

## Universitatea Tehnică Cluj-Napoca

## Facultatea de Automatică şi Calculatoare

## Secţia: Calculatoare, engleză

## Programming Techniques

## ~ OrderManagement ~

Student: Ban Bogdan Marian

Group id: 30425, second year

Professor: Salomie Ioan

Table of Contents

1. Task Objectives. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 03

2. Problem analysis, modelling, scenarios, use cases. . . . . . . . . . . . . . . . . .03

3. Projection (UML diagrams, data structures, class projections, . . . . . . . .04

Interface, relations, packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .06

Sequence ). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .07

4. Implementation and testing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

5. Results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .12

6. Conclusions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .12

7. Bibliography. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .12

1. Task Objectives

Requirement: 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.
  + respectarea claselor minimale din enuntul problemei si folosirea BST - 1p
  + admin view si regular user view - 1p
  + operatii admin (ex: adaugare/stergere de produse - 1p, modificarea stocului - 1p)
  + operatii user normal (filtrare produse - 1p, plasare comenzi - 1p, pastrare si vizualizare istoric comenzi - 1p)
  + documentatie - 3p

2. Problem analysis, modelling, scenarios, use cases

Problem analysis

The problem required us to create an application that processes customer orders. Having limited requirements , the problem itself could have gone in many directions. In general , the guide was the parts from the requirement and the norm.

In order to understand what had to be done I analyzed the application domain and determined at first what each from the minimal classes should do.

Basically, the application has two possibilities. Either you enter as an administrator or as a customer. For each possibility there is a different User Interface with different operations.

Thus when you start the application there is a login window.

In short , for my project the class Order will have initialized in the constructor int orderID, Customer customer, Product product, int quantity. All orders will be placed in a Binary Search Tree. The class OPDept (Order Processing Department) is the class where it occupies with the storing of the orders. The class Customer will have initialized in the constructor String name, int customerID. This will be a customer from the real world. The class Product will depict the real life product. It will have initialized in the constructor int productID, String name, String description, int stock. Finally , the minimal class in my project , the class Warehouse stores the products in a Binary Search Tree.

As for use cases, there will be two, one for the administrator use case, and the other for a regular user (which we will refer to as customer).

Scenarios and Use cases

The scenarios are that you either login as an administrator or as a customer. With the administrator you can search , add , delete, update , increase , decrease , view over-stock , view under-stock and of course clear filter. With the customer you can search , clear filter , order or view history of orders.

3.Projection

UML Diagrams

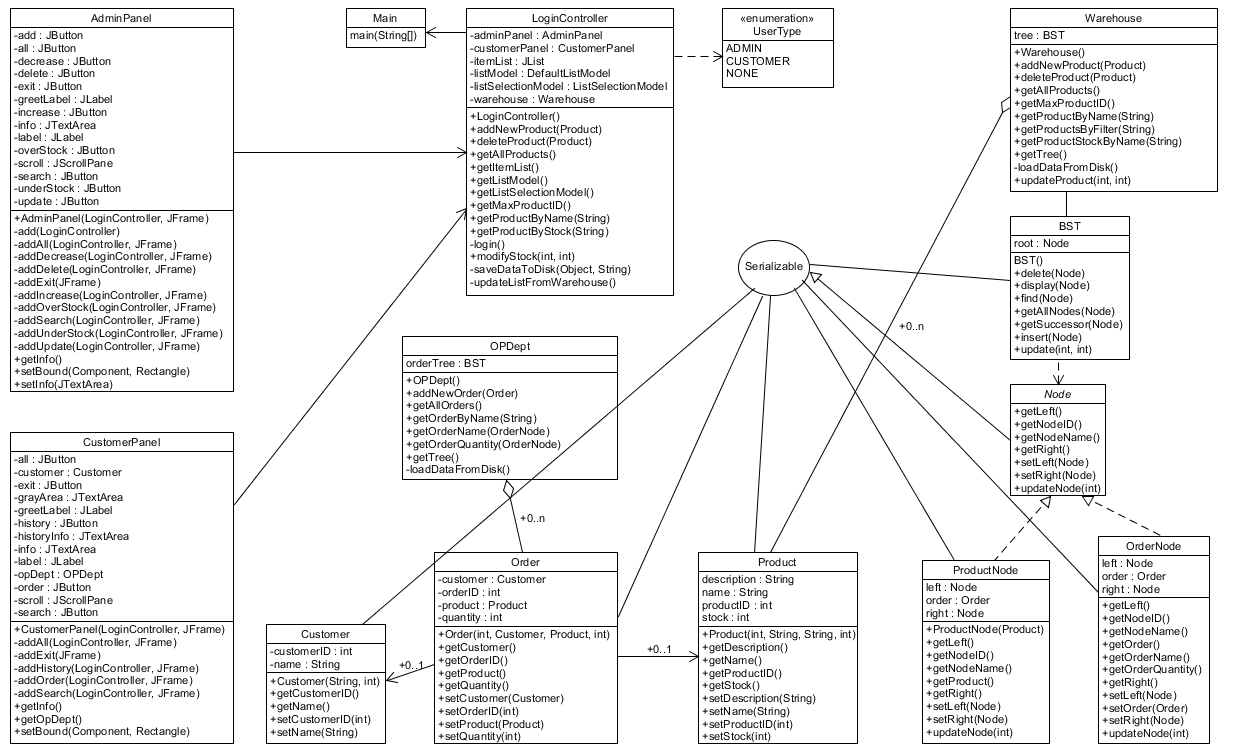
1. User Case Diagrams

Administrator

This diagram shows the actor (administrator) and what he can perform using the application.

Customer

This diagram shows the actor (customer) and what he can perform using the application.

1. Class Diagram

Class projection

Packages

The classes chosen for this project are specific and each one of them handles an important part of the program.

1. app package

This package contains the Main class. From here we start our Controller.

Main Class : this class contains the main, in which we create a new LoginController.

1. app.data package

This package contrain the OPDept class , Warehouse class.

The OPDept class : Represents the Order Processing Department which is in charge of processing the order of our application. This class uses a Binary Search Tree in order to store the orders. We also provide the methods in order to interact with the tree.

The Warehouse class: Represents the Warehouse which is responsible for keeping the Products and their corresponding stock. This class uses a Binary Search Tree in order to store the products. We also provide means to create, updated and delete products.

1. app.graphic package

This package contains the classes that are related to the graphical user interface.

The AdminPanel class : Class that is used for creating the panel visible by the admin user.

The CustomerPanel class : Class that is used for creating the panel visible by the customer user.

The LoginController class : Class that holds the main functionality for our application. Here we create the main JFrame in which a certain panel is attached based on the result of the login. We can have two possible views, one for the admin and one for the customer. Also here we serialize data from our Order and Product model to the disk via our WindowListener.

The UserType enum : Enum used for representing user logins outcomes.

1. app.model package

In this package we have the Customer , Order and Product classes.

The CustomerPanel class : Class used to represent an actual customer.

The Order class : Represents an Order. For each we need a customer,a product and the quantity.

The Product class : Represents a Product. For each we need a name, description and initial stock.

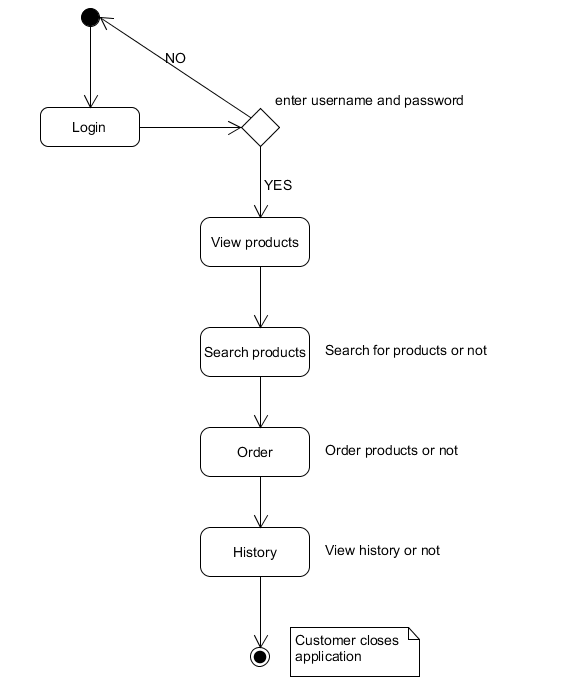
1. app.model.tree package

This package contains the BST , Node , OrderNode , ProductNode classes

The Node class : Interface that provides basic functionality for our tree elements.

The OrderNode class : Class that implements Node for storing orders in our Binary Search Tree. The ProductNode class : Class that implements Node for storing products in our Binary Search.

Activiy diagram:

The BST class : Class responsible for creating a Binary Search Tree. Each tree item is represented by a Node instance.

Activity diagrams are graphical representations of workflows of stepwise activities and actions[1] with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows).[2][3] Activity diagrams show the overall flow of control.

Activity diagrams are constructed from a limited number of shapes, connected with arrows.[4] The most important shape types:

rounded rectangles represent actions;

diamonds represent decisions;

bars represent the start (split) or end (join) of concurrent activities;

a black circle represents the start (initial state) of the workflow;

an encircled black circle represents the end (final state).

Arrows run from the start towards the end and represent the order in which activities happen.

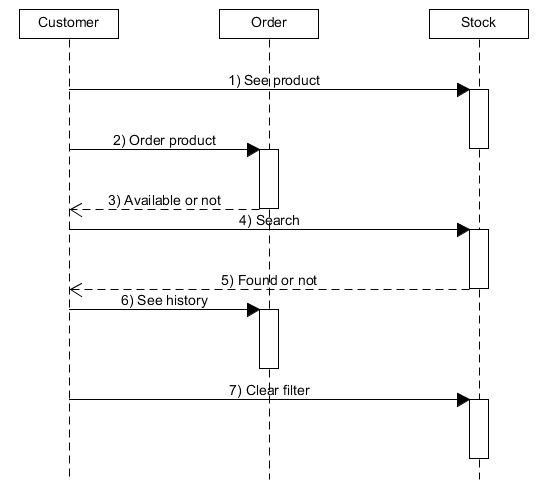
Activity diagrams may be regarded as a form of flowchart. Typical flowchart techniques lack constructs for expressing concurrency.[5] However, the join and split symbols in activity diagrams only resolve this for simple cases; the meaning of the model is not clear when they are arbitrarily combined with decisions or loops.

While in UML 1.x, activity diagrams were a specialized form of state diagrams,[6] in UML 2.x, the activity diagrams were reformalized to be based on Petri net-like semantics, increasing the scope of situations that can be modeled using activity diagrams.[7] These changes cause many UML 1.x activity diagrams to be interpreted differently in UML 2.x.

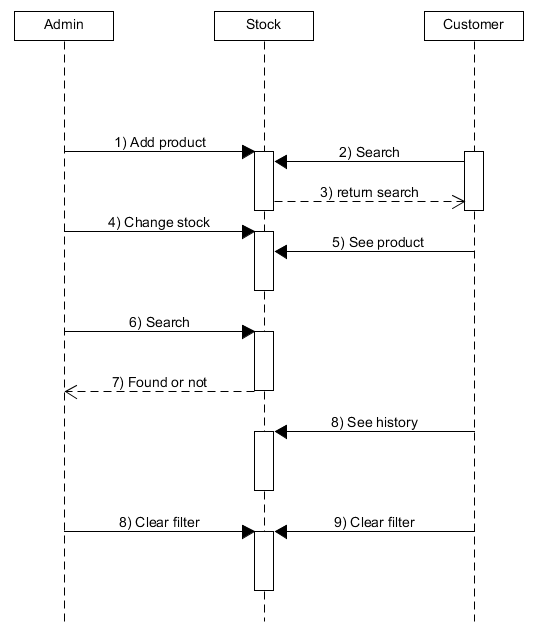
UML activity diagrams in version 2.x can be used in various domains, i.e. in design of embedded systems. It is possible to verify such a specification using model checking technique.[8]

For my personal activity diagram, very simple logic is implied. If the login fails it will as for user and password again. If successful it will open the application where the customer can see the products , order products , search for products, clear filetrs or see his own history.

Sequence diagram nr. 1 First sequence diagram is between the instances Customer,

order, and stock.

A Sequence diagram is an [interaction diagram](https://en.wikipedia.org/wiki/Interaction_diagram) that shows how processes operate with one another and in what order. It is a construct of a [Message Sequence Chart](https://en.wikipedia.org/wiki/Message_Sequence_Chart). A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios.A sequence diagram shows, as parallel vertical lines (*lifelines*), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.

Sequence diagram nr. 2

The second sequence diagram features the Admin , stock and customer

If the lifeline is that of an object, it demonstrates a role. Leaving the instance name blank can represent anonymous and unnamed instances.

Messages, written with horizontal arrows with the message name written above them, display interaction. Solid arrow heads represent synchronous calls, open arrow heads represent asynchronous messages, and dashed lines represent reply messages.[1] If a caller sends a synchronous message, it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and doesn’t have to wait for a response. Asynchronous calls are present in multithreaded applications and in message-oriented middleware. Activation boxes, or method-call boxes, are opaque rectangles drawn on top of lifelines to represent that processes are being performed in response to the message (ExecutionSpecifications in UML).

User Interface

For user interface, I used a frame which a panel gets attached depending on who logs in.

4. Implementation and testing

For implementing the data base part, I used serialization and deserialization.

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

Algortthms

I created this method to load data from disk:

**private** **void** loadDataFromDisk() {

**try** {

FileInputStream fileIn = **new** FileInputStream("C:\\products.ser");

ObjectInputStream in = **new** ObjectInputStream(fileIn);

**this**.tree = (BST) in.readObject();

in.close();

fileIn.close();

} **catch** (IOException e) {

System.***out***.println("FIle not found");

**return**;

} **catch** (ClassNotFoundException c) {

System.***out***.println("Tree class was not found");

c.printStackTrace();

**return**;

}

}

I created this method to save the data to the disk:

**protected** **void** saveDataToDisk(Object object, String path) {

**try** {

FileOutputStream fileOut = **new** FileOutputStream(path);

ObjectOutputStream out = **new** ObjectOutputStream(fileOut);

out.writeObject(object);

out.close();

fileOut.close();

System.***out***.printf("Serialized data is saved in " + path + "\n");

} **catch** (IOException i) {

i.printStackTrace();

}

}

Also for helping me save I used the Window Listener.

The listener interface for receiving window events. The class that is interested in processing a window event either implements this interface (and all the methods it contains) or extends the abstract WindowAdapter class (overriding only the methods of interest). The listener object created from that class is then registered with a Window using the window's addWindowListener method. When the window's status changes by virtue of being opened, closed, activated or deactivated, iconified or deiconified, the relevant method in the listener object is invoked, and the WindowEvent is passed to it.

frame.addWindowListener(**new** WindowAdapter() {

**public** **void** windowClosing(WindowEvent e) {

saveDataToDisk(warehouse.getTree(), "C:\\products.ser");

**if** (result.equals(UserType.***CUSTOMER***)) {

saveDataToDisk(opDept.getTree(), "C:\\orders.ser");

}

}

});

}

In order to show the name , stock and description of an item I made an inner method ListSelectionListener.

**if** (result.equals(UserType.***ADMIN***)) {

ListSelectionListener listSelectionListener = **new** ListSelectionListener() {

**public** **void** valueChanged(ListSelectionEvent listSelectionEvent) {

**if** (itemList.getSelectedValue() != **null**) {

Product product = getProductByName((String) itemList.getSelectedValue());

**if** (product != **null**) {

adminPanel.getInfo()

.setText("Product Name: " + product.getName() + "\nStock: " + product.getStock() + "\nDescription: " + product.getDescription());

}

} **else** {

adminPanel.getInfo().setText("");

}

}

};

itemList.addListSelectionListener(listSelectionListener);

} **else** **if** (result.equals(UserType.***CUSTOMER***)) {

ListSelectionListener listSelectionListener = **new** ListSelectionListener() {

**public** **void** valueChanged(ListSelectionEvent listSelectionEvent) {

**if** (itemList.getSelectedValue() != **null**) {

Product product = getProductByName((String) itemList.getSelectedValue());

**if** (product != **null**) {

customerPanel.getInfo().setText("Product Name: " + product.getName() + "\nStock: "

+ product.getStock() + "\nDescription: " + product.getDescription());

}

} **else** {

customerPanel.getInfo().setText("");

}

}

};

itemList.addListSelectionListener(listSelectionListener);

}

For making the search with filters I created this method:

**public** List<Node> getProductsByFilter(String filter) {

List<Node> allItems = getAllProducts();

List<Node> result = **new** ArrayList<>();

**for** (Node currentNode : allItems) {

**if** (currentNode.getNodeName().toLowerCase().contains(filter.toLowerCase())) {

result.add(currentNode);

}

}

**return** result;

}

5. Results

The results are as expected, having respected the requirement the program starts by asking the username and the password and considering it , it opens the same frame but with different panels.

Seeing from the use case of admin , he can manipulate the products and the stock with the features he has. As for the customer he may order a product , search for products or just see his history of orders.

The relationship between the classes can be seen through the UML class diagram where the arrows are connecting certain classes.

The search button for each admin or customer has a filter meaning that if you search just by a letter it will output every product that has that letter in its name.

I created an abstract class Node to use it when searching the Binary Search Tree in order to create a Binary Search Tree storing products and one storing orders.

The ProductNode and OrderNode both implement the abstract class Node. This allows me to use a single class for the tree because initially I had just for products and I realized I cannot use the same logic for storing the orders.

The LoginController is the most important class. From it the login starts and creates the frame and also the instance of the operational processing department also the list and the listModel.

This class also creates the AdminPanel and the CustomerPanel.

It also creates our only customer. The login() method from this class is done recursively. It could be done differently by creating a separate class but I figured out how to create it using the enum and calling it indefinitely until a correct user name and password is written in the fiels, otherwise it will pop-up a simple error message:

JOptionPane.showMessageDialog(this.getItemList(), "Invalid user or password");

Then the login window will appear again to ask for the username and password and if it succeeds it enters the application .

6. Conclusions

I tried to do as object oriented programming as possible using inherience, interface, encapsulation etc.

My java skills improved considerably by putting a lot of effort and time ino making this project. I am personally content about the results and how it went. The evolution is very interesting. I also enjoyed making a simulation of ordering products because object oriented programming is for me the best.

7. Bibliography

<http://www.tutorialspoint.com/java/java_serialization.htm>

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

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

<https://docs.oracle.com/javase/7/docs/api/java/awt/event/WindowListener.htm>