Technical University of Cluj Napoca

Faculty of Automation and Computer Science

Order Management

Lorand Berekmeri

Group: 30425

Discipline: Programming Techniques

Date: 1 April 2016

Task:

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.

b. Generate use case diagrams, an extended UML class diagram, two sequence diagrams and an activity diagram.

c. Implement and test the application classes. Use javadoc for documenting the classes.

d. 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.

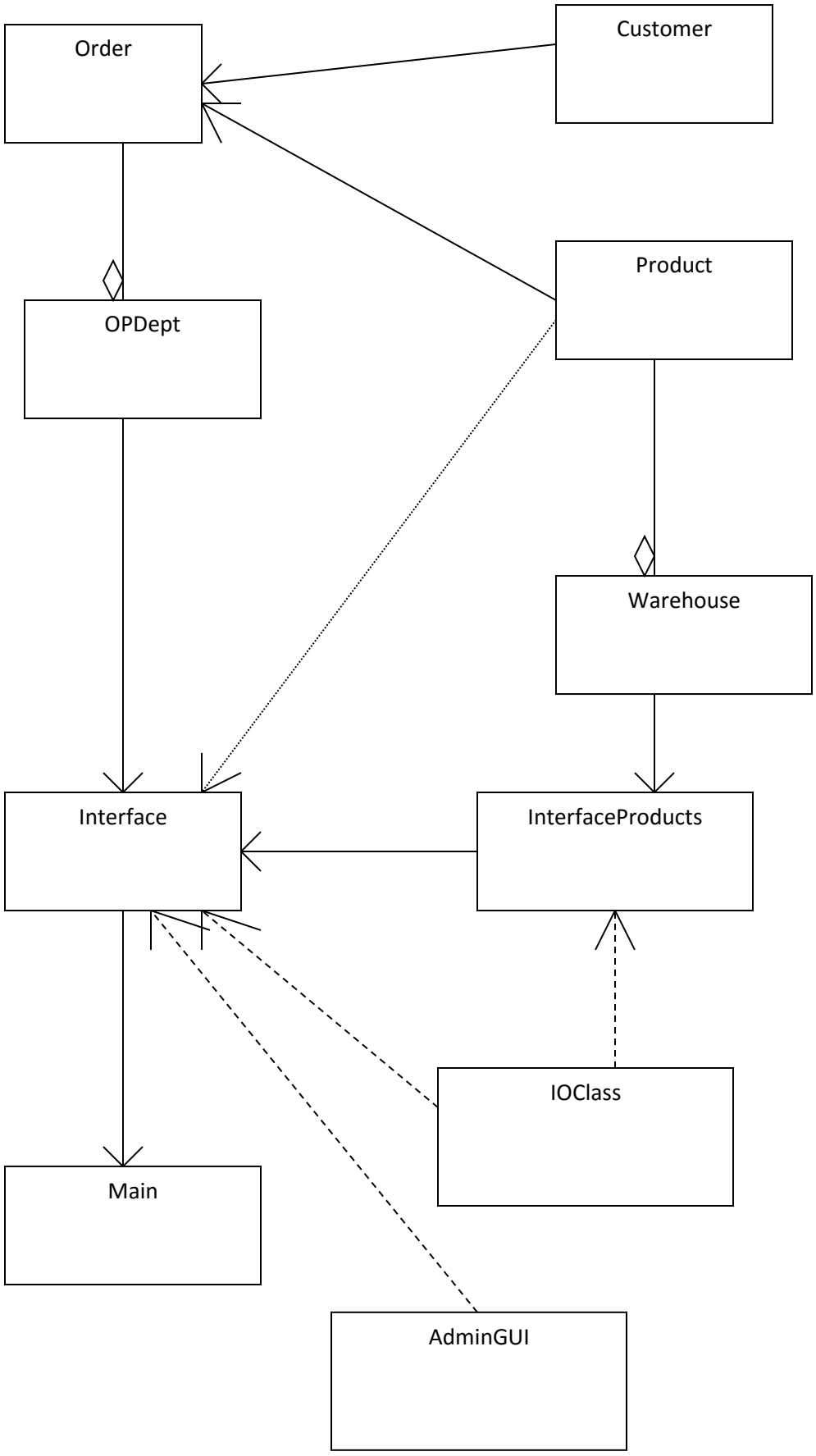
Data structures:

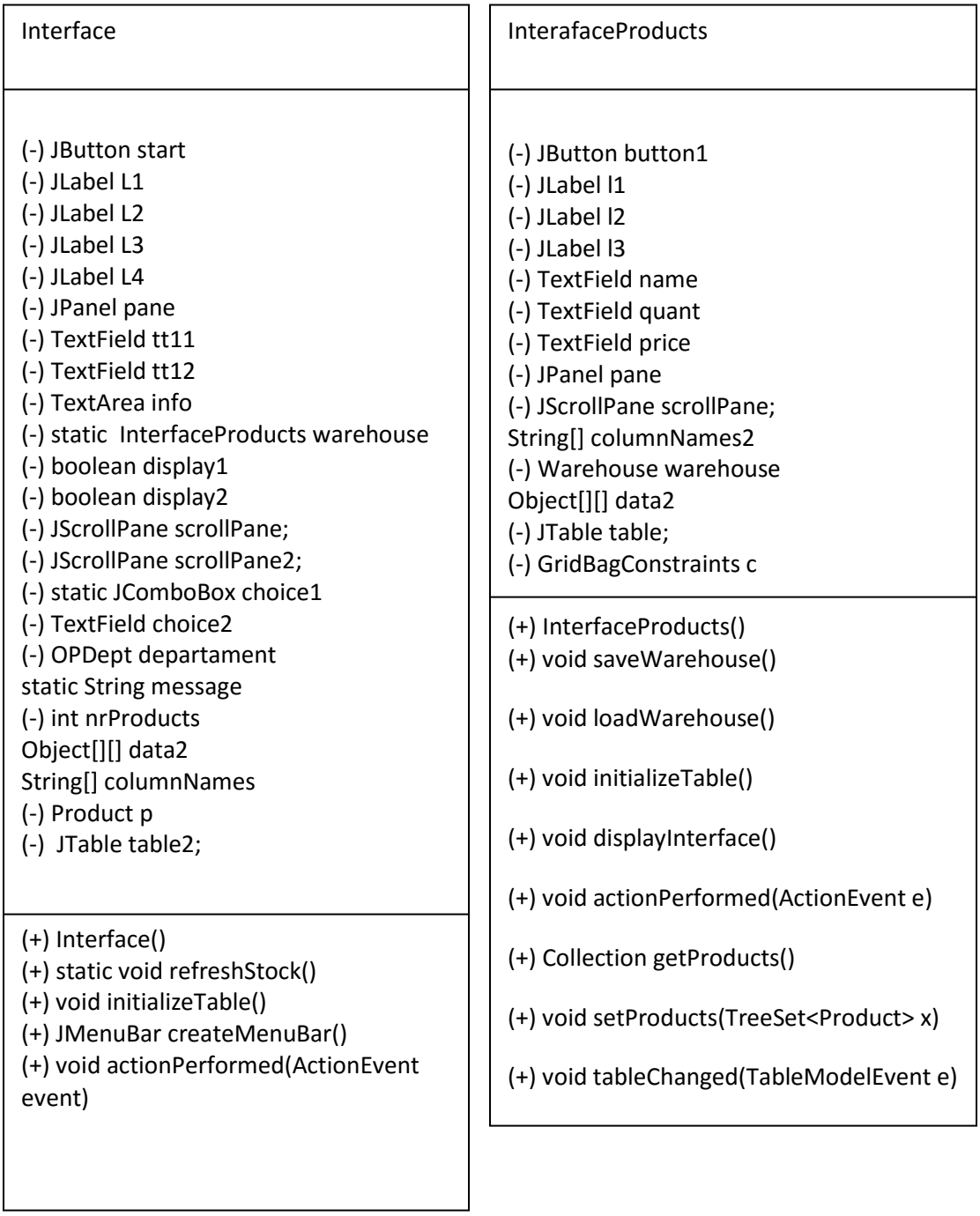
The most important data structure used in this application is the Binary Search Tree used to store orders in OPDept, and products in the Warehouse. In computer science, binary search trees (BST) allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name). Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree.

Binary Search Tree is node based binary tree data structure with the following properties:

** The Left subtree contains the nodes with keys less than the node's key.*
** The Right subtree contains the nodes with keys greater than the node's key.*
** Both the right and left subtree should also be binary search tree.*
** There should not be any duplicate nodes.*

UML Diagram


Order
Customer
OPDept
Product
Warehouse
Interface
InterfaceProducts
IOClass
Main
AdminGUI


Interface
(-) JButton start
(-) JLabel L1
(-) JLabel L2
(-) JLabel L3
(-) JLabel L4
(-) JPanel pane
(-) TextField tt11
(-) TextField tt12
(-) TextArea info
(-) static InterfaceProducts warehouse
(-) boolean display1
(-) boolean display2
(-) JScrollPane scrollPane;
(-) JScrollPane scrollPane2;
(-) static JComboBox choice1
(-) TextField choice2
(-) OPDept departament
static String message
(-) int nrProducts
Object[][] data2
String[] columnNames
(-) Product p
(-) JTable table2;
(+) Interface()
(+) static void refreshStock()
(+) void initializeTable()
(+) JMenuBar createMenuBar()
(+) void actionPerformed(ActionEvent event)
InterafaceProducts
(-) JButton button1
(-) JLabel l1
(-) JLabel l2
(-) JLabel l3
(-) TextField name
(-) TextField quant
(-) TextField price
(-) JPanel pane
(-) JScrollPane scrollPane;
String[] columnNames2
(-) Warehouse warehouse
Object[][] data2
(-) JTable table;
(-) GridBagConstraints c
(+) InterfaceProducts()
(+) void saveWarehouse()
(+) void loadWarehouse()
(+) void initializeTable()
(+) void displayInterface()
(+) void actionPerformed(ActionEvent e)
(+) Collection getProducts()
(+) void setProducts(TreeSet<Product> x)
(+) void tableChanged(TableModelEvent e)

Warehouse
(-) TreeSet <Product> products
(+) Warehouse() (+) Warehouse(TreeSet<Product> x) (+) void setProducts(TreeSet<Product> x) (+) void addProduct(Product p) (+) Collection getProducts()

OPDept
(-) static int clientNumber (-) static int orderNumber (-) TreeSet<Order> orders
(+) OPDept() (+) Collection getOrders() (+) Collection processOrder(TreeSet<Product> x, Product y, int q,String cn, String ca)

Customer
(-) int id (-) String name (-) String address
(+) Customer() (+) Customer(int i, String s1, String s2) (+) String getName() (+) String getAddress() (+) int getId()

Order
(-) int nrOrder (-) Customer client (-) Product prod (-) int totalPrice
(+) Order(int i,int j, Customer c, Product p)(+) Customer getClient() (+) Product getProduct() (+) int getNrOrder() (+) int compareTo(Object o) (+) int getCost() (+) String toString() (+) int compare(Order a1, Order a2)

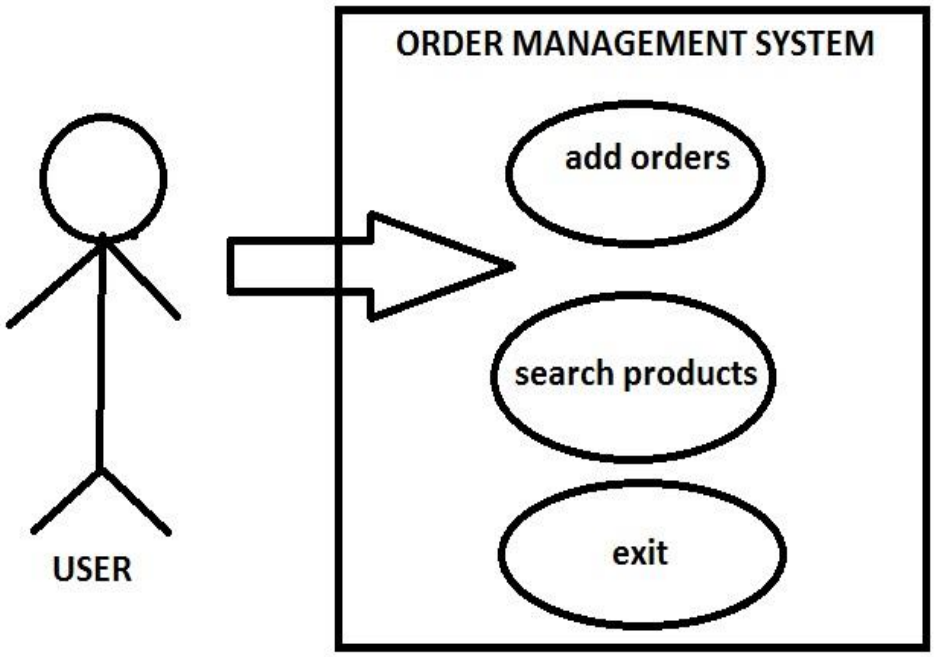
Product
(-) int id (-) String name (-) int price (-) int amount
(+) Product() (+) Product(String s, int b,int c) (+) String getName() (+) int getPrice() (+) int getId() (+) int getAmount() (+) void setAmount(int x) (+) int compareTo(Object arg0) (+) String toString()

IOClass
(+) static void saveInfo(Object oo, String dest) (+) static Object loadInfo(String dest)

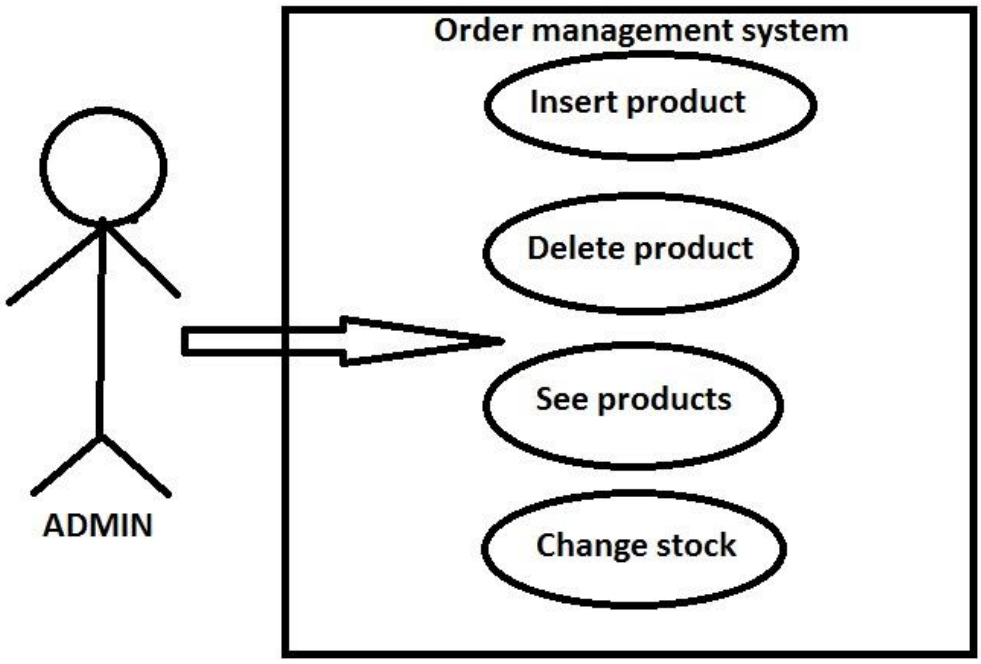
AdminGUI
(-) int i
(+) checkUser()

Main
(+) static void main(String[] args)

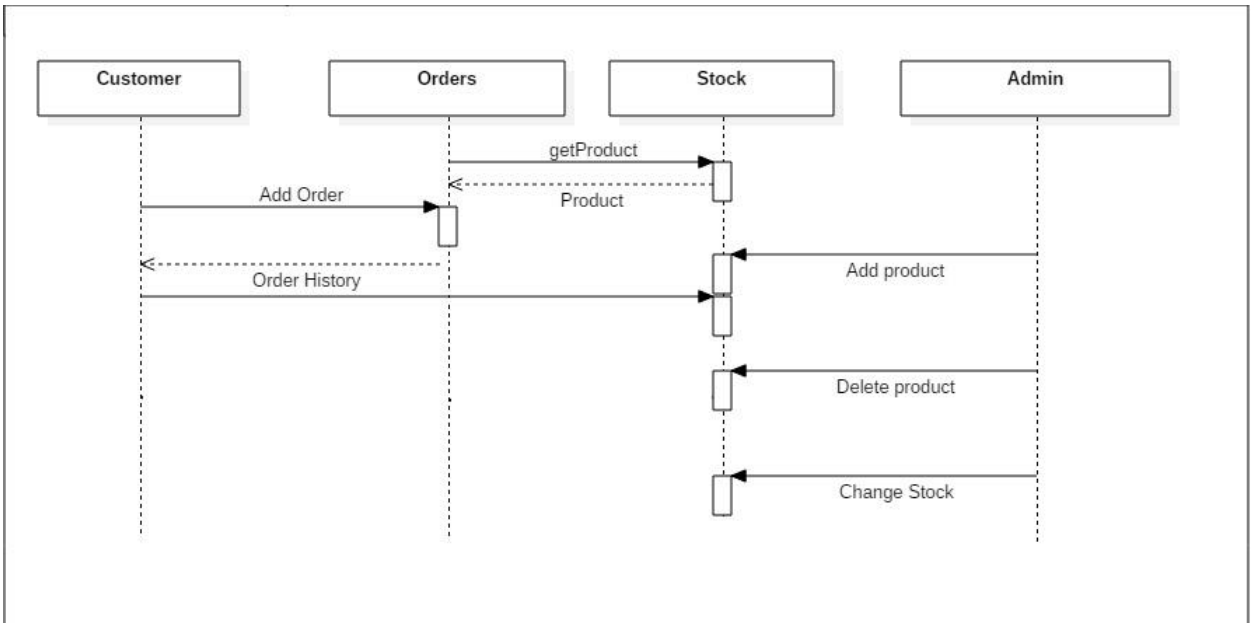
Use case diagrams - User

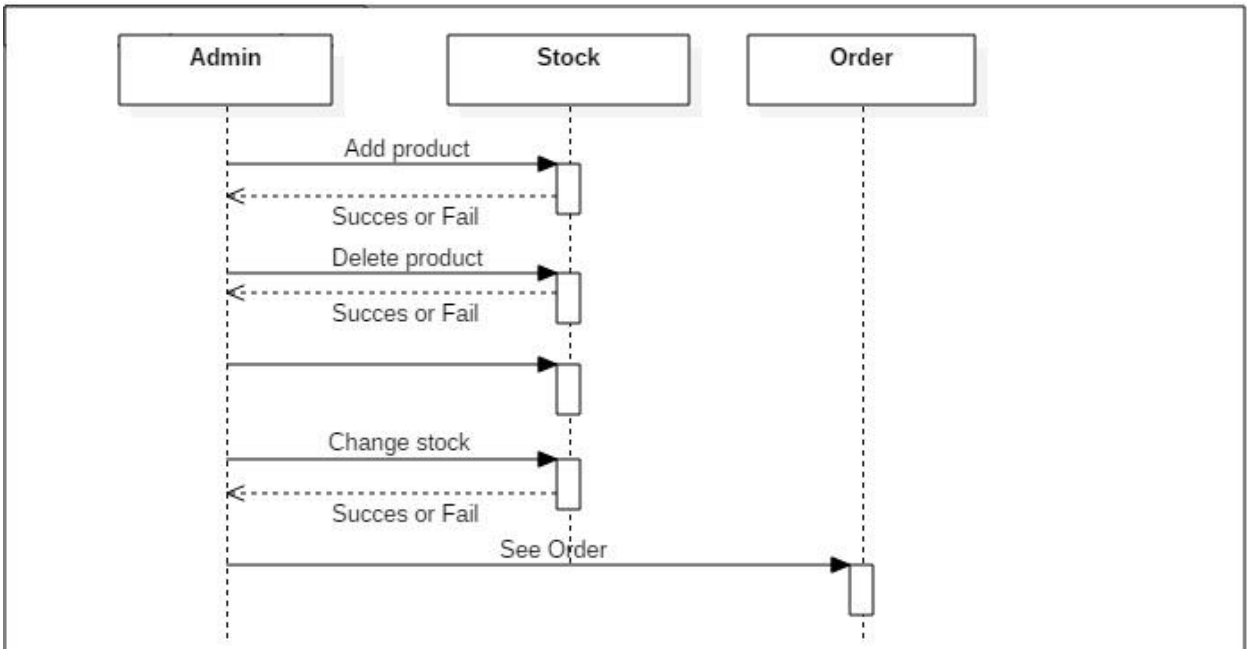


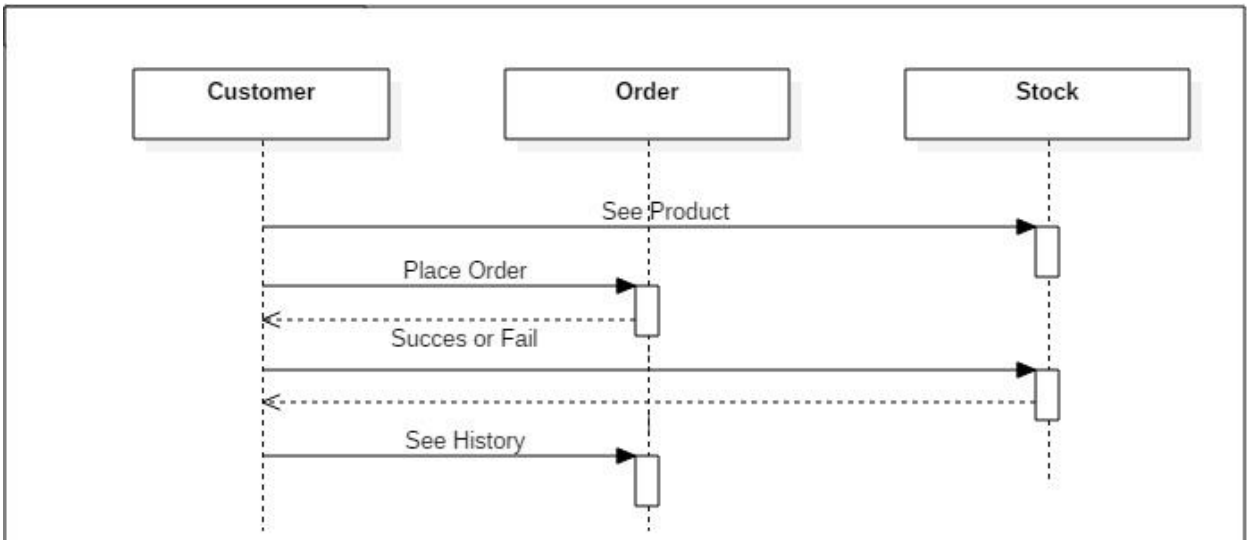
Administrator



Sequence Diagrams







Graphical User Interface

The software developed has a user friendly interface composed of :
A main window with the following components:

1. *a menu with the following components:*
   - *Save products*
   - *Save orders*

Introduce products

Exit

2. label Client name

3. text field to introduce name

4. label Client address

5. text field to introduce address

6. label Product

7. combo box to choose product

8.label Amount

9. text field to read amount

10. button OK to introduce order

11. text area to display the processing messages

12. table to display orders

A second window that opens when the introduce products is selected

1.label Product name

2.text field to read name

3. label Quantity

4. text field to read amount

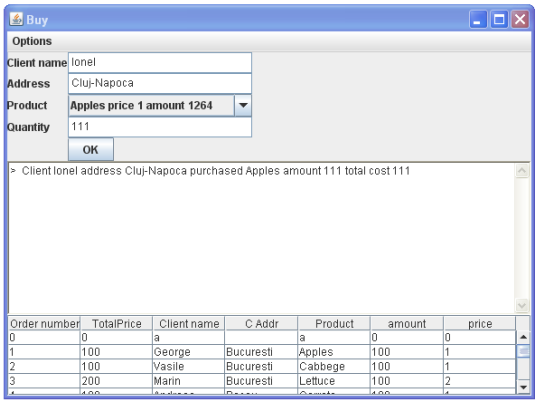
5. label price

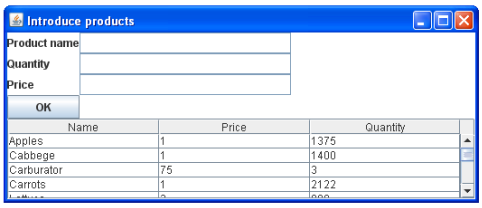
6. text field to read price

7. button to introduce data

8. table to display products

The components are displayed on a panel using the GridBagLayout. This panel is than added on the frame.





Classes

The program contains several classes as already described above. To begin with, there is the class ***Customer****. It represents the entity of a client containing the basic information needed by a client. It contains a name, an id – integer field and an address, a string. Because the fields are private (encapsulation) setter and getter methods were defined. It implements the Serializable interface so it can be written in a file.*

Another class is the ***Product****. It contains the basic information of a product: id, a string representing the name, an integer representing the price and another integer representing the amount. It implements the Serializable interface and the Comparable interface overriding the compareTo method.*

The class ***Order*** *represents the entity of an order. It contains the fields nrOrder and totalPrice. Besides these, it contains an object of type Product and another one of type Customer, representing the client and the product that was bought. Thie class also implements the Serializable and Comparable interfaces.*

The class ***Warehouse*** *represents the warehouse where the stock of products is being kept. For accessing in a short time the products they are kept in a TreeSet. This class also contains getter and setter methods as well as methods to add or remove products. Furthermore, this class is also Serializable.*

The class ***OPDept*** *represents the class that contains the algorithms needed to process the orders. It also contains a TreeSet to keep the orders for a quick access. It contains the method processOrder that processes the order. This method also updates the products in the warehouse and takes decisions regarding the fact when there are not enough products.*

The class ***InterfaceProducts*** *contains the class that creates an interface for the administrator of the warehouse. It is composed of several text fields to introduce a new product and a table to display the available products and the information about them.*

The class Interface contains the user interface. It is composed of a few text fields to fill the information about the customer, a combo box to choose the product and a table to display the orders. The class contains calls to other methods from the classes mentioned above.

The class ***IOClass*** *contains static methods that allow to write and read a serializable object from and to a file.*

The class ***AdminGUI*** *contains only a method which is used for setting a password for the user, in order to introduce products.*

The class ***Main*** *contains the main method to run the program.*

BST CLASS:

Inorder print: the main idea of this method is to print the values in inorder type

Recursive search: we begin first by comparing the value with the root node and if the value is equal to root node value then the search is successful. Otherwise we check if it is greater or smaller than root node. If smaller we go to the left subtree and repeat the above process and if its greater we go to the right subtree and repeat the above process.

Pseudocode:

1. Start at the root node as current node
2. If the search key's value matches the current node's key then found a match
3. If search key's value is greater than current node's
   1. If the current node has a right child, search right
   2. Else, no matching node in the tree
4. If search key is less than the current node's
   1. If the current node has a left child, search left
   2. Else, no matching node in the tree

Insert a node: Insert Node operation also behaves in the same manner as Searching operation. Firstly, it checks whether the key is the same as that of root, if not then we either choose the right subtree or the left subtree depending upon the value passed is greater or smaller than the root node value respectively.
Pseudo Code:
1.Always insert new node as leaf node
2. Start at root node as current node
3. If new node's key < current's key
   1. If current node has a left child, search left
   2. Else add new node as current's left child
4. If new node's key > current's key
   1. If current node has a right child, search right
   2. Else add new node as current's right child

Delete method: Removes the specified key and its associated value from the symbol table It begins by searching on the left hand side and if the one from left differs from null, then goes to right. When finds a node which equals null, it is swapped with the root.

BSTNode Class:

In this class I've been implemented only two constructors, which are taking in account the construction of the binary search tree. I used a value, which is Comparable type, parent key and child (left and right).

```
BSTNode( Comparable newVal ) {
        val = newVal;
        parent = null;
        left = right = null;
    }
```

```
BSTNode( Comparable newVal, BSTNode p ) {
        val = newVal;
        parent = p;
    left = right = null;
}
```

Relations

There are several relation that can be found in the architecture if this project.

The class Warehouse contains a TreeSet of Products, so the relation between them is of aggregation, because this class contains several instances of Products. The class OPDept contains a TreeSet of Orders so the relation between them is of aggregation.

The class Order contains both a Product and a Client, so the relation between them is of association.

The class InterfaceProducts contains an instance of Warehouse, so the relation between them is of association. The class Interface contains an instance of OPDept so the relation between them is of association.

The classes Interface and InterfaceProducts use static methods from the class IOClass, but because there is no instance of this class, the relation between them is of dependency.

The implementation is shown by the fact that classes like Product and Order implement the Comparable interface, while other classes implement the Serializable interface.

Furthermore, the inheritance is shown by the fact that several classes inherit from JFrame.

Below there will be presented each method and algorithm used in it:

1. Class OPDept

java.lang.Object

└ **OPDept**

All Implemented Interfaces:

java.io.Serializable

---

public class **OPDept**

extends java.lang.Object

implements java.io.Serializable

Class that simulates an ORDER PROCESSING DEPARTAMENT. It contains methods to process orders and refresh the stock of objects

Constructor Detail

OPDept

public **OPDept**()

constructor

Method Detail

getOrders

public java.util.Collection **getOrders**()

get orders

Returns:

collection of orders

---

processOrder

public java.util.Collection **processOrder**(java.util.TreeSet<Product> x, Product y, int q, java.lang.String cn , java.lang.String ca)

method that processes an order and refreshes the stock of products

Parameters:

x - the stock of products

y - the product bought

q - amount of product bought

cn - customer name

ca - customer address

Returns:

the refreshed tree set of products after transaction

2. **Class Interface**

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

public class **Interface**

extends javax.swing.JFrame

implements java.awt.event.ActionListener

The class Interface represents the class that creates the user interface. It displays several labels and text fields on a JPanel using a GridBaglayout and has a Jmenu. It extends JFrame and implements ActionListener

Constructor Detail

Interface

public **Interface**()

constructor- display user interface

Method Detail

refreshStock

public static void **refreshStock**()

refreshes stock of products displayed in combo box

---

init

public static void **init**()

initializes the combo box with available products

---

initializeTable

public void **initializeTable**()

initializes table with orders

---

createMenuBar

public javax.swing.JMenuBar **createMenuBar**()

create a menu bar

Returns:

menu bar created

---

actionPerformed

public void **actionPerformed**(java.awt.event.ActionEvent event)

Specified by:

actionPerformed in interface java.awt.event.ActionListener

3. **Class Warehouse**

java.lang.Object

└ **Warehouse**

All Implemented Interfaces:

java.io.Serializable

---

public class **Warehouse**

extends java.lang.Object

implements java.io.Serializable

Class that models the entity of a Warehouse It contains a tree set of products it implements the serializable interface so it can be saved in a file

Constructor Detail

Warehouse

public **Warehouse**()

constructor

Warehouse

public **Warehouse**(java.util.TreeSet<Product> x)

constructor

Parameters:

x - tree set of products

Method Detail

setProducts

public void **setProducts**(java.util.TreeSet<Product> x)

set a new tree set of products

Parameters:

x - new tree set of products

---

addProduct

public void **addProduct**(Product p)

add a new product to the warehouse

Parameters:

p - new product to be added to the warehouse

---

getProducts

public java.util.Collection **getProducts**()

get products

Returns:

collection of products

4. **Class Customer**

java.lang.Object

└ **Customer**

All Implemented Interfaces:

java.io.Serializable

---

public class **Customer**

extends java.lang.Object

implements java.io.Serializable

Class that models the entoty of a customer containing the basic information about a customer that purchases some products

Constructor Detail

Customer

public **Customer**()

constructor

Customer

public **Customer**(int i, java.lang.String s1, java.lang.String s2)

constructor

Parameters:

i - id

s1 - name

s2 - address

Method Detail

getName

public java.lang.String **getName**()

get name

Returns:

name

---

getAddress

public java.lang.String **getAddress**()

get address

Returns:

address

---

getId

public int **getId**()

get id

Returns:

id

5. **Class InterfaceProducts**

All Implemented Interfaces:

java.awt.event.ActionListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.util.EventListener, javax.accessibility.Accessible, javax.swing.event.TableModelListener, javax.swing.RootPaneContainer, javax.swing.WindowConstants

---

public class **InterfaceProducts**

extends javax.swing.JFrame

implements java.awt.event.ActionListener, javax.swing.event.TableModelListener, java.io.Serializable

Constructor Detail

InterfaceProducts

public **InterfaceProducts**()

constructor display user interface

Method Detail

saveWarehouse

public void **saveWarehouse**()

save products in binary file

---

loadWarehouse

public void **loadWarehouse**()

load products from binary file

---

initializeTable

public void **initializeTable**()

initialize table with elements

---

displayInterface

public void **displayInterface**()

display user interface

---

actionPerformed

public void **actionPerformed**(java.awt.event.ActionEvent e)

Specified by:

actionPerformed in interface java.awt.event.ActionListener

---

getProducts

public java.util.Collection **getProducts**()

get products

Returns:

tree set of products

---

setProducts

public void **setProducts**(java.util.TreeSet<Product> x)

set products

Parameters:

x - new tree set of products

---

tableChanged

public void **tableChanged**(javax.swing.event.TableModelEvent e)

check if table elements changed

Specified by:

tableChanged in interface javax.swing.event.TableModelListener

6. **Class IOClass**

java.lang.Object

└ **IOClass**

---

public class **IOClass**

extends java.lang.Object

Constructor Detail

IOClass

public **IOClass**()

Method Detail

saveInfo

public static void **saveInfo**(java.lang.Object oo, java.lang.String dest)

save an object into a file

Parameters:

oo - the object to be saved

dest - name of file

---

loadInfo

public static java.lang.Object **loadInfo**(java.lang.String dest)

read an object from a file

Parameters:

dest - the name of the file where the object is

Returns:

the object read

7. **Class Order**

java.lang.Object

└ **Order**

All Implemented Interfaces:

java.io.Serializable, java.lang.Comparable, java.util.Comparator<Order>

---

public class **Order**

extends java.lang.Object

implements java.lang.Comparable, java.util.Comparator<Order>, java.io.Serializable

Class that models the entity of am order Contains objects of type product and client It implements the Comparable interface and Serializable interface and the Comparator interface

Constructor Detail

Order

public **Order**(int i, int j, Customer c, Product p)

constructor

Parameters:

i - number of order

j - total price

c - customer

p - product

Method Detail

getClient

public Customer **getClient**()

get client

Returns:

client

---

getProduct

public Product **getProduct**()

get product

Returns:

product

---

getNrOrder

public int **getNrOrder**()

get number of order

Returns:

number of order

---

compareTo

public int **compareTo**(java.lang.Object o)

compare to method

Specified by:

compareTo in interface java.lang.Comparable

---

getCost

public int **getCost**()

get total price

Returns:

total price

---

toString

public java.lang.String **toString**()

create a string representation of the object

Overrides:

toString in class java.lang.Object

---

compare

public int **compare**(Order a1, Order a2)

Specified by:

compare in interface java.util.Comparator<Order>

8. **Class Product**

java.lang.Object

└ **Product**

All Implemented Interfaces:

java.io.Serializable, java.lang.Comparable

---

public class **Product**

extends java.lang.Object

implements java.lang.Comparable, java.io.Serializable

This class represents the entity of a product. It contains the information about the products. It also implements the Serializable and Comparable interfaces

Constructor Detail

Product

public **Product**()

constructor

Product

public **Product**(java.lang.String s,

int b,

int c)

constructor

Parameters:

s - name

b - price

c - amount

Method Detail

getName

public java.lang.String **getName**()

get name

Returns:

name

---

getPrice

public int **getPrice**()

get price

Returns:

price

---

getId

public int **getId**()

get id

Returns:

id of product

---

getAmount

public int **getAmount**()

get amount

Returns:

amount

---

setAmount

public void **setAmount**(int x)

set amount

Parameters:

x - new amount

---

compareTo

public int **compareTo**(java.lang.Object arg0)

override compareTo method

Specified by:

compareTo in interface java.lang.Comparable

---

toString

public java.lang.String **toString**()

string representation of this object

Overrides:

toString in class java.lang.Object

Result

This program shows a way of processing some orders implying a list of products. It takes into account just data regarding the products, not data about the client like bank accounts or ways of paying for the merchandise.

Conclusion

By creating this application I have acquired more information about the order management idea and how most of the markets/stores are working, of course, in a general point of view. Even though at first sight the program performs, further testing should be taken. There also should be implemented a way of dealing with bank accounts and ways of paying. There also should be used a better graphical interface and some searching options for the products.

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*

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*

Bibliography:

http : // mathworld.wolfram . com / Polynomial

http://stackoverflow.com/

https://en.wikipedia.org/

https://www.google.ro/

http://www.dreamincode.net/