Entity type of objects interface and the Serializable interface. It has to override the compare method,in order to rightfully insert a Entity object in a Tree Set.The compare method is often ovveriden with this purpose when using Tree Sets.1 1 1 1 1 1 1 1 1 1

## Services

The SerializationService deals with keeping data up to date and avoiding hardcoding the data.We are mostly interested in keeping Customers,Products and Orders updated,therefore the class contains methods that deal with the serialization and deserialization of each of the object classes. **Serialization** is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called **deserialization**.

A class must implement **Serializable** interface present in **java.io** package in order to serialize its object successfully. **Serializable** is a **marker interface** that adds serializable behaviour to the class implementing it.The files used in this puropose of serializing/deserializing data are “warehouse.ser”,”orders.ser” and “customers.ser”.1 1 1 1 1 1 1 1 1 1 1 1 1 1

## Views

The **LogIn View** creates the visual login screen that all the users see. It has two input boxes for username and password and contains method that check the validity of this information with the users from CustomerSample class.The CustomerSample class contains an arraylist of users and their passwords and when the user enters his username and password it check the consistency of those input data.If they are not consistent with some data from the arraylist,the app asks for registration.When registering,the input data is stored in the array and later serialized to remember the user at the next login.

The **CustomerView** creates the GUI that the user sees after login in.As for the other interface classes,I have used Java Swing tools to create this GUI. Swing API is set of extensible GUI Components to ease developer's life to create JAVA based Front End/ GUI Applications. It is build upon top of AWT API and acts as replacement of AWT API as it has almost every control corresponding to AWT controls. Swing component follows a Model-View-Controller architecture .This class contains numerous methods and components in order to obtain the desired view.

The **Admin View** correspond to the administrator view.It contains mainly a table where the administrator can add products,delete products or update the stock of a certain item. The table is nicely organized and was implemented with a JTable offered by the Swing API.After each modification of this table that initially contains the data retrieve from the files,using deserialization in the Admin View constructor,the data also has to be updated in the files,and here the serialization comes in and does this part.In the Admin View it is the warehouse that needs to be deserialized in the constructor and later uopdated in the files using the serialization service.This view also has a Order button which leads us to the next class,the Orders View.

The **Orders View** also corresponds to the administrator view.It also contains a table,but this time the attributes of the administrator are restricted.He can only finih a method,aka delete it.This class also requires the serializing and deserializing methods,because when opening the view,it has to retrieve orders data from the “orders.ser” file and later update if an order has been deleted using serialization.

## The main structures used

In this project I have used Java Collection classes such as Array Lists and Tree Sets.

The class **Tree Set<E>** provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

I have used the constructor **TreeSet(Comparator comp)** which constructs an empty tree set that will be sorted according to the given comparator.I have used the predefined methods **void add(Object o)** ,which adds the specified element to this set if it is not already present, **boolean remove(Object o)** ,which removes the specified element from this set if it is present and **Comparator comparator()** ,which returns the comparator used to order this sorted set, or null if this tree set uses its elements natural ordering.

The **ArrayList** class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

# Things I have leaned

This project was a lot more complex than the first one. I had to deal with two different instances of a user and to think about the problem in two ways, which was interesting. Also, it was nice to create an application that does many things – I think it’s the most complex application I’ve realized to date and I’m happy I pulled it off.

5.1 Things I learned

- I learned (or re-learned) to work with Tree Sets

- I learned to use the Serialization interface,which has proved to be a lot more complicated than I previously considered,because it has constantly thrown errors ar run time that I barely managed to solve.

- I learned to better manage the GUI part (Especially the repaint function which was a bit of a headache before)

- I learned to work with a more complex data model, with abstract classes

- I learned to make better use of generics..in general.

5.2. Future improvements

- showing product information with hyperlinks

- adding pictures to products in a more efficient way,that it permits adding more images and having less code;

- a feature to compare the specs of two products

- filters applied simultaneously

- many other small tweaks and improvements that I haven’t got the time to do