

Homework 3

Order Processing

Buzea Vlad-Calin

30422

Table of Contents

1. Objective of the assignment.....	3
2. Problem analysis.....	3
2.1. Modelling.....	3
2.2. Use cases	4
2.3. Scenarios.....	4
3. Design.....	6
3.1. UML Class Diagram	6
3.2. Data structures	9
3.3. Algorithms.....	10
3.4. Class design	10
3.5. Interface	13
4. Implementation and testing.....	17
5. Results.....	18
6. Conclusions, what has been learned, further developments	19
7. Bibliography.....	19

1.Objective of the assignment

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.

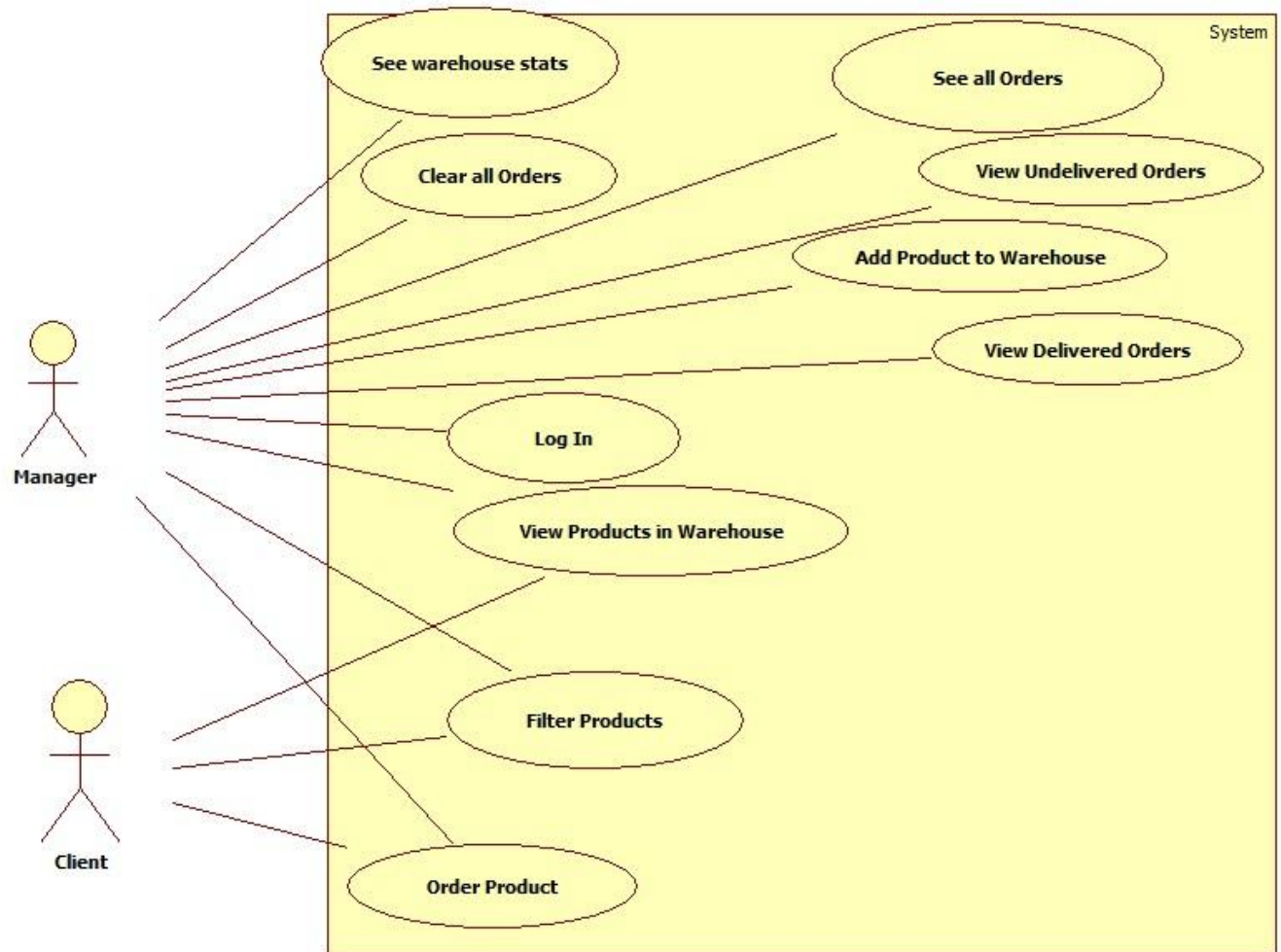
- a. Analyze the application domain, determine the structure and behavior of its classes, identify use cases and generate use case diagrams, an extended UML class diagram and two sequence diagrams.
- b. Implement the application classes. Use javadoc for documenting classes.
- c. Implement a system of utility programs for reporting such as: under-stock, over-stock, totals, filters, etc.
- d. Write the appropriate test drivers.

2.Problem analysis

2.1 Modeling

Order Management refers to storing orders and performing operations on them. Of course these are a virtual representation of a real world operation. A virtual product listed in the application is a real object and a Customer registered is a person that requires a service. He can select his product from a list of products. Of course, they have to be physically stored in a warehouse, that has a stock of each product.

2.2 Use Cases



2.3 Scenarios

Scenario I

a) Identification summary

Title: Addition of a product in the warehouse by the manager

Summary: This use case allows the manager to specify all the attributes of a certain product and then add it to the warehouse

Actor: manager

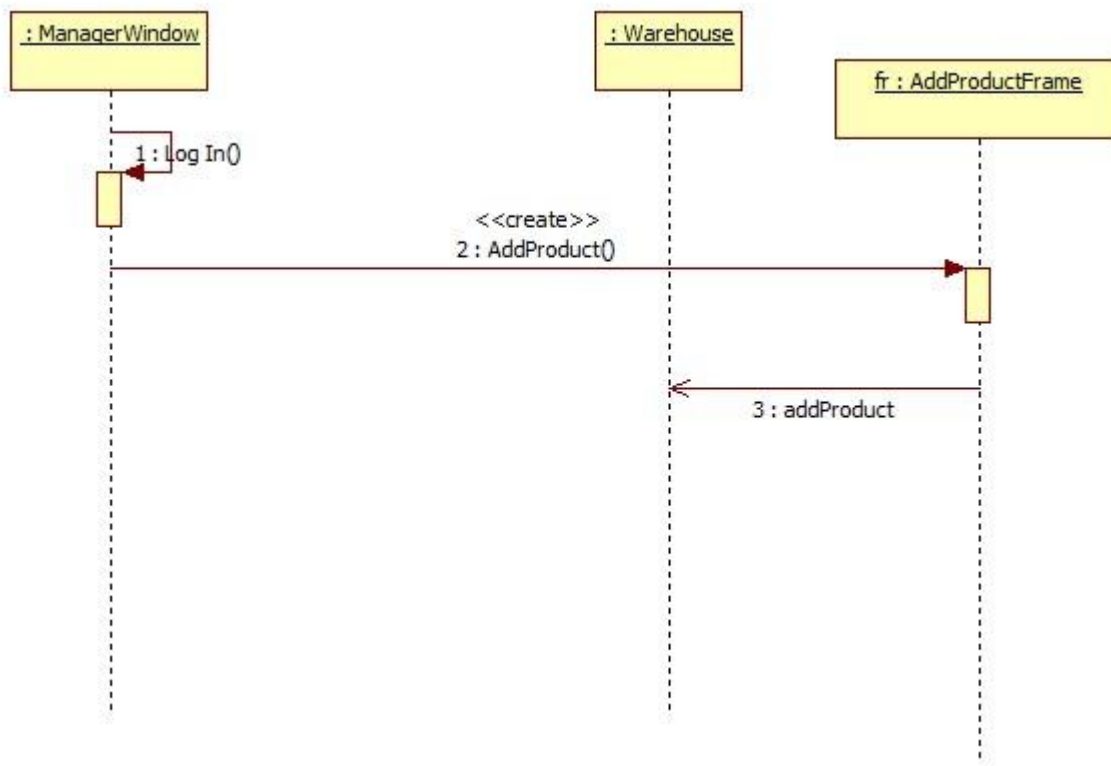
b) Flow of events

Preconditions: The user interface did not malfunction and the user input data was correct.

Main success scenario:

1. The manager starts the Order Management application and selects „Manager Log In”.
2. The application requests a password for authentication.
3. The manager introduces the password.
4. The application verifies correctness of input.
5. The application offers the manager interface .

6. The manager selects „Add product to Warehouse”
7. A new window opens and the manager introduces the name, producer and price of the product.
8. Manager introduces stock corresponding to product and clicks „Submit”.
9. The application verifies correctness of input data and adds product to warehouse if not already in the list.
10. If the product already exists, then the stock in the warehouse is updated.
11. The manager can now choose to add another product (go to step 6) or log out.



Scenario II

a) Identification summary

Title: Placement of order by the customer

Summary: This use case allows the customer to specify all the attributes of a certain product and then place an order containing that product

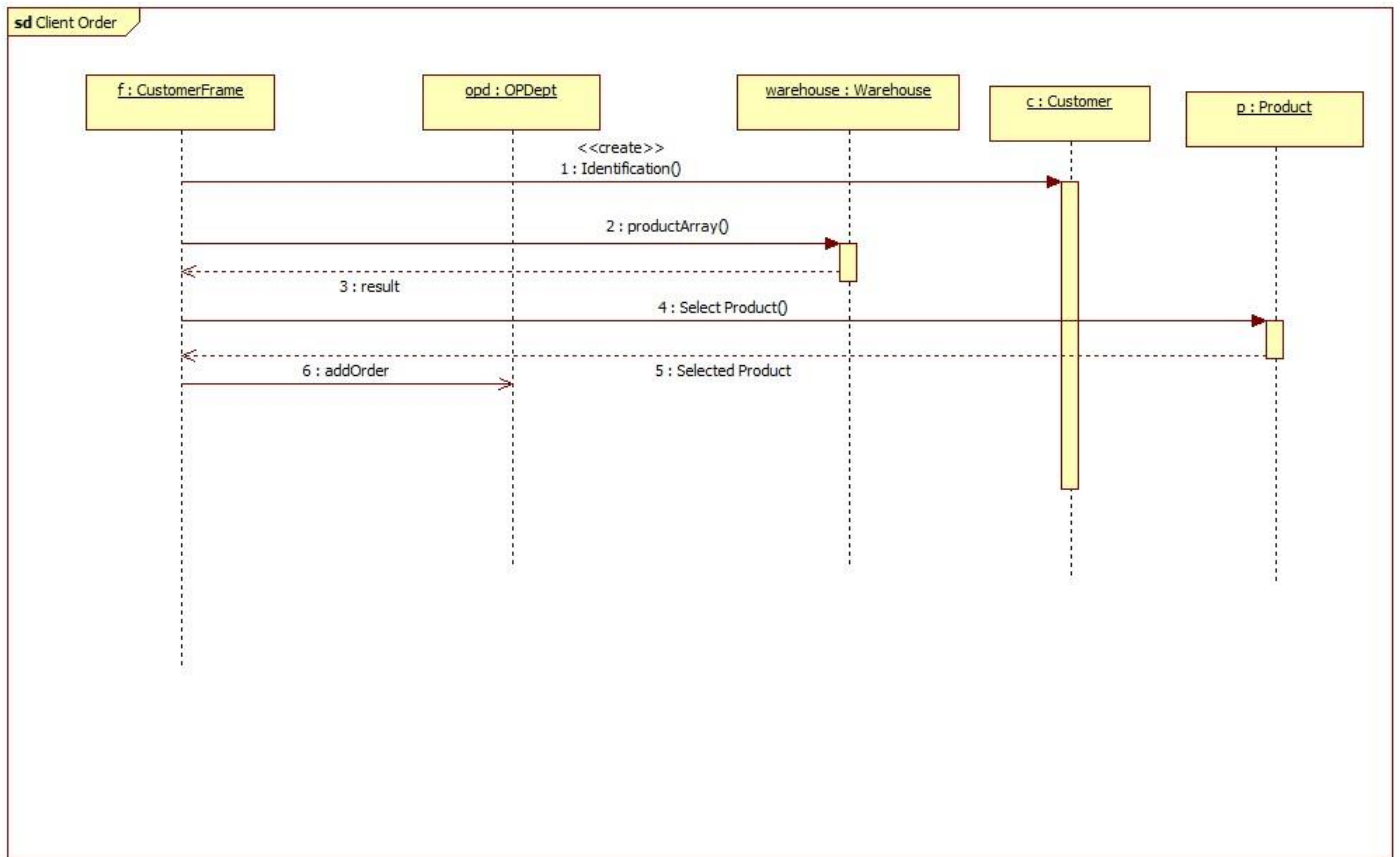
Actor: customer

b) Flow of events

Preconditions: The user interface did not malfunction and the user input data was correct.

Main success scenario:

1. The customer starts the Order Management application.
2. The customer selects „Order Product”
3. The customer introduces the SSID, First Name, Last Name, and Address(optionally email and phone number)
4. The customer selects product from list.
5. The customer specifies the quantity.
6. The customer presses the „ORDER” button
7. The application verifies correctness of input.
8. Application adds new order to the department.

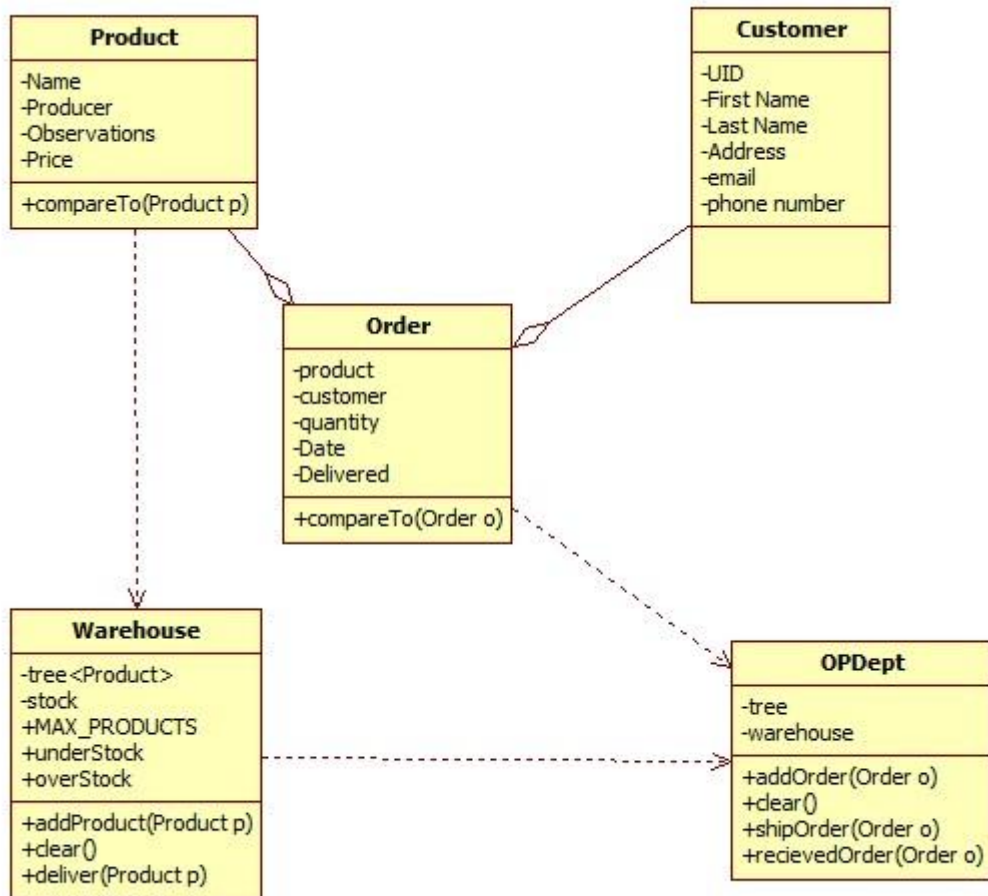


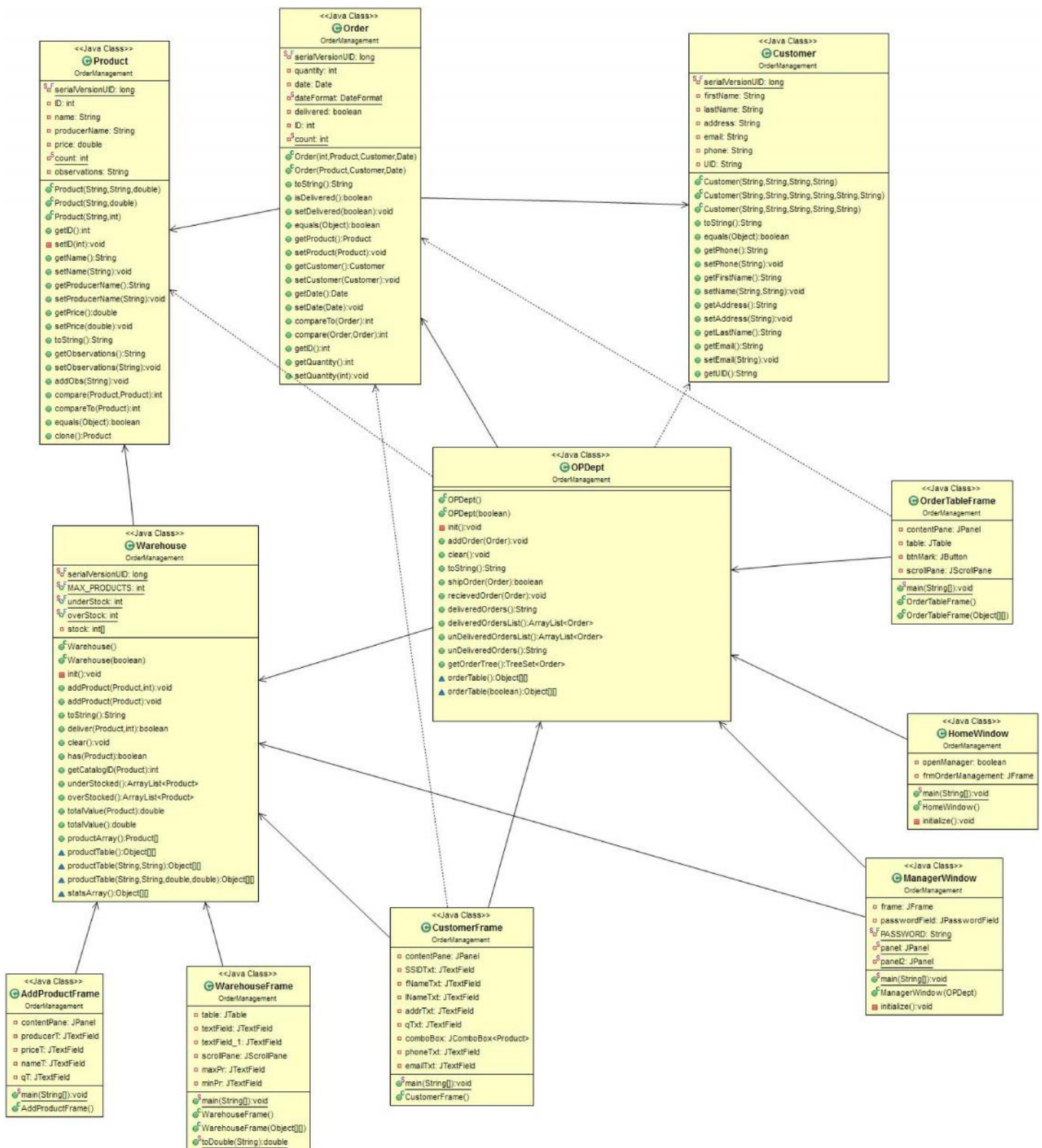
3. Desing

3.1 UML Class Diagram

The first UML class digram to be preseted has the purpose to show the correlation between the main specidied classes: Product, Customer, Order, Warehouse and OPDept. It also outlines the main features these classes should have.

The second UML class diagram is the extended UML diagram, that shows all the implemented classes and all their features.





3.2 Data Structures

The most important data structure used in this application is the Binary Search Tree used to store orders in OPDept, and products in the Warehouse. The Java API already contains an implemented Binary Search Tree, under the name of TreeSet.

```
public class TreeSet<E>  
extends AbstractSet<E>  
implements NavigableSet<E>, Cloneable, Serializable
```

A NavigableSet implementation based on a TreeMap. A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed $\log(n)$ time cost for the containsKey, get, put and remove operations.

Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.

The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed $\log(n)$ time cost for the basic operations (add, remove and contains).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its compareTo (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

Note that this implementation is not synchronized. If multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSortedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

```
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));
```

The iterators returned by this class's iterator method are fail-fast: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

This class is a member of the Java Collections Framework.

Since:

1.2

Another data structure used is Date, for saving the time an Order took place:

```
public class Date
```

```
extends Object
```

```
implements Serializable, Cloneable, Comparable<Date>
```

The class Date represents a specific instant in time, with millisecond precision.

Ordinary arrays were also used in this program to form the tables where the orders and products are displayed.

3.3 Algorithms

The algorithms used in this application are those related to Binary Search Trees. Some of them are implemented by the TreeSet class, some had to be adapted. When talking about trees we must have:

- Insertion: implemented in TreeSet as add(E e) method. It adds the elements in its specific location, as a leaf, and then rebalances the tree.
- Search: to find an element in a tree we may use the contains(Object o) method implemented by the TreeSet, or the has(Object o) method implemented by us. The first option is an optimal search, based on the specified comparator and equals method. But, sometimes we may search only for partially equal object (ex. Products with the same name, even if from different producers). Thus, in this case we can use the has(Object o) to see if we have any matches and then use a more specific method to extract them. The has(Object o) method iterates the tree in order and compares the each elements. This takes us $O(n)$ time, comparing to $O(\lg n)$ in the first case.
- Delete: we do not need such an operation. To reset the application, we use the clear() method that creates an empty tree and saves it in the files.

3.4 Class Design

We will present the CRC cards corresponding to each class

Product	
Functions:	Collaborator classes:
<ul style="list-style-type: none">- Has name, producer and price- Must be associated and ID to have easier access- May have observations- ID must not restart on application relaunch	

Customer	
Functions:	Collaborator classes:
<ul style="list-style-type: none">-Has first name and last name-Has SSID, that uniquely identifies each customer-Has address-Has phone number-Has email	

Order	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Models a real life order - Associates a Customer with an ordered Product - Is made at a specific date - A quantity is also included for the product - Orders can be compared by their date - Can be already delivered or not 	Customer Product

Warehouse	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Models real life warehouse corresponding to a store - Contains products - They are placed in a Binary Search Tree (TreeSet) - Each product has associated a stock - Defines thresholds for under-stock and over-stock - Has a maximum storing capacity - Can add product to the tree - Can return a list of products - Can ship products - If we try introduce a product with the same name, producer and price as an already existing one in the tree, the stock will be updated, but no duplicate product will be added 	Product TreeSet

OPDept	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Models a real life order managing department - Stores all the placed orders in a Binary Search Tree (TreeSet) - Can return a list of orders - Can add new Order - Can ship an order - Can mark product as delivered - Knows delivered and undelivered orders 	Order TreeSet Warehouse

HomeWindow	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Provides a general user interface - Can redirect to : warehouse, order product or manager log in 	WarehouseFrame CustomerFrame ManagerWindow JFrame JButton

ManagerWindow	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Allows access only to manager by requesting a password - Manager can view all orders, view delivered orders, view undelivered orders, clear all orders - Manager can also add new Product to warehouse (or update stock) 	OPDept Warehouse JPanel JFrame JButton OrderTableFrame AddProductFrame

OrderTableFrame	
Functions:	Collaborator classes:
-Displays all orders in from OPDept in a table -Can display delivered or undelivered orders	OPDept

AddProductFrame	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Recieves as input the name, producer and price of a product - Recieves the quantity to be added - Adds transmitted product to warehouse 	Warehouse JTextField JButton

CustomerFrame	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Allows Customer to place order - Customer introduces personal data(SSID, name, address . . .) - Customer selects product from a list, retrieved from the Warehouse - Customer specifies quantity to order - Customer finalizes order by pressing ORDER button - Forms Order from Customer and Product - Adds Order to OPDept 	Customer Product Order OPDept Warehouse

WarehouseFrame	
Functions:	Collaborator classes:
<ul style="list-style-type: none"> - Facilitates display of products stored in the warehouse - Uses a Table - Shows different attributes, depending on the caller(Customer or Manager) 	Warehouse

3.5 Interface

I will now present the proposed user interfaces

The general User interface:



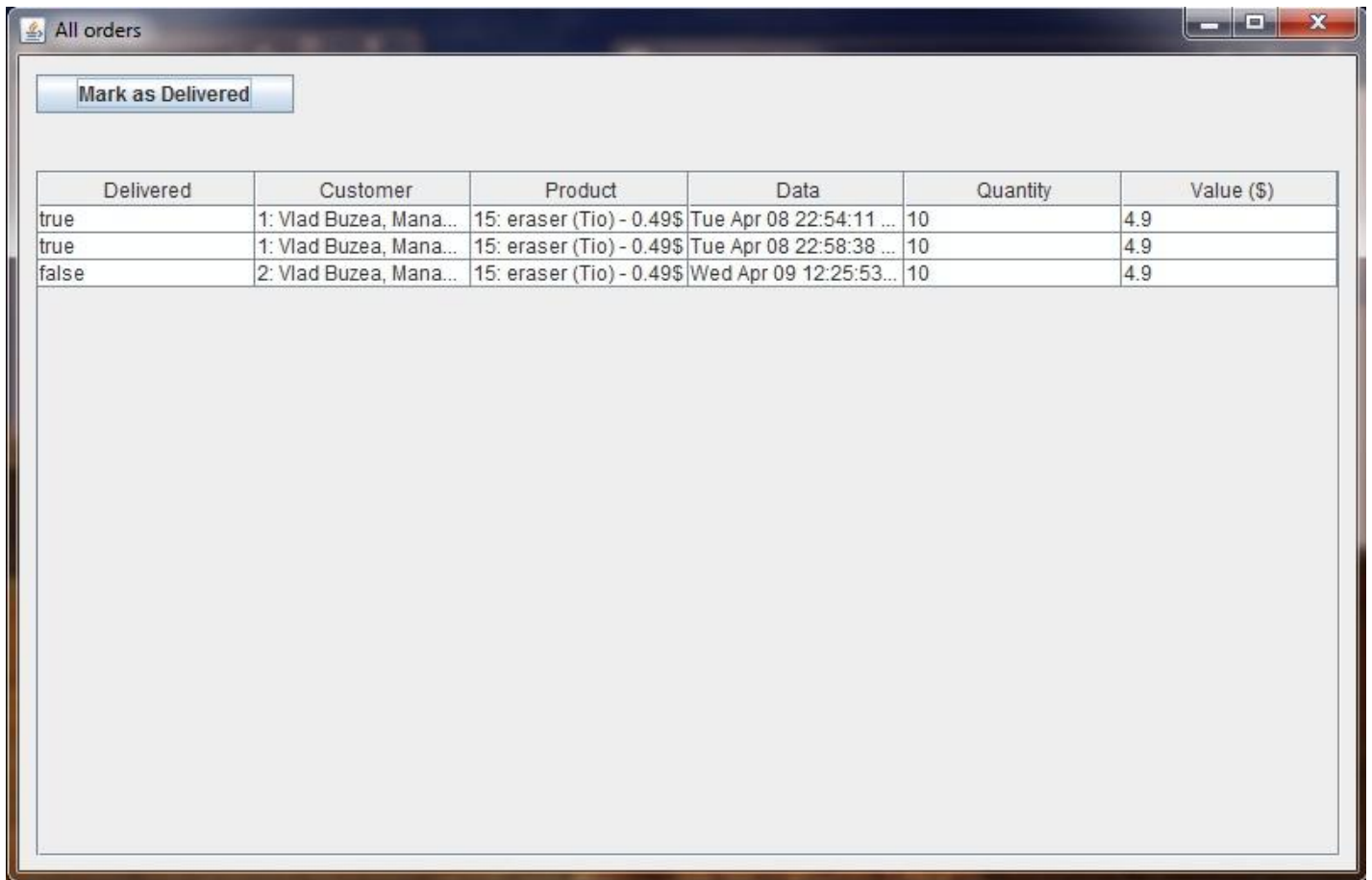
The ManagerWindow is composed of two panels. The first one is a login panel visible until the correct password is provided and set invisible afterwards



The second panel is a commands panel, that is set visible only after the correct password has been typed in the first panel.



If the Manager selects to View Orders, the OrderTableFrame will pop out:



The Manager can also view Warehouse Stats, case in which a new WarehouseFrame will pop up. This frame has the next aspect:



The values are placed in a JTable that is in it's turn placed in a JScrollPane as a container.

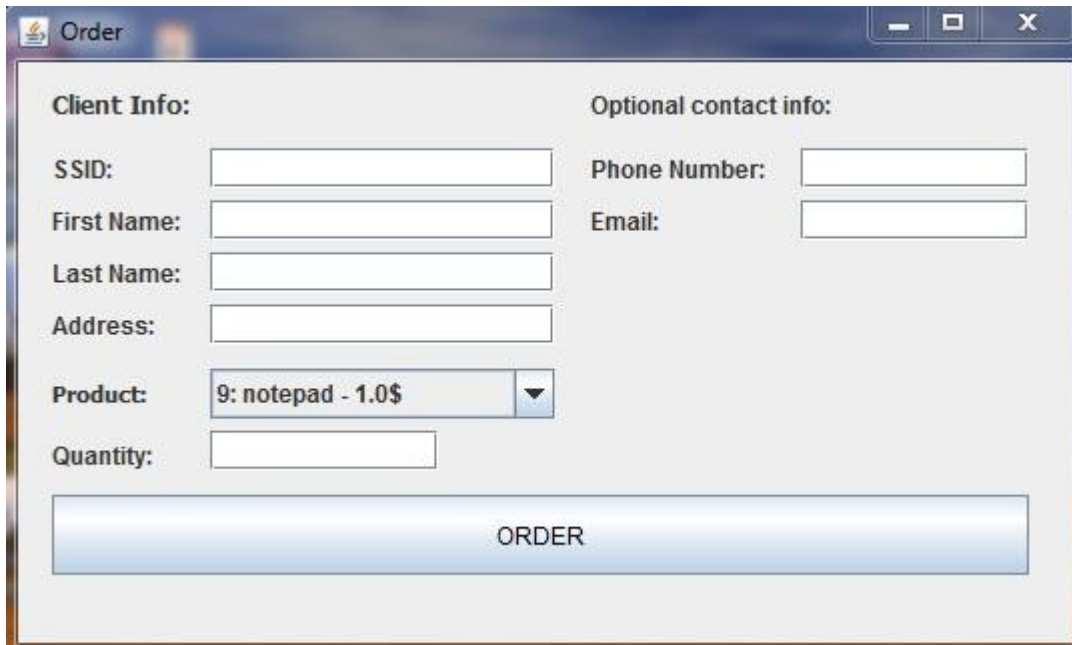
If the manager chooses to add a new product to the warehouse, a new AddProductFrame will be created:



The 'Add Product' dialog box contains the following fields and controls:

- Product:**
 - Name:** Text input field
 - Producer:** Text input field
 - Price:** Text input field
 - Quantity:** Text input field
- Add to Warehouse:** A large blue button at the bottom.

If we press the Order Product button on the HomeWindow, a new instance of CustomerFrame will be created:



The 'Order' dialog box contains the following fields and controls:

- Client Info:**
 - SSID:** Text input field
 - First Name:** Text input field
 - Last Name:** Text input field
 - Address:** Text input field
- Optional contact info:**
 - Phone Number:** Text input field
 - Email:** Text input field
- Product:** A JComboBox showing '9: notepad - 1.0\$' with a dropdown arrow.
- Quantity:** Text input field
- ORDER:** A large blue button at the bottom.

One can see that products are selected from a JComboBox such that we reduce the possibility of error. We can be sure that the ordered product exists in our warehouse (even if the stock is not enough). So a customer may order a product even if it is not available in stock. It is then left to the manager to solve the issue.

If we press the Warehouse button on the HomeWindow, a new instance of WarehouseFrame will be created, having presented only product information, not warehouse stats:

ProductID	Name	Producer	Price (\$)	Stock
9	notepad		1.0	15
11	laptop		999.99	15
15	eraser	Tio	0.49	110
16	eraser	Tio	4.9	20
18	Ruler	Pelikan	0.89	1200

4. Implementation and testing

Implementation was done using Eclipse IDE, in Java programming language. The resulted classes are a mapping of what was described in the UML class diagram and class design. Further on, I will present the most relevant functions in the applications. Getter and setter methods will be skipped due to their lack of complexity.

The equals method has been overridden for the classes: Customer, Product and Order. Two customers are considered to be equal if they have the same SSID (UID). Two products are considered to be equal if they have the same name, producer and price. The product ID is a warehouse dependent field, that can also be called catalogID, so it does not influence the real characteristics of product. Besides that, different ID's can and will be created for the same Product at instantiation. Two orders are considered to be equal if they contain the same product, customer and date. Practically speaking, there is an extremely small chance that two orders are placed in the exact same time, but for that purpose, we check the Customer and product as well for equality.

When comparing orders we take into account the date they were made. When comparing products we use the ID (catalogID). This lays a foundation for ordering the trees that will contain these items.

Regarding the Warehouse class, I have implemented 3 class constants:

- `MAX_PRODUCTS` - represents the maximum storing capacity
- `underStock` – represents the minimum number of items a product must have on stock to assure optimal delivery
- `overstock` – represents the maximum number of items a product should have on stock

Warehouse also has as fields :

- `int[] stock`: represents the stock associated to each item. To find the stock corresponding to a products we access the cell that has the number equal to the products ID
- `TreeSet<Product> tree` : contains the products in the warehouse, as a Binary Search Tree. Products are ordered using their ID

As methods, I find important the following:

- `public void addProduct(Product p, int stockVal)` : adds a product to the warehouse, in its corresponding place and associates the transmitted stock value in the `stock[]` array. If the product already exists, the stock is updated.
- `public boolean deliver(Product p, int cant)` : removes requested quantity from stock. Returns true if possible, false if not enough stock available.
- `public void clear()` : resets the warehouse. It will have no products stored.

- **public boolean** has(Product p) : returns true if the any of the contained products equals the parameter product.
- **public int** getCatalogID(Product p) : returns the catalog ID of a product by searching for the product in the tree and returning the found ID. In this way, we may operate on the stock.
- **public double** totalValue() : computes the total value stored in the warehouse. This is done by multiplying each product stock with the product price and adding all the results.
- **Object[][]** productTable(String name,String producer) returns a matrix with product name, producer price and stock. All products must start with the transmitted name and producer name. If none are specified, then all the products are returned. This method is used for viewing in table form the content of the warehouse.
- **Object[][]** statsArray() returns a more detailed matrix of products, for the manager GUI. This matrix has as columns the name, the producer, the stock, the total value of those Products, and a stock Balance

The OPDept class has as fields :

- `TreeSet<order>` tree : contains all orders
- Warehouse warehouse : references the store warehouse

Regarding important methods, we can outline the following:

- **public void** addOrder(Order o) : adds an order to the tree
- **public void** clear() : resets the tree containing the orders; deletes all orders
- **public boolean** shipOrder(Order o) : sends products from warehouse to customer
- **public void** recievedOrder(Order o) : marks the order as being received by the customer
- **Object[][]** orderTable(**boolean** delivered) : returns a matrix such that the orders may be displayed in a JTable. The first column returned is the delivery status (delivered or undelivered) , the second column is the Customer, the third is the Product, the fourth is the Date, the fifth is the quantity ordered and the last column is the cost of the order.

Testing has been done through the drivers corresponding to each class: CustomerDriver, OPDeptDriver, OrderDriver, ProductDriver and WarehouseDriver. In these abstract classes we have instantiated the tested the corresponding classes by calling all their methods. The results have not been saved for this documentation, but tested input data has responded perfectly.

5. Results

A quick overview of the application shows that we have reached the following results:

- An easy access application
- View over the available products and filtering the results by name, producer and price
- Product ordering
- Password protected access for manager
- Managerial perspective: warehouse stats, view of orders
- Possibility to add a new product
- Possibility to reset
- Tracking of delivered and undelivered orders

6. Conclusions, what has been learned, further developments

Throughout this homework I have learned:

- How to read and write objects from/in a file
- How to use the class TreeSet<E>
- How to use JTables and JComboBoxes
- How to use keyEvents

Although the project is in working condition, it is just the base of the framework regarding Order Management. As further developments, one may easily point out the following:

- Possibility to cancel an order
- Users with different priorities
- Separation of order time and shipment time
- Supplementary information regarding Orders and Products
- A better GUI

7. Bibliography

<http://stackoverflow.com/questions/5865453/java-writing-to-a-text-file>

<http://www.javapractices.com/topic/TopicAction.do?id=42>

Java API (j2se7)