March 2016

Laboratory Assignment 2

Order Management System

Farcas Silviu Vlad

30425/1

Contents

1. Introduction

1.1 Task Objective

The task of this assignment is defined as follows:

TP Lab – Homework 2

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.*

Our order processing application should thus handle basic market acquisition operations such as buying different quantities of different products, searching for specific items or viewing recent order status and overall order history. Also, admins would have the possibility of modifying stock of items and deleting/adding new products.

1.2. Personal Approach

The aim of this paper is to present one way of implementing the required system, based on a simple and minimal GUI that provides the user with all possibilities described above. The user would log in either as an admin, or a customer empowered with the specific capabilities. That is, for admin, adding/deleting products and modifying stock and for customer, filtering products, buying products and viewing order history. The GUI is based on a FrameStack class which enables going back and forth through windows using a simple "Back" button, similar to mobile applications. Thus, in the same run of the program one can easily log in as an admin, then switch to customer and so on and so forth.

2. Problem Description

2.1. Problem Analysis

E-commerce has been around for roughly 15 years and it's a booming sector of worldwide market economy. There are many customer oriented websites or application, and our system intends to mimic some of those.

First things first, we will need products. What are products? Products are items belonging to the marketplace mainly described by name and price. Then, we will need a place to store the products. We call it a "Warehouse". In a Warehouse, every product is associated with a stock (the number of products available for selling at a given time).

Secondly, we need orders. What is an order? It is a collection of products, of different quantities, that is uniquely identified by an ID and that possesses a status (pending, processing, delivered, failled). Orders are stored with the help of an "Order Processing Department" (OPDept).

The GUI should be capable of making a distinction between admins and customers (based on a username/password login.) The User should be thus of different kinds: "Admin" and "Customer". After login, each view should be differentiated depending on the user.

The operations requested by the problem can be attributed to different kind of users, namely two: "admins" and "users". For the sake of simplicity we consider only one user and only one admin, but this app could easily be extended to accommodate multi-user paradigm. This is a starting point for future developments.

2.2 Modeling

A product is described by name and price. An Order is described by an ID, a Status, and an ArrayList of OrderItems. A Status is described by the timeReceived (the time when the order was received), the timeProcessed (the time when the order has finished processing), the timeDelivered (the time the order was successfully delivered) or timeFailled (the time the order has failed to be delivered). An OrderItem is described by a Product and a quantity. A ProductStock is described by a Product and a stock. The Warehouse is composed of a TreeSet of ProductStocks and the OPDept is composed of a TreeSet of Orders. A User can be an Admin or a Customer. The UserRepository has an Admin and a Customer. Each User is associated to a username, a password and a type of UserType type.

2.3 Scenarios

One can log in either as a user or a customer. From here, multiple possible scenarios emerge.

2.3.1. Admin

The admin wants to modify the stock, for example. Next to the desired product to be modified, they press the “+” for increasing the stock or “-” for decreasing the stock. Or they can press “x” for deleting the product from the warehouse. Also, if they want to add a product, they need to specify the name, price and initial stock for the product and press the “Add” button. All the modification are visible instantly on screen, due to the refresh() method which we will address later. However, the system does not check whether the input name, price and stock are syntactically correct, that is, if price is a double, if stock is an integer etc. It also doesn’t check whether the user has entered any data at all. This could be a starting point for future development. At any time the admin can press

"Back" and is redirected to the main LogIn page, where they can switch to the customer view.

2.3.2. Customer

Logging in as a customer requires, as usual, a username and a pass. If correct, the user is directed to the customer homepage, where we can see three buttons: "Search", "Buy", "History". If the user wants to search for specific products, they must type the name of the product in the search bar and press Search. If the user enters nothing, all products are displayed. Then, if the customer wants to buy a product, they press the "+" button to the right of the desired product. They can also press "-" in case they want to decrement the quantity of products to be bought. When the user has selected all products to be bought, they press the Buy button, which instantaneously create a new order into the OPDept and add it to the History. The button also resets all the quantities of the selected products. If the user wants to trace their current order, or view past orders, they must press the "History" button, which will open a new window with all orders, that display the status of orders in real-time. The user can press "Back" to return to the main Customer view. The user can press "Back" to return to the main LogIn screen.

2.4 Use cases

They are three possible actors in our use case: a regular user, which can be either an admin or a customer. The regular user can log in as an admin or a customer. The admin can:

- increase stock: for the selected item, the stock is increased inside the warehouse.
- decrease stock: for the selected item, the stock is decreased inside the warehouse.
- add products: enter a name, a price and an initial stock
- delete products: delete the selected items.

The customer can:

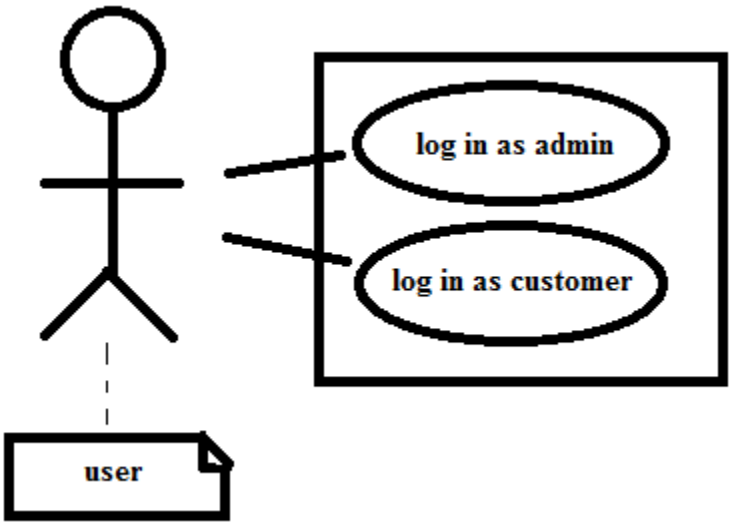
- search for items: filter products in warehouse based on name.
- buy products: increase or decrease the quantity of the desired product and then buy them. This refreshes the screen.
- view order history: view the real-time updated status of all the ordered products.

3. Design

3.1. UML Diagrams

In the following we will present Use Case diagrams, Class Diagram, Sequence Diagrams and Activity Diagram.

3.1.1. Use Case Diagrams



Use Case Description

The user can log in as admin or customer.

Trigger

The user presses submit.

Actors

The user

Preconditions

The username and password are defined in Users file.

Goals (Successful Conclusion)

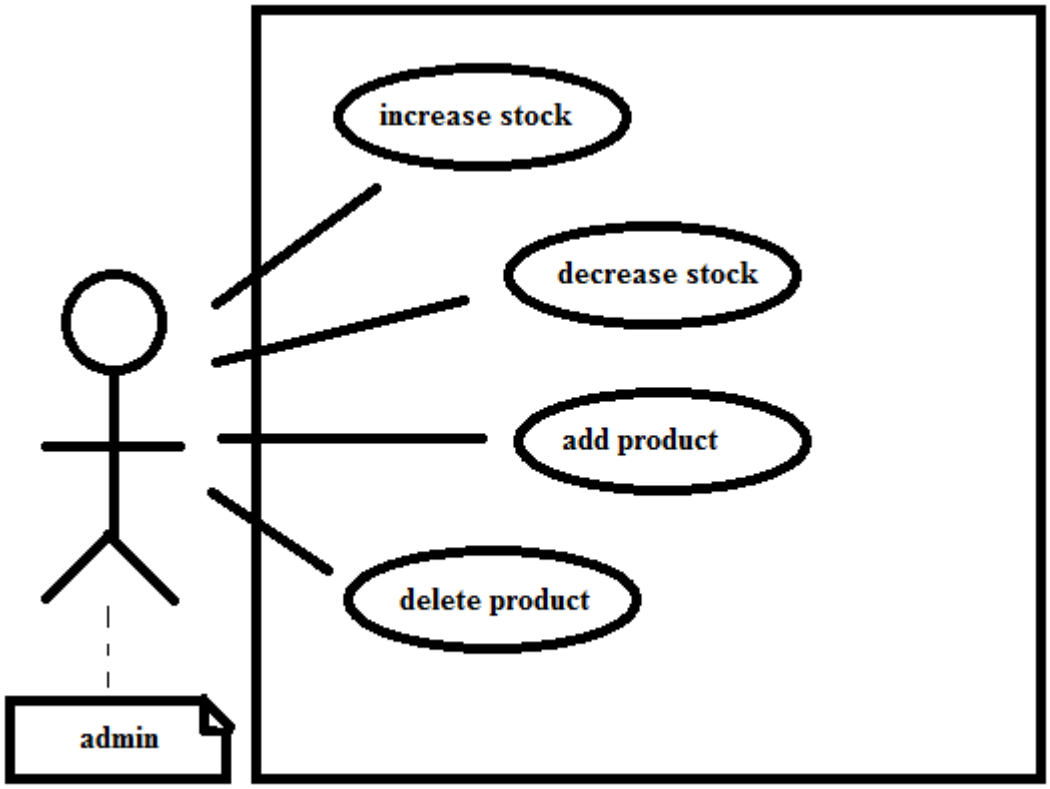
User authenticates as admin or customer

Failed Conclusion

User doesn’t authenticate.

Steps of execution

1. The user enters username and password.
2. The user presses submit.



Use Case Description

The admin views products. The admin can increase/decrease stock or add/delete product.

Trigger

The admin presses a button.

Actors

The admin

Preconditions

--

Goals (Successful Conclusion)

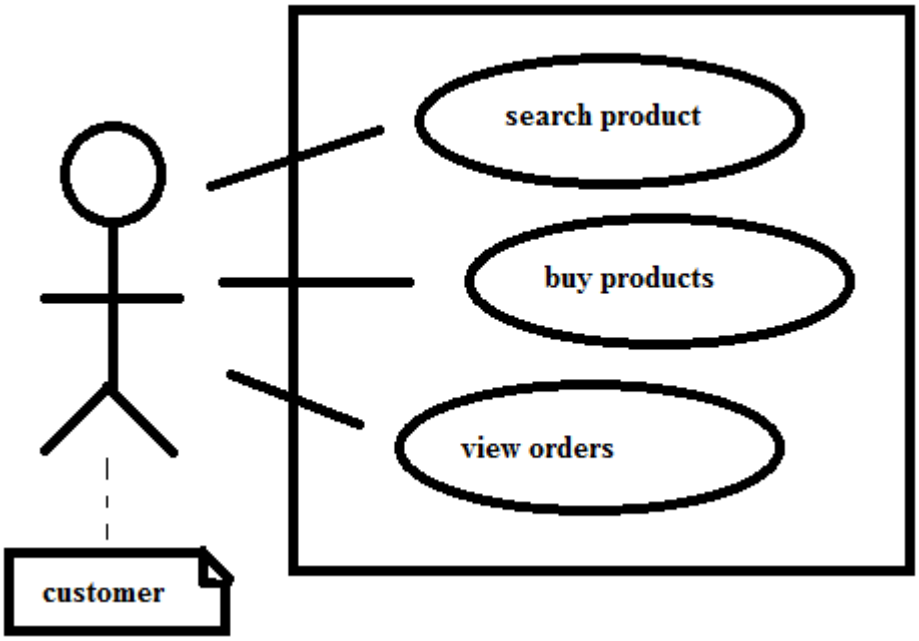
Modify product information in warehouse.

Failed Conclusion

--

Steps of execution

3. The admin presses a button
4. The requested operation is performed.



Use Case Description

The customer searches for products. The customer modifies the desired quantity. The customer can buy products selected. The customer can view history of orders.

Trigger

The customer presses a button.

Actors

The customer

Preconditions

For buying products, the customer needs to search for them

Goals (Successful Conclusion)

Customer can view searched items. Customer can buy selected products. Customer can view order history + keep track of order status.

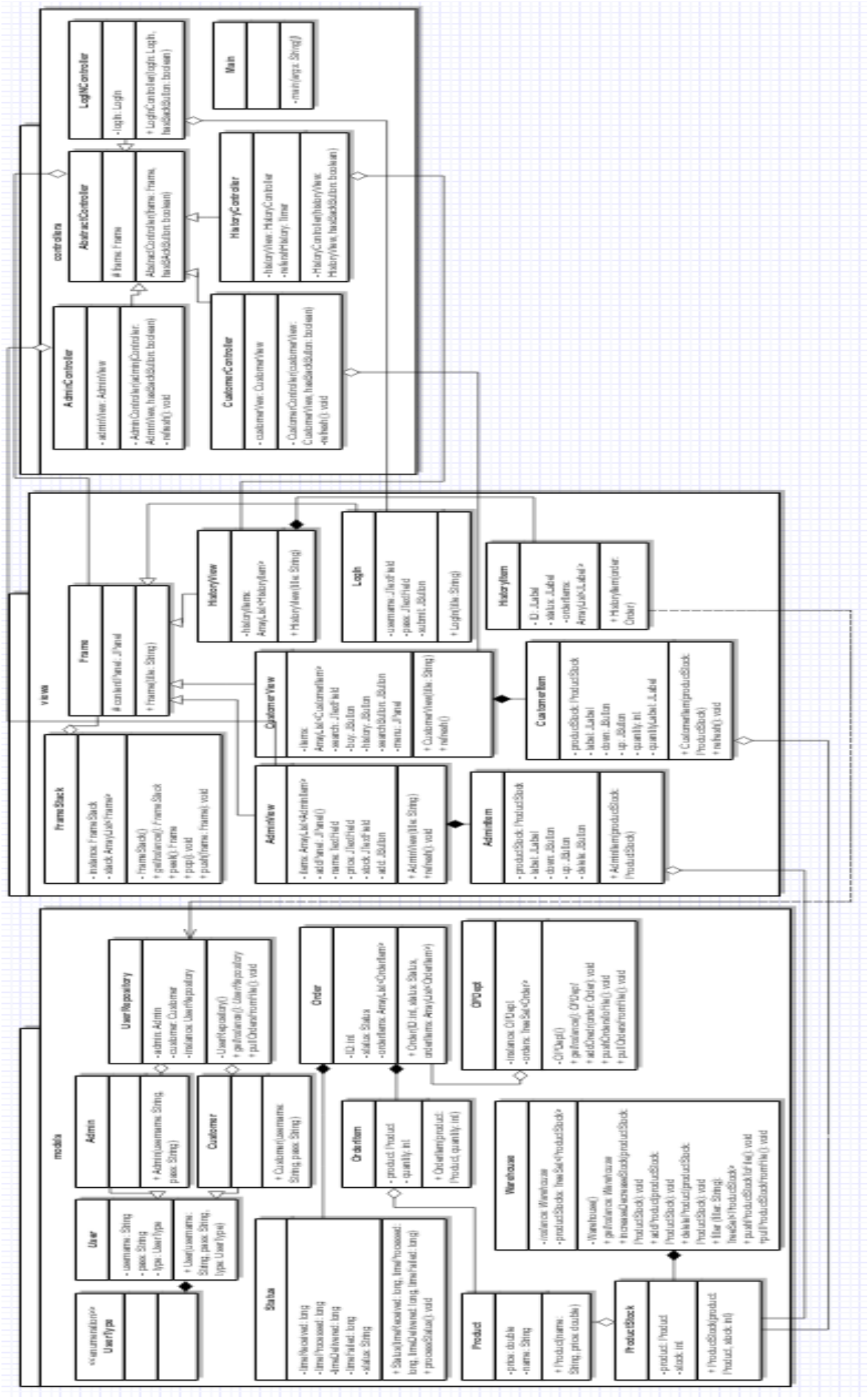
Failed Conclusion

--

Steps of execution

1. The customer enters search name.
2. The customer presses the search button.
3. The customer presses “+” or “-” for modifying the product quantity.
4. The customer presses Buy button.
5. The customer presses History button.
6. The customer keeps track of his orders.

1.1.2. Class Diagram (next page)


controllers
LogINController
- logIn: LogIn
+ LogInController(logIn: LogIn, hasBackButton: boolean)
Main
- main(args: String[])
AbstractController
frame: Frame
- AbstractController(frame: Frame, hasBackButton: boolean)
HistoryController
- historyView: HistoryController
- refreshHistory: timer
- HistoryController(historyView: HistoryView, hasBackButton: boolean)
AdminController
- adminView: AdminView
- AdminController(adminController: AdminView, hasBackButton: boolean)
- refresh(): void
CustomerController
- customerVar: CustomerView
- CustomerController(customerView: CustomerView, hasBackButton: boolean)
- refresh(): void
views
Frame
contentPanel: JPanel
+ Frame(title: String)
HistoryView
- historyItems: ArrayList<HistoryItem>
+ HistoryView(title: String)
LogIn
- username: JTextField
- pass: JTextField
- submit: JButton
+ LogIn(title: String)
HistoryItem
- ID: JLabel
- status: JLabel
- orderItems: ArrayList<JLabel>
+ HistoryItem(order: Order)
FrameStack
- instance: FrameStack
- stack: ArrayList<Frame>
- FrameStack()
+ getInstance(): FrameStack
+ peek(): Frame
+ pop(): void
+ push(frame: Frame): void
CustomerView
- items: ArrayList<CustomerItem>
- search: JTextField
- buy: JButton
- history: JButton
- searchButton: JButton
- menu: JPanel
+ CustomerView(title: String)
+ refresh()
CustomerItem
- productStock: ProductStock
- label: JLabel
- down: JButton
- up: JButton
- quantity: int
- quantityLabel: JLabel
+ CustomerItem(productStock: ProductStock)
+ refresh(): void
AdminView
- items: ArrayList<AdminItem>
- addPanel: JPanel()
- name: TextField
- price: JTextField
- stock: JTextField
- add: JButton
+ AdminView(title: String)
+ refresh(): void
AdminItem
- productStock: ProductStock
- label: JLabel
- down: JButton
- up: JButton
- delete: JButton
+ AdminItem(productStock: ProductStock)
models
UserRepository
- admin: Admin
- customer: Customer
- instance: UserRepository
- UserRepository()
+ getInstance(): UserRepository
+ putOrders(name): void
Order
- ID: int
- status: Status
- orderItems: ArrayList<OrderItem>
+ Order(ID: int, status: Status, orderItems: ArrayList<OrderItem>)
CPUapt
- instance: CPUapt
- orders: tree Set<Order>
- CPUapt()
+ getInstance(): CPUapt
+ addOrder(order: Order): void
+ pushOrders(): void
+ putOrders(name): void
Admin
+ Admin(username: String, pass: String)
Customer
- Customer(username: String, pass: String)
OrderItem
- product: Product
- quantity: int
+ OrderItem(product: Product, quantity: int)
Warehouse
- instance: Warehouse
- productStocks: tree Set<ProductStock>
- Warehouse()
+ getInstance: Warehouse
+ increaseQuantity(productStock: ProductStock): void
+ addProduct(productStock: ProductStock): void
+ deleteProduct(productStock: ProductStock): void
+ filter(filter: String): treeSet<ProductStock>
+ pushProductStock(): void
+ pullProductStock(): void
User
- username: String
- pass: String
- type: UserType
+ User(username: String, pass: String, type: UserType)
«enumeration»
UserType
Status
- timeReceived: long
- timeProcessed: long
- timeDelivered: long
- timeFailed: long
- status: String
+ Status(timeReceived: long, timeProcessed: long, timeDelivered: long, timeFailed: long)
+ processStatus(): void
Product
- price: double
- name: String
+ Product(name: String, price: double)
ProductStock
- product: Product
- stock: int
+ ProductStock(product: Product, stock: int)

The class diagram shows the modelling of our problem into classes. There are 26 classes, one abstract class and one enumeration. One must know that not all fields and methods were represented on the diagram, only the ones that are important to our implementation (for example, getters/setters are not shown). Also, not all dependencies have been drawn, otherwise it would make the diagram unreadable. I have used inheritance for

1) class Admin and Customer, which inherit abstract class Use;
2) classes LogIn, AdminView, CustomerView, HistoryView which inherit class Frame
3) classes LogInController, CustomerController, AdminController, HistoryController which inherit class AbstractController

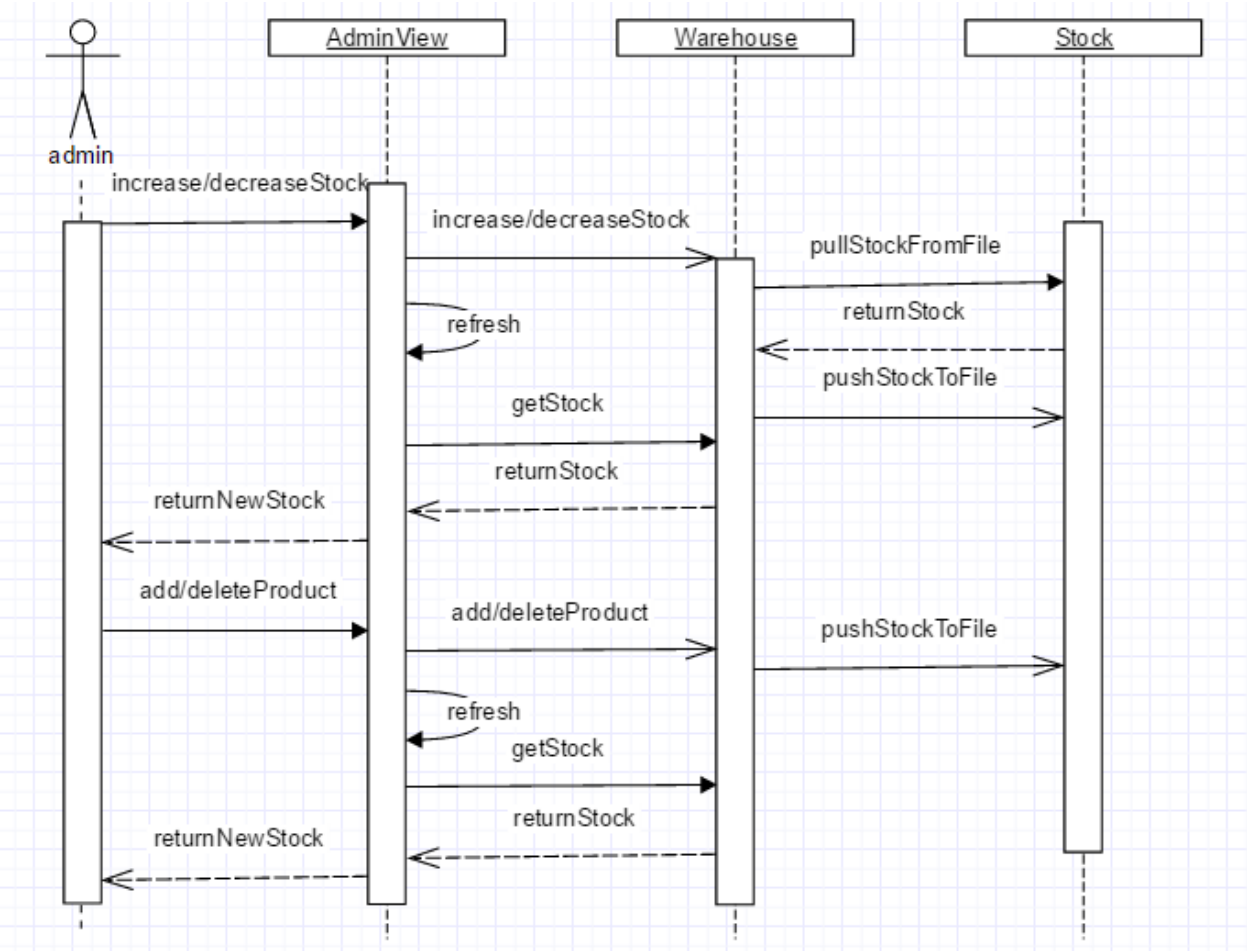
I have used aggregation/composition between:

1) UserRepository has an Admin and a User
2) OrderItem has a Product
3) Order has a Status an many OrderItems
4) OPDept has many Orders
5) ProductStock has a Product
6) Warehouse has many ProductStocks
7) FrameStack has many Frames
8) AdminItem ahs a ProductStock
9) CustomerItem has a ProductStock
10) AdminView has many AdminItems
11) CustomerView has many CustomerItems
12) HistoryView has many HistoryItems
13) AbstractController has a Frame
14) CustomerController has a CustomerView
15) AdminController has an AdminView

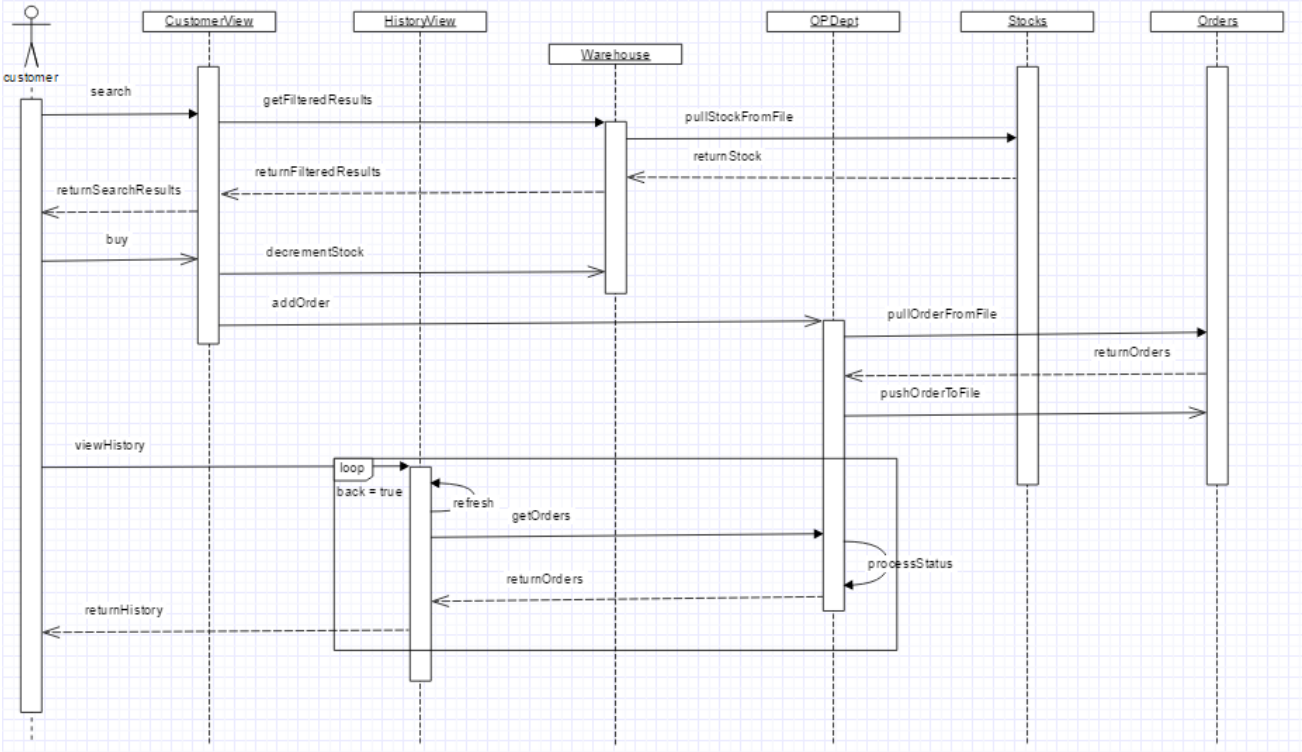
16) LogInController has a LogInView
17) HistoryController has a HistoryView

The rest of the relations are dependencies.

1.1.3. Sequence Diagrams

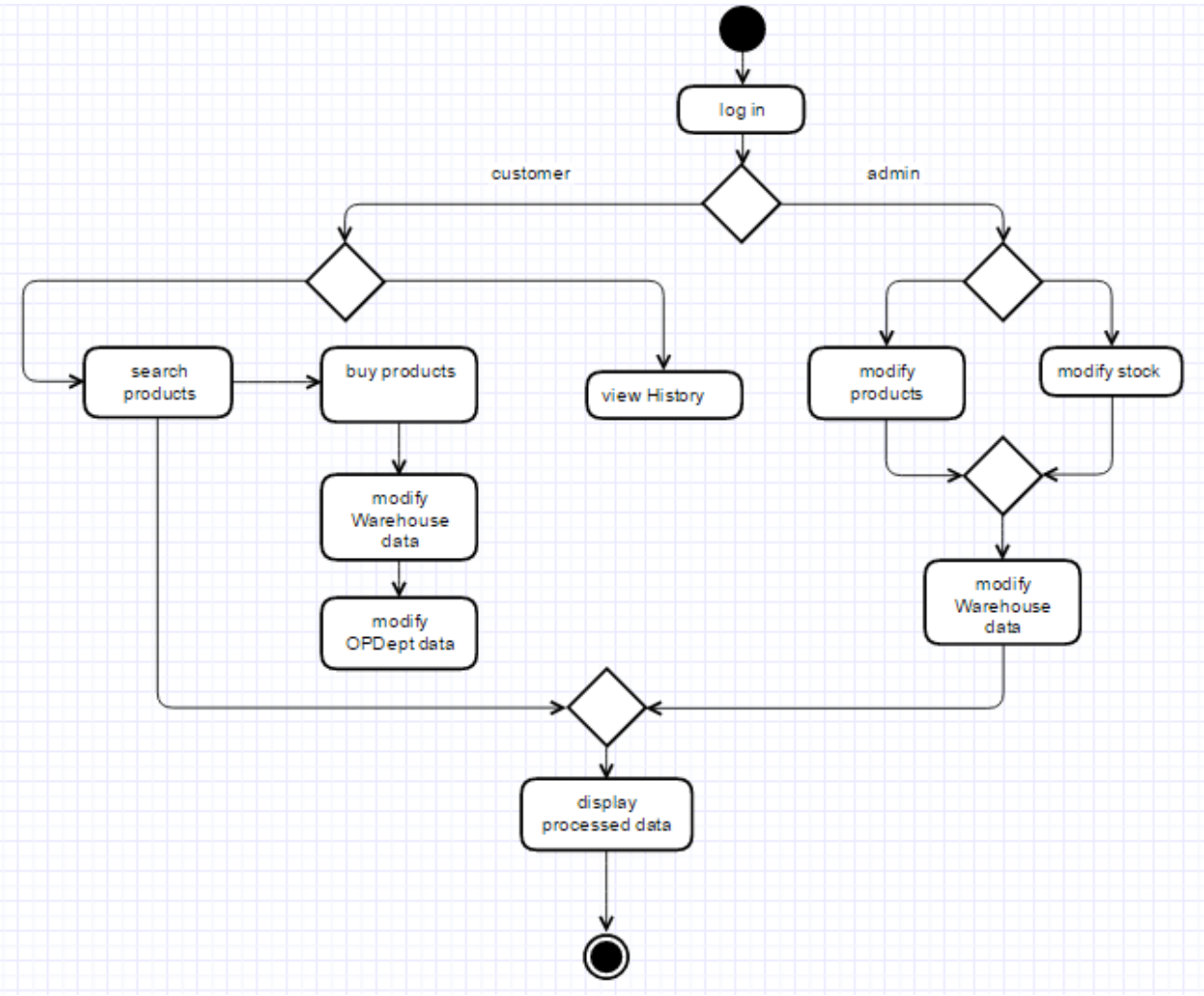


The sequence diagrams shows a possible turnout of the user – machine interaction**.** Here, the admin can increase/decrease stock, or add/delete product.



The customer can search, buy or view history.

1.1.4.Activity Diagram (next page)



1.2. Data Structures

Besides primitives type, like int, double and String our project heavily makes use of Data Structures from the Java Collections Framework. Among other collections such as ArrayLists, the key Data Structure to our program is the TreeSet collection. The problem description requires us to utilize a Binary Search Tree for storing orders. TreeSet is an implementation of a red-black tree, which is a Balanced Binary Search Tree. TreeSet makes operations such as adding, deleting and searching perform in O(log n). It compares objects by using a comparator mentioned in the constructor. For our problem, we use a

TreeSet<Order> and a TreeSet<ProductStock>. The comparators for these TreeSet are fully provided below:

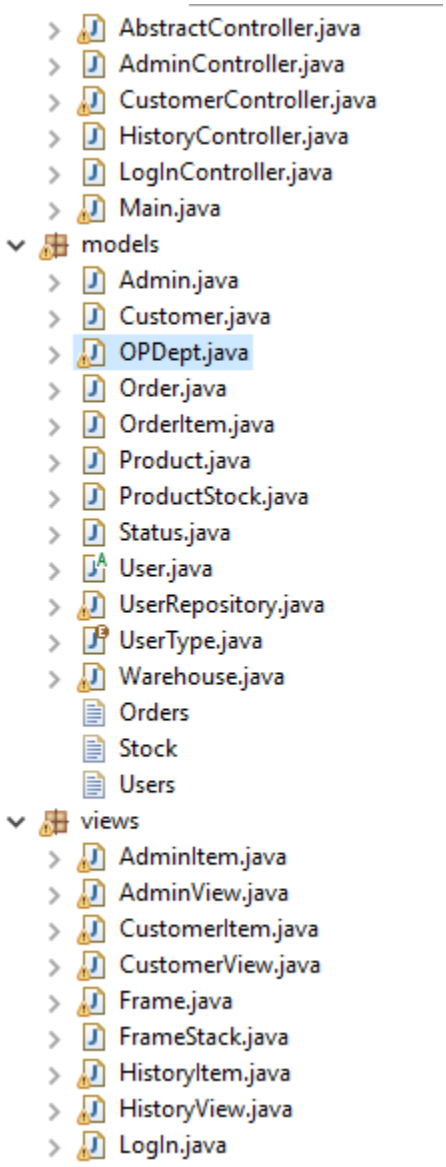
```
public class MyComparator implements Comparator<Order> {

		@Override
		public int compare(Order o1, Order o2) {
			return String.valueOf(o1.getStatus().getTimeReceived())

	.compareTo(String.valueOf(o2.getStatus().getTimeReceived()));
		}
	}



public class MyComparator implements Comparator<ProductStock> {

		@Override
		public int compare(ProductStock p1, ProductStock p2) {
			return
p1.getProduct().getName().compareTo(p2.getProduct().getName());
		}
	}
```

1.3. Class Design

The problem domain was decomposed into sub-problems, and for each sub-problem a solution was build. By assembling the solutions we obtain the general solution to our problem. The sub-problems were:

- how to correctly model an order processing system?
- how will be implemented the user interface?
- how can we assure user interaction with our models?

For each of the above mentioned, we came up with a package that would support the features of order processing system, namely increasing/decreasing orders, adding/removing products, buying, searching and viewing history, the user interface and user interaction with our model. These packages follow the Model-View-Controller software architecture and they are as follows:



The UserType enum has two members: ADMIN and CUSTOMER. The **User** class is an abstract class that has a username, a pass and a type of type UserType. The **Admin** and **Customer** classes are extensions of the User class. The **UserRepository** class has an Admin and a Customer, and can write changes and load users from Users file. The UserRepository is implemented using the Singleton Pattern, that is, only an instance of the UserRepository is possible to create. This creation is

done in the authentication process when the user submit a username and a pass.

The **Product** class has a name and a price. The **ProductStock** class has a Product and a stock. The ProductStock class only makes sense when associated with the Warehouse class. The **OrderItem** class has a Product and a quantity. It only makes sense when associated to an Order. The **Status** class has many time related fields (timeReceived, timeProcessed, timeDelivered / timeFailed). It also makes use of an ingenious processStatus() algorithm, which we will detail later, in the Algorithms section.

The **Order** class has an ID, a status, and an ArrayList of OrderItems. The order is perfectly described by its ID, and additionally we add a status and a list of products and their quantities.

The **OPDept** class is similar to the UserRepository in that it uses a file to write all information on and at initialization, it pulls from the file all the orders that were previously processed. Also, it is similar in the sense that it is unique, hence, it is modelled with the Singleton Pattern. The only instance is created when the user buys something (adds a new order) or wants to view history of orders. The Order uses, as mentioned in section 1.2., a TreeSet to store orders, making basic operations perform in O(lgn) time. We make use of only one basic operation, which is add().

The **Warehouse** class is another Singleton build class, and it is instantiated when the admin modifies stock/products or when user buys, because this requires a modification of the stock. As mentioned in section 1.2., the Warehouse uses a TreeSet to store products and stock. The filter operation is more intriguing and will be later detailed in the Algorithms section (it makes use of regex'es). The basic operations we makes use of are: increase/decreaseStock(), add(), delete() and filter(). As before, there are operations pushProductsToFile() and

PullProductsFromFile() which perform file manipulation in order to store the products.

The **Frame** class is the basic window frame of our app. All windows the user interacts with extend Frame.

The **FrameStack** is the class that is responsible for assuring the mobile-like appearance of our app, the Back button facility. It is constructed under the Singleton Pattern, and defines three basic operations: peek(), which returns the last frame added, push(), which pushes a frame onto the stack, and pop(), which deletes a frame from the stack. Needless to say that the FrameStack actually implements a stack with the help of an ArrayList (for a basic definition of a stack, view http://users.utcluj.ro/~jim/DSA/index.html, Lecture 1). Everytime a new Frame is created, the push() operation of FrameStack is called.

The LogIn is the home screen. The user is asked for a username, a pass and the they click on the submit button when finished. If correctly entered an admin's credentials, and AdminView window appears. If correctly entered a customer's credentials, a CustomerView appears.

An **AdminItem** is a basic item extending JPanel that will be displayed on the AdminView window. It is composed of a products stock and the label associated to it, a "-", a "+" and a "x" button used for decrementing or increasing the stock or for deleting the product. An AdminItem also contains the necessary functions for setting the action listeners on these buttons.

An **AdminView** is the home window for the admin. It contains an ArrayLsit of AdminItems that compose the operating panel for the admin which will modify the contents of the Warehouse, and also contains an extra panel which is used for adding new products. This panel is composed of a name textfield, a price textfield, an initial stock textfield and an "add" button. The class also defines an interesting refresh() strategy which is later documented in the Algorithms section.

The refresh() method is called whenever we modify a stock or we add or delete a product. Also, the necessary setActionListener methods are there in this class.

A **CustomerItem** is class extending JPanel and is the basic component of a CustomerView window. It contains a ProductStock and the affiliated label, a quantity and the affiliated label and the buttons "+" and "-" for modifying the desired quantity of a product. Also, the required setActionListener methods are there in this class. We also have a refresh method for the quantity label, fully reproduced below:

```
public void refresh() {
      this.quantityLabel.setText(String.valueOf(quantity));
}
```

A **CustomerView** is the basic home window of a customer. It contains the CustomerItems, which represent buyable products the customer can acquire. It also conatins a menu JPanel that has a search textfield, a search button, a buy button and a history button. If the search button is pressed, the filter() command in the warehouse is executed with the specified filter in the search textfield. If the Buy button is pressed, a new order is added to the OPDept with the non-null quantities of the products and the quantities will be reset. When pressing History, a HistoryView window will open enabling the user to track the last order but also see previous orders. The last order always comes last in the view, since the orders are ordered in the OPDept after their timeReceived parameter. The CustomerView presents a refersh() method similar to AdminView.

A **HistoryItem** is a JPanel consisting of an ID, a status and an ArryaList of orderItems (that is, products and their quantities). It is the basic building block of a HistoryView. Each HistoryItem is constructed around an Order.

A **HistoryView** is the window displaying the HistoryItems. It has a refresh() method which is similar to AdminView.

The **AbstractController** is the main controller class which all controllers, except Main, extend from. It implements the BackButtonActionListener:

```
private class BackButtonListener implements ActionListener {
	public void actionPerformed(ActionEvent e) {
		FrameStack.getInstance().pop();
	}
}
```

The **LogInController** is the controller responsible for the welcome window LogIn. It contains a LogIn field and it implements a special SubmitButtonActionListener class which launches an AdminView or a CustomerView.

The **AdminController** contains an AdminView. It is responsible for controlling the AdminView interactions with the models. It implements all the ActionListener classes required for AdminView, plus an extra refresh() method:

```
	public void refresh() {
		for (AdminItem ai : adminView.getItems()) {
			ai.setDownButtonActionListener(new
DownButtonActionListener(ai));
			ai.setUpButtonActionListener(new
UpButtonActionListener(ai));
			ai.setDeleteButtonActionListener(new
DeleteButtonActionListener(ai));
		}
	}
```

The **CustomerController** contains a CustomerView. It is responsible for controlling the CustomerView interactions with the models. It implements all the ActionListener classes required for CustomerView, plus an extra refresh() method:

```
public void refresh() {
		for (CustomerItem ci : customerView.getItems()) {
			ci.setDownButtonActionListener(new
DownButtonActionListener(ci));
			ci.setUpButtonActionListener(new
UpButtonActionListener(ci));
		}
	}
```

The **HistoryController** contains a HistoryView and is responsible fro controlling the HistoryView interactions with the models. It has also a Timer, which refreshes periodically the window.

This concludes a quick introduction into the aspect of our class design.

1.4. Interfaces

Our design does not make use of Interfaces.

1.5. Relations

In Object Oriented Programming, it is a good habit to design classes as loosely coupled as possible, in order to avoid error propagation. I tried to stick with this principle and, with few exceptions, I think I managed to fulfill this requirement. In this design, one can found the following relations: inheritance, aggregation, dependency and innerness.

1.6. Packages

The design follows the Model-View-Controller (MVC) architecture. It has three packages: models, views, controllers.

1.7. Algorithms

In the following we will detail only the more intriguing algorithms used in our program.

Fisrt, we start with processsStatus() method in our Status class.

```
public void processStatus() {
            int random;
            if (System.currentTimeMillis() - timeReceived > 5000){
                  status = "processing";
            }
            timeProcessed = timeReceived + 5000;
            if ((System.currentTimeMillis() - timeProcessed > 5000) &&
(timeDelivered == 0 && timeFailed == 0)) {
                  random = (int) (Math.random() * 10);
                  if (random <= 8) {
                        timeDelivered = timeProcessed + 5000;
                        timeFailed = 0;
                        status = "delivered";
```

```
                } else {
                        timeDelivered = 0;
                        timeFailed = timeProcessed + 5000;
                        status = "failed";
                }
        }
        if (timeDelivered != 0) {
                status = "delivered";
        }
        if (timeFailed != 0) {
                status = "failed";
        }
    }
```

When creating a Status object, timeReceived will always be initialized to the current system time in miliseconds and the status is by default “pending”. If 5 seconds have passed, the status becomes “processing”. Then, the timeProcessed is set 5 secons after timeReceived, regardless of the actual time. If another 5 seconds have passed since timeProcessed, and we still didn’t delivered it or failed it, in 80% of the cases the status will be “delivered”, the timeDelivered is set, and timeFailed = 0. In the rest of 20% of cases, the status is “failed”, the timeFailed is set and tiemDelivered = 0. This is done using a special random variable which randomizes the delivering process. Then, in case the tiemDelivered is already set (or timeFailed), the status becomes “delivered” (or “failed”, respectively).

Another interesting algorithm would be the one that helps us to filter search results. The filter method in the Warehouse class return a TreeSet which is a subset of the products TreeSet which corresponds to the searched pattern.

```
    public TreeSet<ProductStock> filter(String filter) {
        TreeSet<ProductStock> filtered = new TreeSet<ProductStock>(new
MyComparator());
        for(ProductStock ps: productStocks){
            if(ps.getProduct().getName().matches(".*(" + filter +
").*")){
                filtered.add(ps);
            }
        }
        return filtered;
    }
```

The method makes use of regex'es.

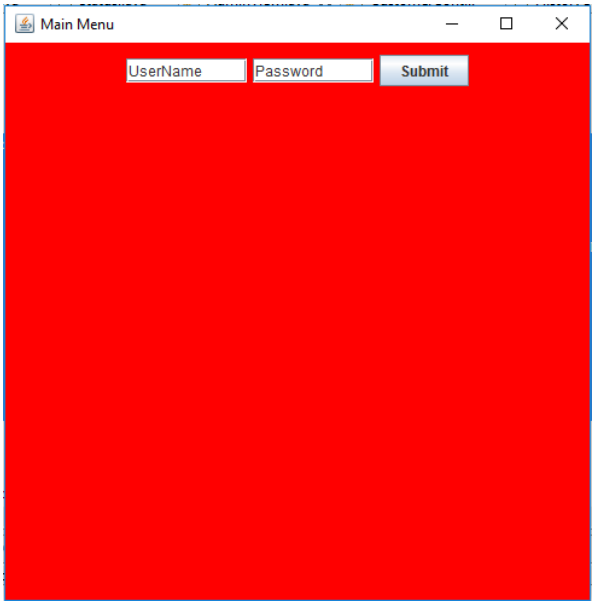
The last algorithm we are going to consider is the refresh() method present in the AdminView (with slight variations in CustomerView and HistoryView).

```
public void refresh(){
		contentPanel.removeAll();
		contentPanel.revalidate();
		this.repaint();
		items = new ArrayList<AdminItem>();
		for(ProductStock ps: Warehouse.getInstance().getProductStock()){
			items.add(new AdminItem(ps));
		}
		for(AdminItem ai: items){
			contentPanel.add(ai);
		}
		contentPanel.add(addPanel);
		contentPanel.revalidate();
		this.repaint();
	}
```

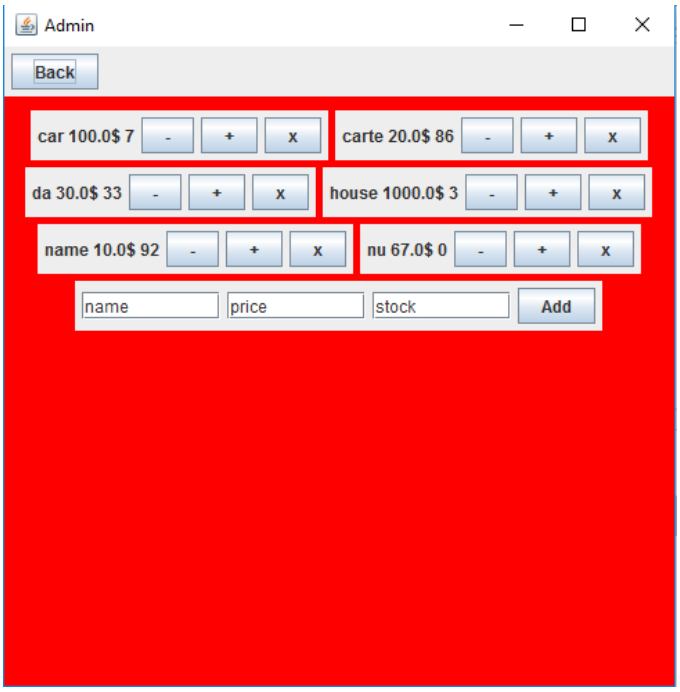
First, all the elements from the contentPanel are erased. Then, the Panel repaints. Then, we reconstruct the AdminItems from the upgraded Warehouse, and then we add all the items to the panel. Then, we repaint again.

1.8. User Interface

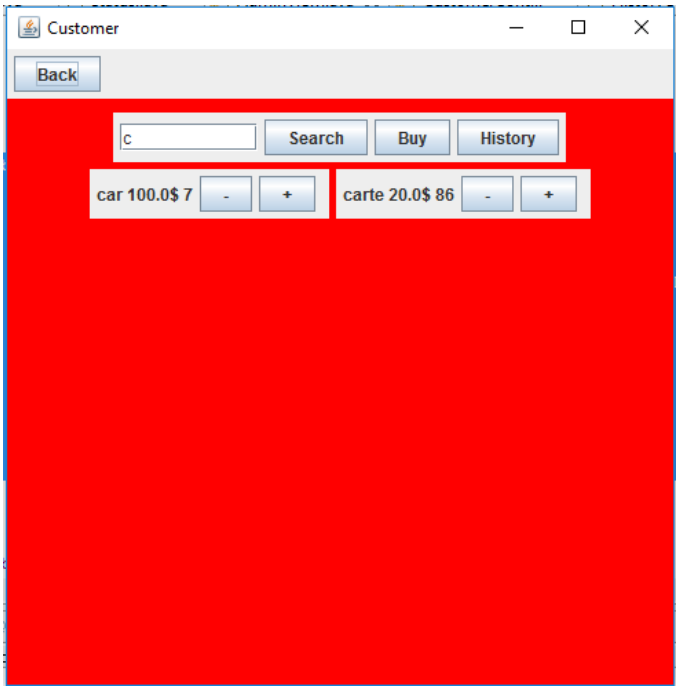
The base screen:



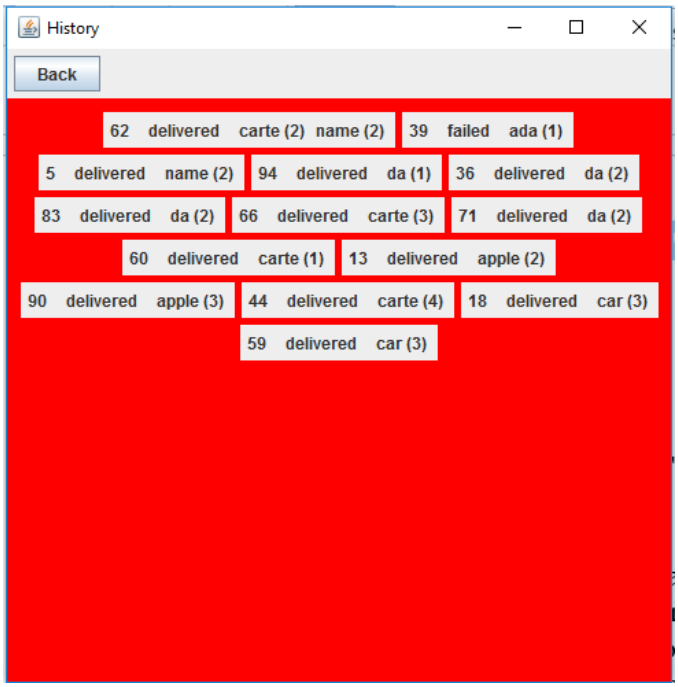
The admin view after entering admin credentials:



The customer view after going back and entering customer credentials:



The history view:



2. Implementation and Testing

The project was developed in Eclipse IDE using Java 8, on a Windows 10 platform. It should maintain its portability on any platform that has installed the SDK. It was heavily tested, but new bugs could be discovered in the near future. One of the main inconveniences are that the program is not protected against all kinds of illegal input data, only to some. This could be the subject of future development.

3. Results

The application is a user-friendly, helpful app that can perform basic stock related operations. It is a mini-version of a real e-commerce platform and is a good starting point for future developments.

4. Conclusions

4.1. What I've Learned

TIME IS PRECIOUS! I had to solve complicated problems of time management which taught me good organization skills. I learned that a good model is always a key to a successful project and that a bad model could ruin your project in the end. I learned never to get stuck on one little bug, and if I do, ask for help, either from the TA or from colleagues or on different forums. I am looking forward to apply what I have learned in future projects.

4.2. Future Developments

We could try to make the program more user – protected in that we could assume the user doesn't know to enter valid data each time. We could for example show a message if the user presses the "add" button in admin view and the price is not an integer number.

Also, the app could be extended to accommodate to the multi-user paradigm. This would necessitate that in the UserRepository Class to have an ArrayList of admins and an ArrayLsit of customers, and for each customer to define the own OPDept in order to display the orders for each customer in proper way.

5. Bibliography

[1] http://www.uml.org/

[2] Barry Burd, *Java For Dummies*, 2014

[3] Kathy Sierra, Bert Bates, *Head First Java*, 2005

[4] Steven Gutz, Matthew Robinson, Pavel Vorobiev, *Up to Speed with Swing*, 1999

[5] Joshua Bloch, *Effective Java*, 2008

[6] http://www.stackoverflow.com/

[7] http://docs.oracle.com/

[8] http://www.coderanch.com/forums

[9] Derek Banas' Channel on YouTube
https://www.youtube.com/channel/UCwRXb5dUK4cvsHbx-rGzSgw