VIA UNIVERSITY COLLEGE
ICT ENGINEERING

Warehouse Management System

Project Report

SEP-2

Bashar Gabbara (246678)
Bianca Sgondea (240312)
Mihai Cristian Pavel (245485)
Tamara Hassan (247581)

Supervisors
Henrik Kronborg Pedersen
Troels Mortensen

December 2016

Table of Contents

Abstract

Many warehouses nowadays have invested in creating a warehouse management system to help them manage their inventory. Before a management system, the process of keeping track of inventory was all manual work, allowing for a large margin of human error. Keeping track of orders by customers, the products that are available at the warehouse, the orders to the suppliers, and everything that comes with managing an inventory of a warehouse was all done by pen and paper at some point and evolved to management systems. There is much time wasted, confusion between the employees, and the level of accuracy drops. With a management system, the process will be more dynamic and quicker than usual. There will no longer be employees hired to keep track of the amount of products/pallets that exist within the warehouse; the system will take care of all that automatically. The process will include minimal errors due to eliminating human error as much as possible. The drawings and analysis and an interview with Reitan Distributions, gave us a clear idea on how a system as such should be built. The warehouse management system is a system which enables the user to set up a connection with the server to retrieve and add data to the database regarding inventory-wise. The system can be accessed by three different users, each with different functionalities; the administrator, the picker, and customer. The administrator can manage pickers, customers, suppliers, products and register incoming goods. The customer is only allowed to make an order and send it to the warehouse. The picker has a few options, to check in/check out, start an order, finish an order, and mark an order as incomplete. The system has different interfaces depending on the user which helps the system be organized and easily read.

Introduction

The world of technology is evolving day to day and more companies are integrating computer based systems to aid them in managing their systems. As companies grow, they came to the realization that having a structured system plays a vital role for the company's success.

Warehouse Management System is a system that will help any relatively small warehouse to keep track of their inventory and manage the large quantity of data going through their warehouse. All the large or small supermarkets, gas station stores and many more, rely on warehouses for their products. If warehouses use a computer based management system it would make the process easier and faster since there is no need to manually keep track of every detail happening each day. The Warehouse Management System relies on a database that stores all the information a warehouse would require to have. The database is running on a server and users (clients) may connect to the server based on who the user is.

There are three main users of the system, administrators, pickers, and customers. Each user-type has a unique predefined password to access the system. With each different login, a different interface appears based on the functionality the user has access to.

The administrator has a few more options than any of the other users. He/she can manage products, manage invoices, check the orders, manage pickers, registers incoming goods, manage suppliers, manage customers, and manage the stocks in storage.

The picker has less functionality than the administrator. In order for a picker to start on an order, he/she has to check in. The picker should also check out when they leave work that day. Other than that, the picker may start an order, finish an order, and also mark an order as incomplete if the picker has any sudden emergency and had to leave. The incompletion of an order makes the order available again for another picker to work on it where the first picker left off.

The customer has the minimal functionality; all the customer does is make an order. The customer fills out the order sheet by selecting the products they want to buy from the warehouse and indicate the amount for each product. When the customer clicks on the “Send” button, the order is sent to the administrator immediately and is ready for a picker to start working on it when the time comes to work on that order.

System Purpose

The purpose of the system is to ease the process of managing the inventory of a warehouse.

Analysis

Requirements

Functional Requirements

Product Backlog

1. As a user, we want to be able to login to the system as three different users, the admin, the customer, and the picker. We want a unique password for each type of user.
2. As a user logged in as an admin, I want to be able to do the following:
   a. Manage Products:
      i. adding a category to the inventory
      ii. removing a category
      iii. adding a new product: give it a picking position in the warehouse
      iv. removing a product

        v. check products' expiry date by entering a certain date
        vi. checking a product's information
        vii. view all products of a certain category
    b. Invoice History:
        i. List of invoices that include: Invoice number, date order was made, date and time of shipment, customer ID, prices, products that were shipped, and total amount.
        ii. I would like to be able to view the invoice details.
    c. Check Orders:
        i. I want two lists, a list of available orders and a list of orders in process. The picker should also be able to view this page.
        ii. I would like to be able to view the order details for available orders and see which orders are currently in progress.
    d. Manage Pickers:
        i. I would like to be able to go through a list of the pickers at the company with their details.
        ii. I would also like to see a list of pickers that have started their shift at the current date and time.
    e. Receiving Goods:
        i. As an admin, I want to be able to register the merchandise and the amount that the warehouse received to the inventory.
        ii. The system should automatically find empty places in the warehouse based on the amount of pallets and generate a unique pallet ID for each pallet.
        iii. Each pallet should include a pallet ID, the amount on the pallet, the location in the warehouse, and the expiry date.
    f. Manage Suppliers
        i. Add a supplier
        ii. View a list of all suppliers
    g. Manage Customers
        i. Add a customer
        ii. View a list of all customers
    h. Manage Stocks
        i. As an admin, I want to be able to search through the products in the warehouse and see how much is left of each product in order to know when to order more.

3. As a user logged in as a picker, I would like to do the following:
    a. I want to be able to start my shift by inserting the picker ID, and also end my shift using the same picker ID.
    b. Once logged in, I want to be able to see all the available and in process orders. I should have the opportunity to select the first available order (sorted by date and time of shipment). I am required to enter my picker ID and have the choice to print the picker's order details before starting the order.

    c. Once the order has been started, it is removed from the available list, and added to the processing queue.
    d. If a picker needs to leave while his/her order is being processed, the picker can press on the incomplete button and add the same picker ID that had started the order and choose the products that have already been picked. Once the status of the order becomes incomplete, the order is taken away from the processing queue, and added to the list of available orders again (at the top of the list).
    e. If a picker finishes the order, then he/she can press on the finish button that will generate a label with details of the order and an invoice for the customer which is added to the invoice history list.
4. As a user logged in as a customer, I would like to do the following:
    a. I would like to see a list of categories and pick the products I want to order from each category; and specify the amount in boxes (price is according to 1 box).
    b. Once my order is done, I want to be required to fill out my unique customer ID (so I can be identified), as well as the date and time of shipment.
    c. I would like to see a text area where I can leave a comment for the warehouse if necessary.
5. The system should automatically check the amount left in the picking positions, the system should find the pallet for that same product (using the product ID) from the storage based on the expiry date, and add it to the picking position.
6. The system will filter the orders received from customers by the order of the picking positions. This is added in the order details the picker has access to.

Release Backlog

User Story ID	Tasks	Estimated Story Points
1	As a user logged in as an admin, I want to be able to manage products.	100
2	As a user logged in as an admin, I want to have access to an invoice history.	55
3	As a user logged in as an admin, I want to be able to check the orders.	40
4	As a user logged in as an admin, I want to manage pickers.	60
5	As a user logged in as an admin, I want to be able to register merchandise received by the warehouse.	55
6	As a user logged in as an admin, I want to manage the suppliers.	40
7	As a user logged in as an admin, I want to manage the customers.	40
8	As a user logged in as an admin, I want to be able to manage the stocks in storage.	40
9	As a user logged in as a picker, I want to be able to start and end my shift.	25
10	As a user logged in as a picker, I want to be able to start an order.	60
11	As a user logged in as a picker, I should be able to stop at any point during the order and call it incomplete, and I should be able to finish an order.	90
12	As a user logged in as a customer, I would like to see a list of different categories and pick the products I want from each category.	40
13	As a user logged in as a customer, I want to be able to specify a certain	30

	amount for each product I am ordering, provide the date of shipment, and provide a comment if necessary.	
14	As a user, we want to be able to login to the system based on the different people who are going to use the system.	20
15	As developers, we are going to create a use case diagram along with use case descriptions for our system.	8
16	As developers, we are going to create activity diagrams for each use case description.	8
17	As developers, we are going to create one sequence diagram based on one use case description in which we find is most interesting.	3
18	As developers, we are going to document each step we do. We will have documentation for each design pattern we implement.	8
19	As developers, we are going to document the client/server system.	5
20	As developers, we are going to document SCRUM.	10
21	As developers, we are going to have documentation for Unified Process.	3
22	As developers, we are going to create a project report.	50
23	As developers, we are going to create a process report.	25

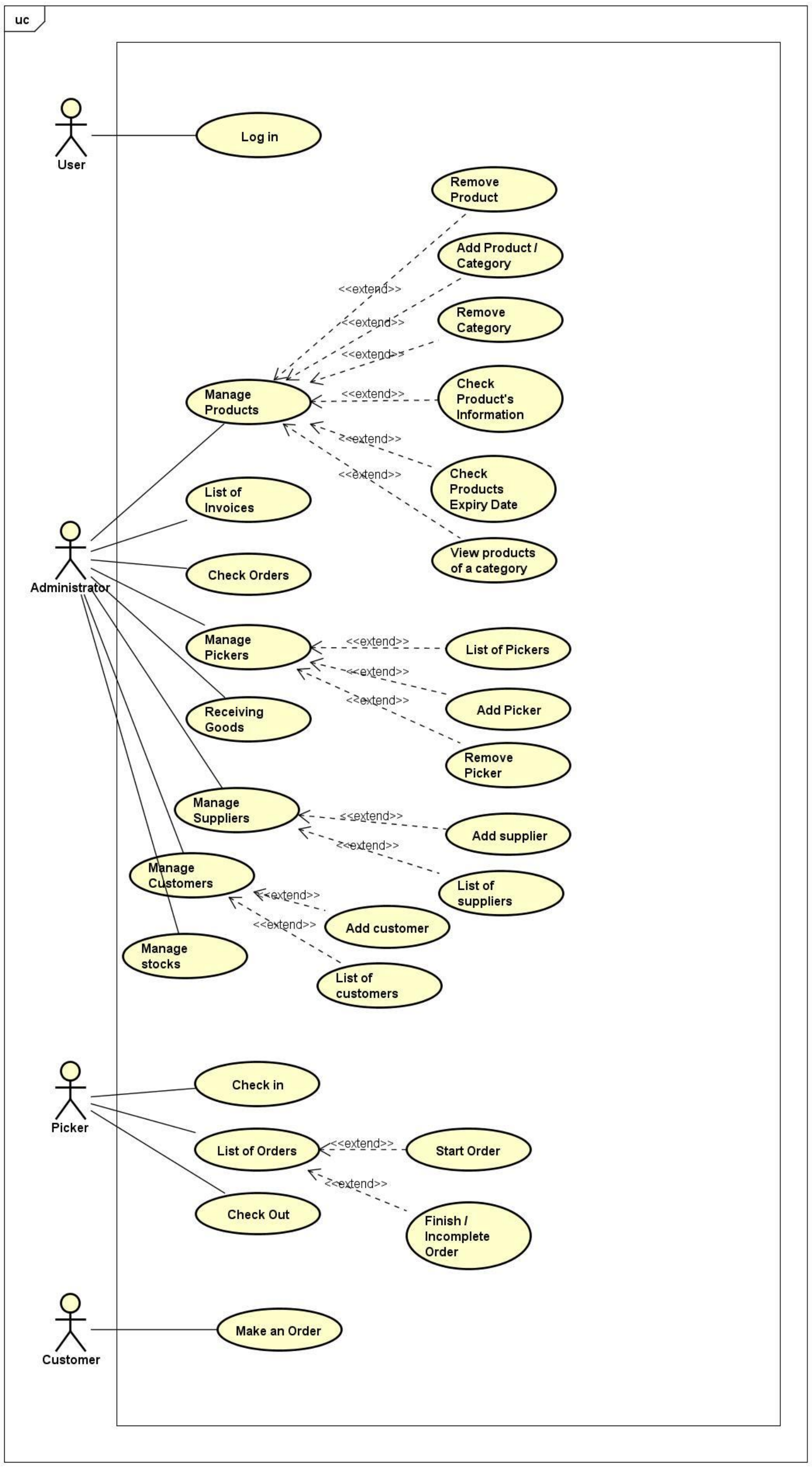
Non-Functional Requirements

1. System is a Java application which uses PostgreSql as a database system.
2. System should have an interface that is easily used by everyone, even non-developers.
3. The system should be updated at all times.

Use Case Modelling

As part of the analysis process, some diagrams are drawn to help visualize our system in a non-programming sense. The first type of diagram presented is the use case diagram. It consists of the actors of the system along with functionalities that belong to each actor. This diagram is shown below.

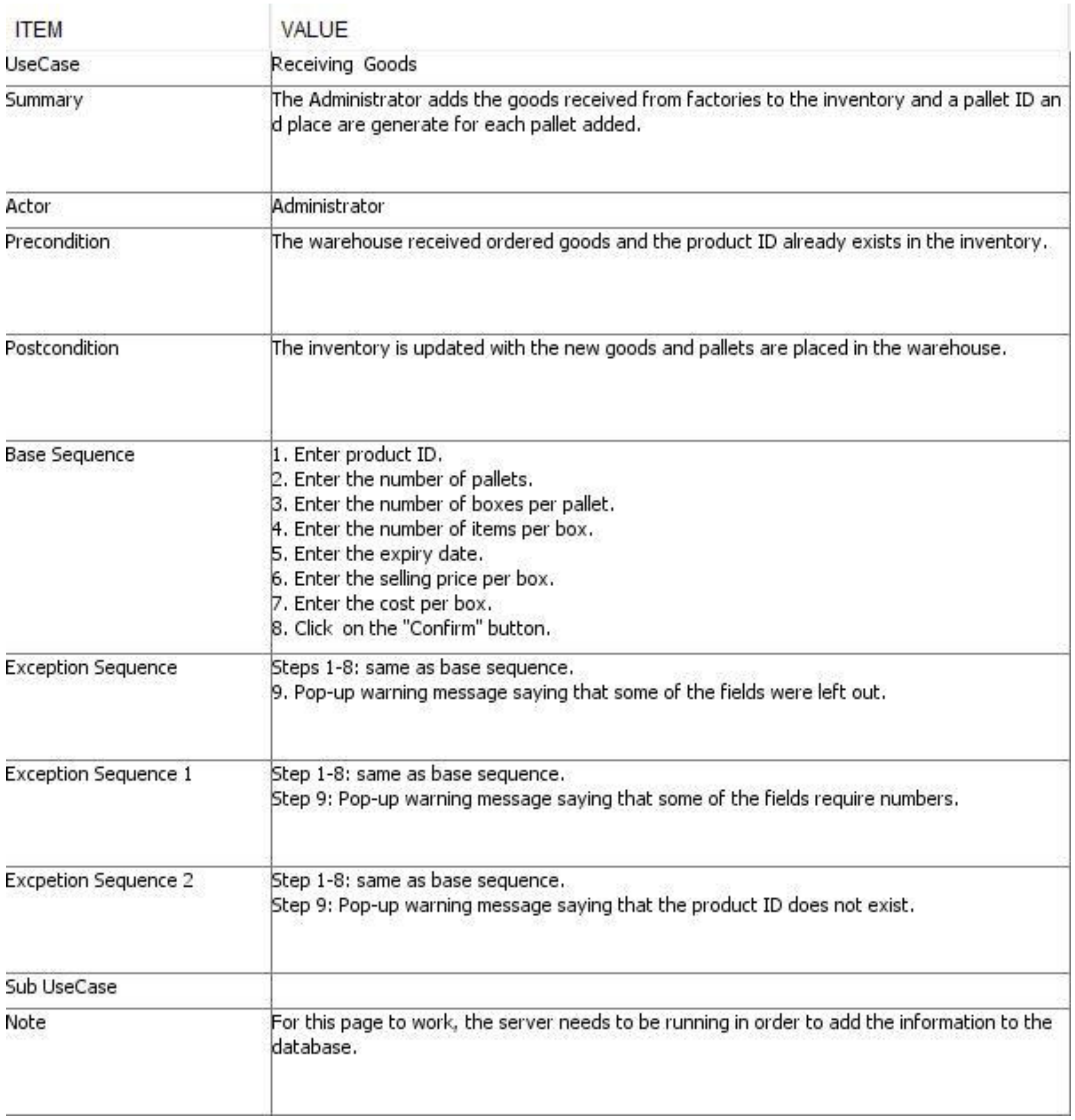
Use Case Diagram



There are 3 main actors of our system, the administrator, the picker, and the customer. The fourth actor called user, is only there to show that there is a log in page that decides which actor the user is, based on a predefined username and password. All the lines from the actor pointing towards bubbles, are the main functionalities of that specific actor. There are also second lines pointing to another bubble with the word ‘extends’ above it. That means that through that main functionality, the actor has the choice of more detailed functionality.

Use Case Description

A use case description is a more detailed version of a use case diagram which takes every functionality and explains it even further; the functionality is broken down into steps and exceptions if necessary. An example of one the use case descriptions can be found below, and the reason that one was chosen was because we believe it was the most interesting one.



ITEM	VALUE
UseCase	Receiving Goods
Summary	The Administrator adds the goods received from factories to the inventory and a pallet ID an d place are generate for each pallet added.
Actor	Administrator
Precondition	The warehouse received ordered goods and the product ID already exists in the inventory.
Postcondition	The inventory is updated with the new goods and pallets are placed in the warehouse.
Base Sequence	1. Enter product ID. 2. Enter the number of pallets. 3. Enter the number of boxes per pallet. 4. Enter the number of items per box. 5. Enter the expiry date. 6. Enter the selling price per box. 7. Enter the cost per box. 8. Click on the "Confirm" button.
Exception Sequence	Steps 1-8: same as base sequence. 9. Pop-up warning message saying that some of the fields were left out.
Exception Sequence 1	Step 1-8: same as base sequence. Step 9: Pop-up warning message saying that some of the fields require numbers.
Excpetion Sequence 2	Step 1-8: same as base sequence. Step 9: Pop-up warning message saying that the product ID does not exist.
Sub UseCase	
Note	For this page to work, the server needs to be running in order to add the information to the database.

A use case description consists of all of the above. The base sequence shows that if everything goes according to the plan, those are the steps that should be followed. There are 3 exception sequences in this case which handles all the invalid input or if anything went wrong from the base sequence.

We documented each use case description but only included one in this report. The rest of the use case descriptions can be found in Appendix B – Use Case Descriptions which is a pdf file within the same folder as this report.

Actor Description

We have three main actors of our system which can be found through the use case diagram;

there is an administrator, a picker, and a customer. There is a fourth user in the use case diagram called user, which is only there to indicate that each user needs to login in order to access the administrator page, the picker page, and the customer page. All the users contribute in making the inventory a well-structured one, an inventory that deals with orders being made, keeping track of the products moving in and out of the warehouse and keeping track of invoices.

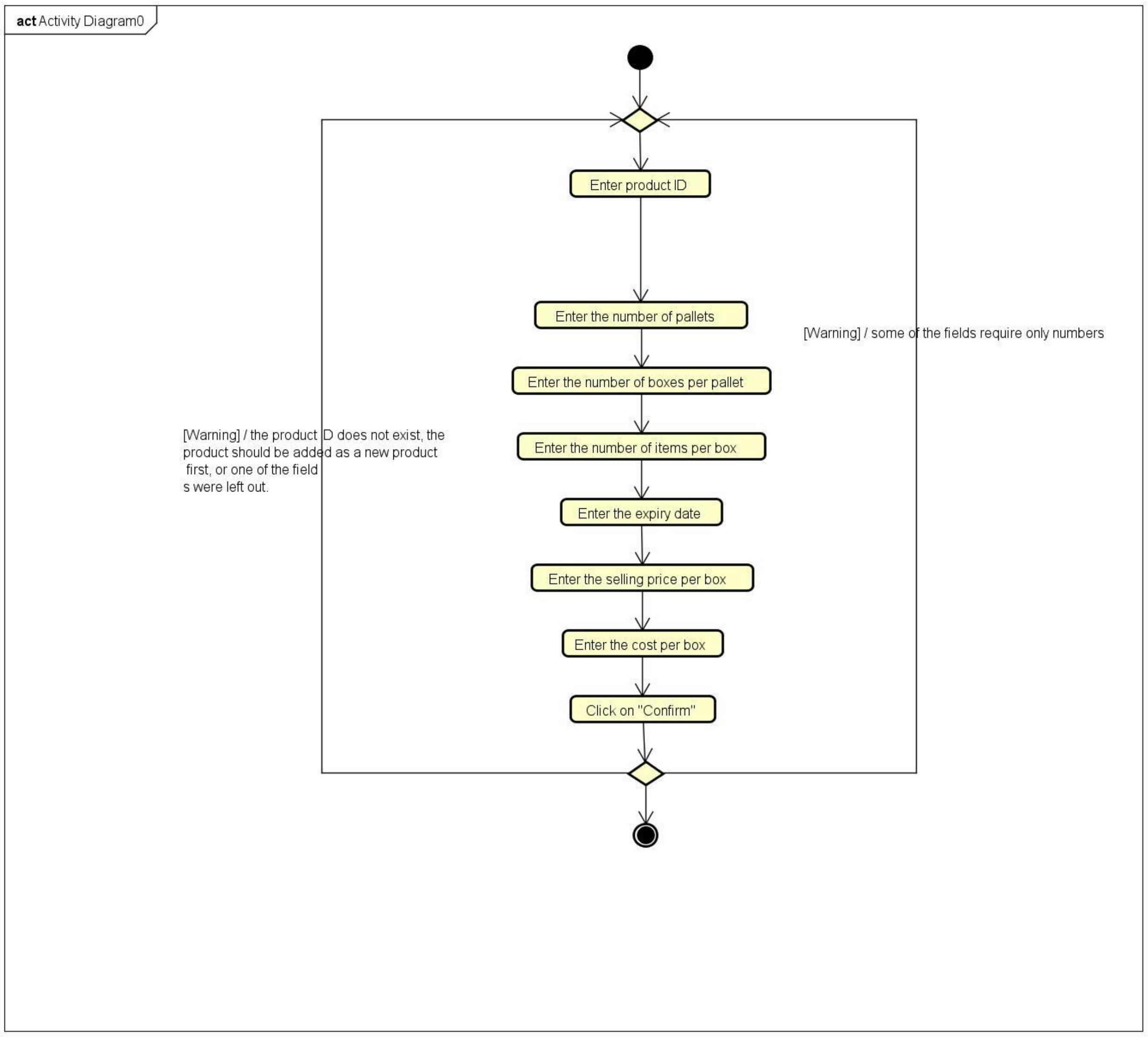
All users depend on one another in order for the warehouse management system to function appropriately. The administrator has more functionality than the other users as they have to manage the other users along with all the organization of the inventory. The administrator manages products, manages pickers, manages suppliers, manages customers, manages orders, views invoices, and can update stocks. As evident, the administrator is the most vital user of the system and without him/her, there would be no suppliers and customers added to the system. But, without the customers and the pickers being different users, the administrator would have no work to do.

The second user is the picker. The picker plays a vital role in working on the orders that are sent by the customers. Without the pickers, the warehouse would just be receiving orders and not being able to send out the shipment. The picker can start an order, finish an order and also mark an order as incomplete. If a product is not available at the warehouse, the order can still be finished and an invoice will be generated without that product that was not available. The difference between incomplete and finishing an order is that the incompletion of an order is used in the case of the picker needing to leave work or such. When marked as incomplete, the order is back on the available list and an invoice is not generated unlike when finishing an order. The other functionality the picker has is checking in and out of work. We implemented it to ensure that the picker trying to start an order has been checked in and is actually at work at that specific time. We also used that functionality to ensure that the picker who started the order is the same picker who is finishing it.

The customer is the final user of our system. Without the customer page, we would not be able to receive any orders. The customer picks a category and selects all the items he/she would like from that category. The customer needs to fill out the amount they would want and add the items to the basket before moving on to another category. Once the order is filled and sent, the administrator and picker receive it automatically.

Activity Diagrams

Another type of diagram that also explains a detailed functionality of the system but uses a little more technical visuals than a use case description is an activity diagram. As shown below, we used the activity diagram that is related to the use case description that was explained in the previous sections of this paper.

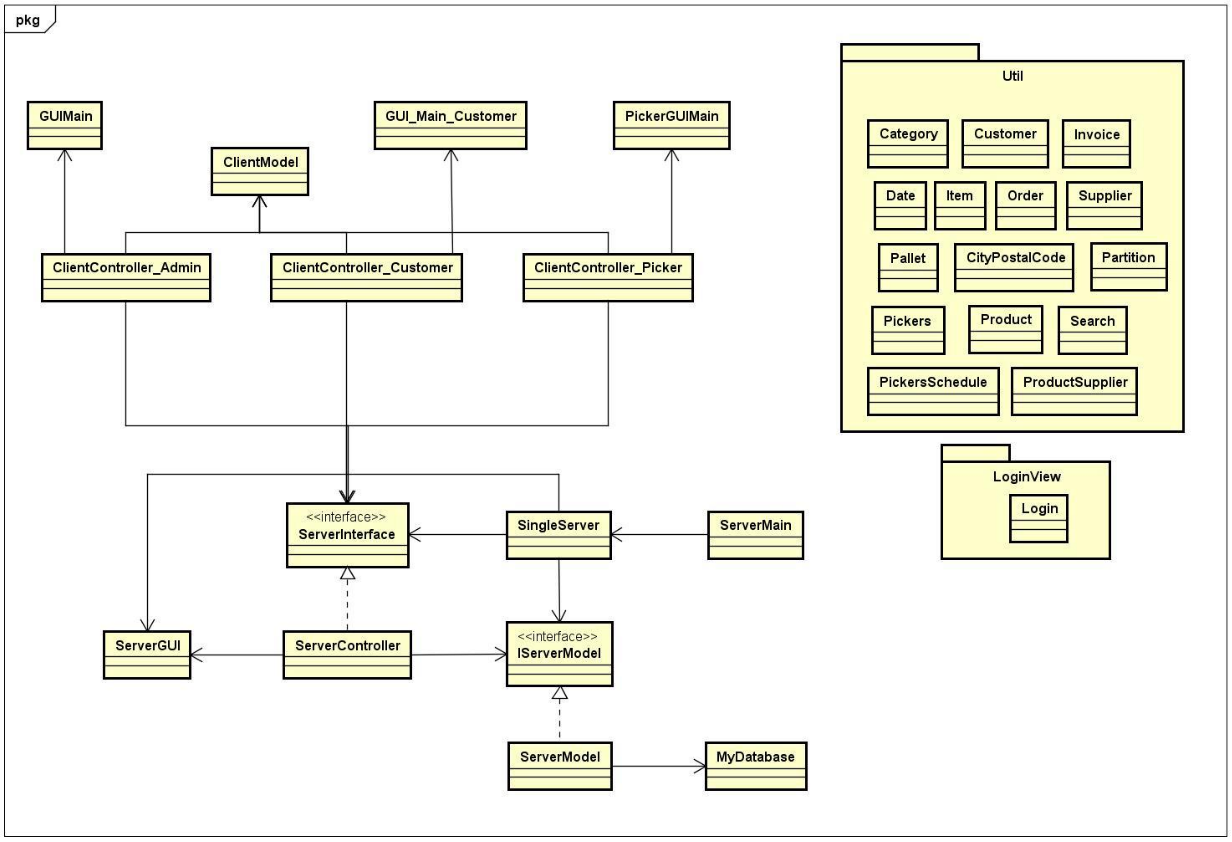

act Activity Diagram0
Enter product ID
Enter the number of pallets
[Warning] / some of the fields require only numbers
Enter the number of boxes per pallet
[Warning] / the product ID does not exist, the product should be added as a new product first, or one of the fields were left out.
Enter the number of items per box
Enter the expiry date
Enter the selling price per box
Enter the cost per box
Click on "Confirm"

We created an activity diagram for each use case description but the rest of the diagrams can be found in Appendix C – Activity Diagrams which is a separated pdf file within the same folder.

Design

Analysis is usually done before all the programming begins; but design on the other hand, is the way the system has been designed and structured. The one very obvious way to document the structure of the system is using a class diagram which can be found below.

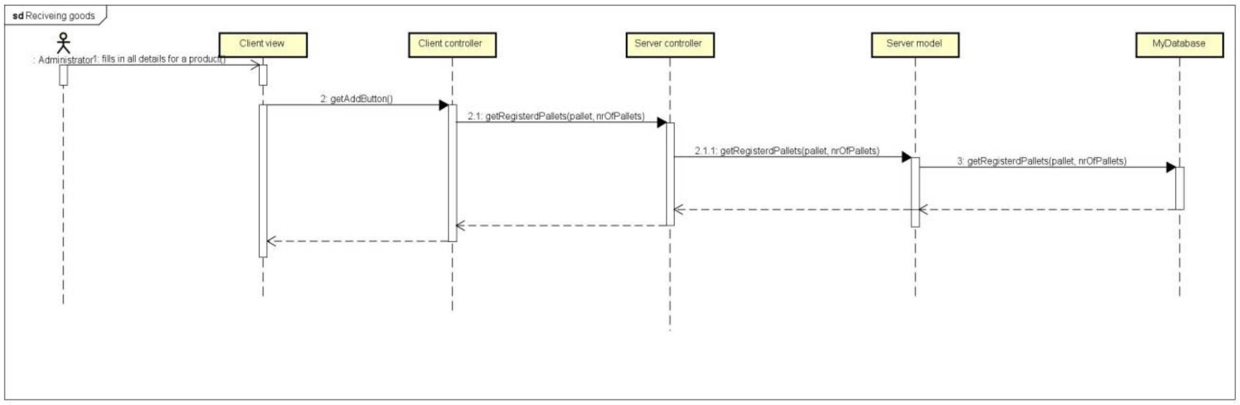
Class Diagram



The system created consisted of multiple packages with at least a few classes in each package. In order to visualize our system, we decided to include the most important connections of the classes and then convey the rest of the classes that are somehow used as utility classes in a package on their own, as shown to the right of the diagram. The system follows a model view controller pattern and that is evident from the class names. There is a server controller which is the main connection between the server view and the server model. The client controller each talk to the server interface which is implemented by the server controller and thus holds all the methods that the clients would need. The client model does not do much because the client requests for a specific action to be done and the server does not send it the entire database, rather, it sends it exactly what the client is asking for, in order to lower the traffic on the server. Each client controller has its own GUI implemented.

Sequence Diagram

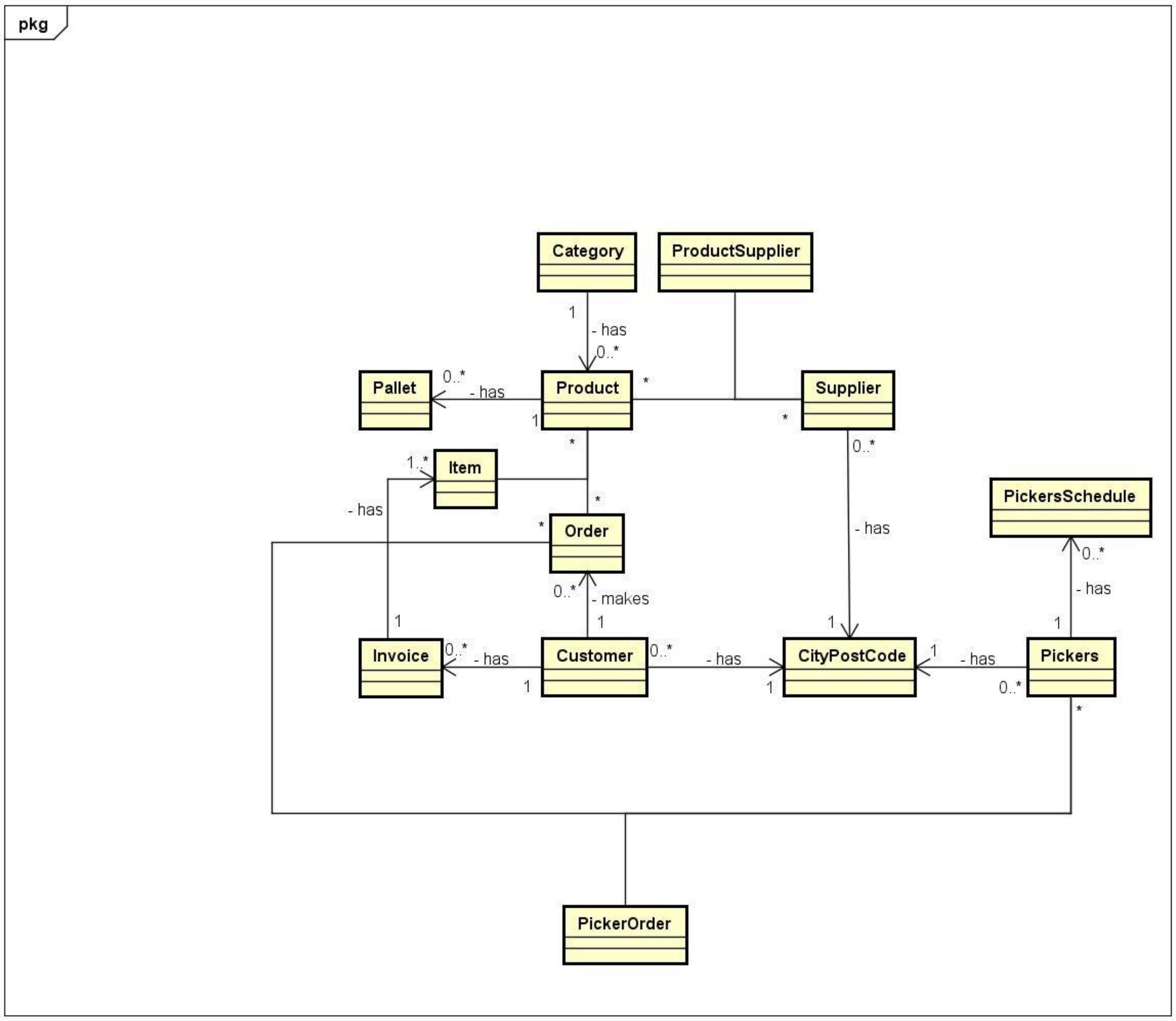
A sequence diagram is used to explain the process of how a certain functionality takes place. Again, we decided to use the same functionality as we showed in the use case description and the activity diagram for consistency.


sd Reciveing goods
: Administrator
Client view
Client controller
Server controller
Server model
MyDatabase
1: fills in all details for a product()
2: getAddButton()
2.1: getRegisterdPallets(pallet, nrOfPallets)
2.1.1: getRegisterdPallets(pallet, nrOfPallets)
3: getRegisterdPallets(pallet, nrOfPallets)

The sequence diagram above shows who the actor is. The yellow boxes above convey the classes that the methods are being called from, starting with the view all the way to the server model and the database. This process is to register incoming goods to the inventory and each line has a statement above it which is exactly what the method in that specific class is called. The dotted arrows are return messages to show that once the database has been modified, it returns something back all the way to the view which the administrator can now see.

EER Diagram

An EER diagram does not regard any Java related design, but the way our database is designed with all the tables. Each entity of the database is in a box on its own with the name of the entity printed above. An EER diagram shows the relationship between the entities, in terms of which way the arrow is pointing, and also the multiplicity between them. For example, a category has products according to the arrow, and according to the cardinality, one category can have 0 to many products, and only one specific product can belong to one category. When it comes to relationships that are many to many, a third attribute needs to be created and that is evident in three places, the relationship between the product and order, the product and supplier, and also the relationship between the pickers and order. Both of these relationships are many to many, therefore a third attribute had to be created such as Item for the product and order entities which includes the product ID as a primary key from the product table and the order number as a primary key from the order table. In the next section, there is a logical data model of the EER diagram which shows all the attributes of each entity and what primary keys and foreign keys exist in each one.



Logical Data Model

Category (cname,nrofproducts)

- Primary key (cname)

Supplier (supname, phone, email, street, postcode)

- Primary key (supname)
- Foreign key (postcode) references **Citypostcode**(postcode)

Citypostcode (postcode,city)

- Primary key (postcode)

Customer (id, name, street, postcode, phone, email)

- Primary key (id)
- Foreign key (postcode)

Invoice (invoiceno, customerid, dateofshipment, dateoforderreceived, totalexvat, vat, totalinclvat)

- Primary key (invoiceno)

- Foreign key (customerid) references **Customer**(id)

Item (productid, amount, ordernumber, picked, short)
- Primary key (productid, ordernumber)
- Foreign key (productid) references **Product** (id)

Order (ordernumber, dateofshipment, dateorderreceived, customerid,availability)
- Primary key (ordernumber)
- Foreign key (customerid) references **Customer**(id)

Pallet (id, productid, nrofboxesperpallet, itemsperbox, expirydate, cost, sellingprice, aisle, depth, floor)
- Primary key (id)
- Foreign key (productid) references **Product** (id)

Product (id, pname, cname)
- Primary key (id)
- Foreign key (cname) references **Category**(cname)

Productsupplier (pid, supname)
- Primary key(pid,supname)
- Foreign key(supname) references **Supplier**(supname)

Picker (id, cpr, name, street, postcode, phone,email)
- Primary key(id)
- Constraint unique(cpr)
- Foreign key(postcode) references **Citypostcode**(postcode)

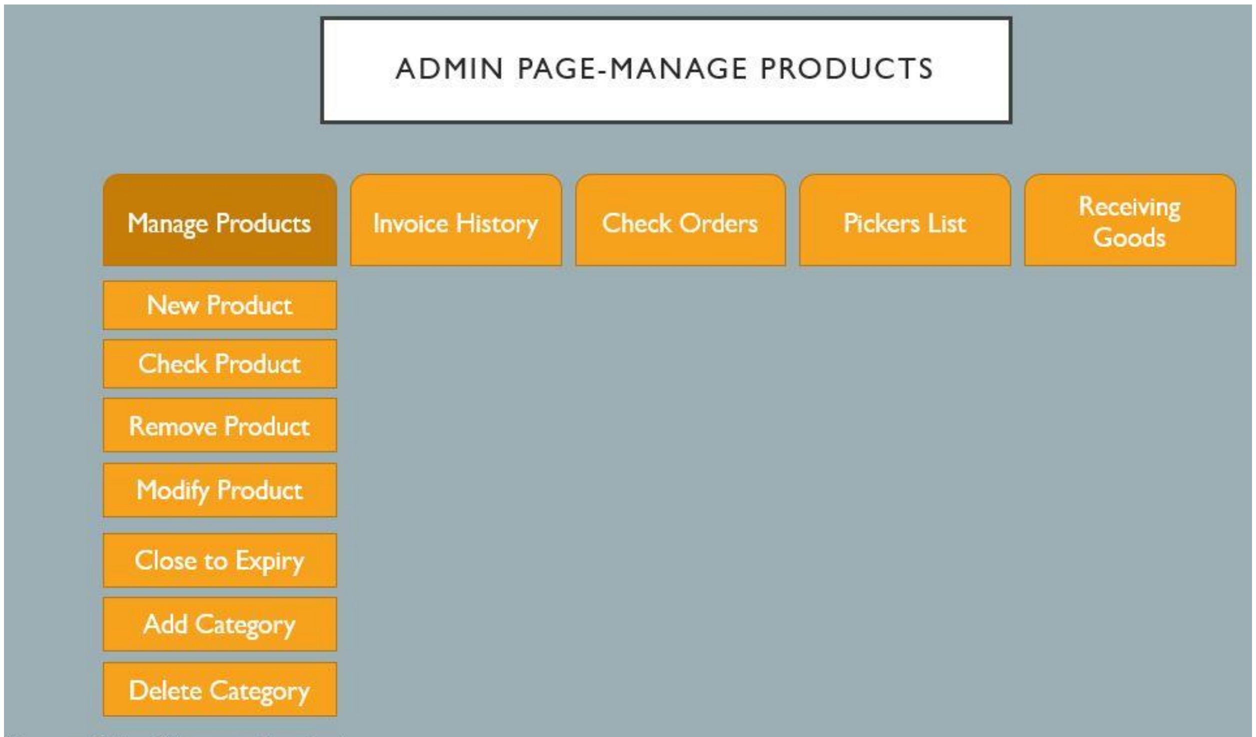
Pickersschedule (id, checkin, checkout)
- Primary key (id,checkin)
- Foreign key id references **Pickers** (id)

Pickerorder (ordernumber, pickerid)
- Primary key (ordernumber, pickerid)
- Foreign key pickerid references **Pickers** (id)

GUI Design

GUI stands from Graphical User Interface which allows the user to interact with the system. In the beginning, we tried to use a GUI builder that is built into Eclipse but it proved to be more complicated to work with the auto-generated code.

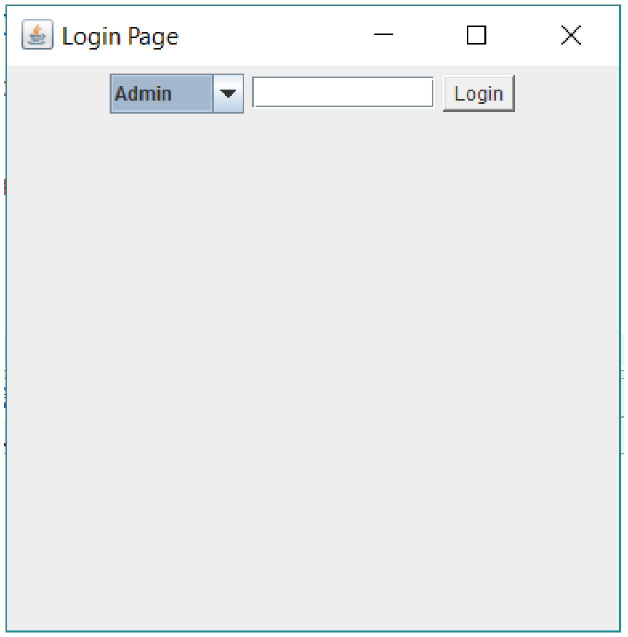
To begin with, we created a sketch for each page we thought we would implement on pen and paper and then implemented the drawings in PowerPoint so they can more visual. The process of drawing each page was extremely helpful because we realized that as we did that, we discussed a lot about the functionality of our system. The whole process took us a while because we were discussing about the functionality as we agreed on what the pages should generally look like. Below is just an example of what one of our very first drawings looked like:



Menu of the Manage Products

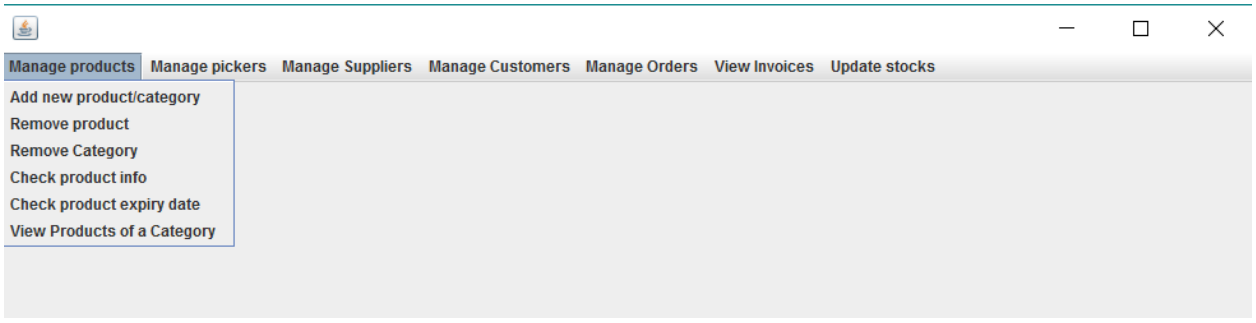
Later in this section is a screenshot of our Administrator page and the difference is definitely obvious but it is also very similar to our first few drafts of sketches.

In order to limit the functionality for each user, three different interfaces were implemented, one for each type of user. The difference between the users is signified by the log in page which is the same interface for all users. What creates the difference in the interfaces later on is the password that is used according to the username that was chosen.



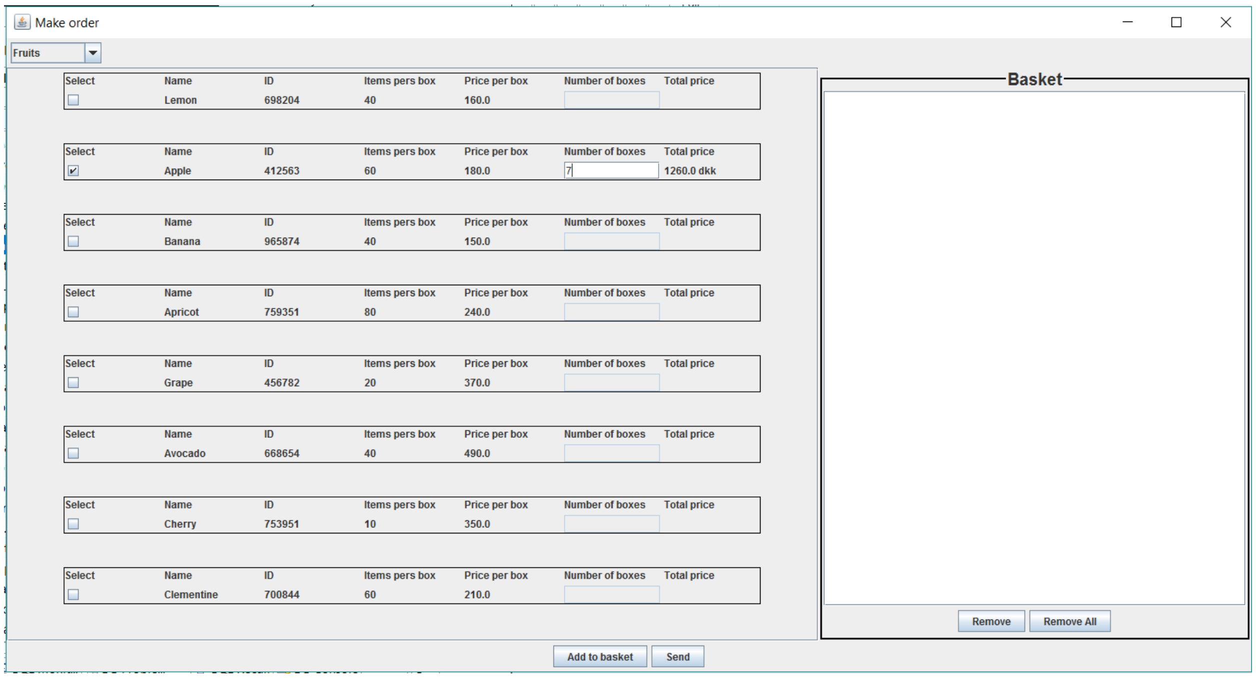
The screenshot above presents the log in page which starts different GUI's based on the user type.

The GUI for the administrator side has more options than any other type of user. The options are listed as menu items, each menu item opens a new panel. Under each menu item there are sub menu items, which organizes the way the data is conveyed.



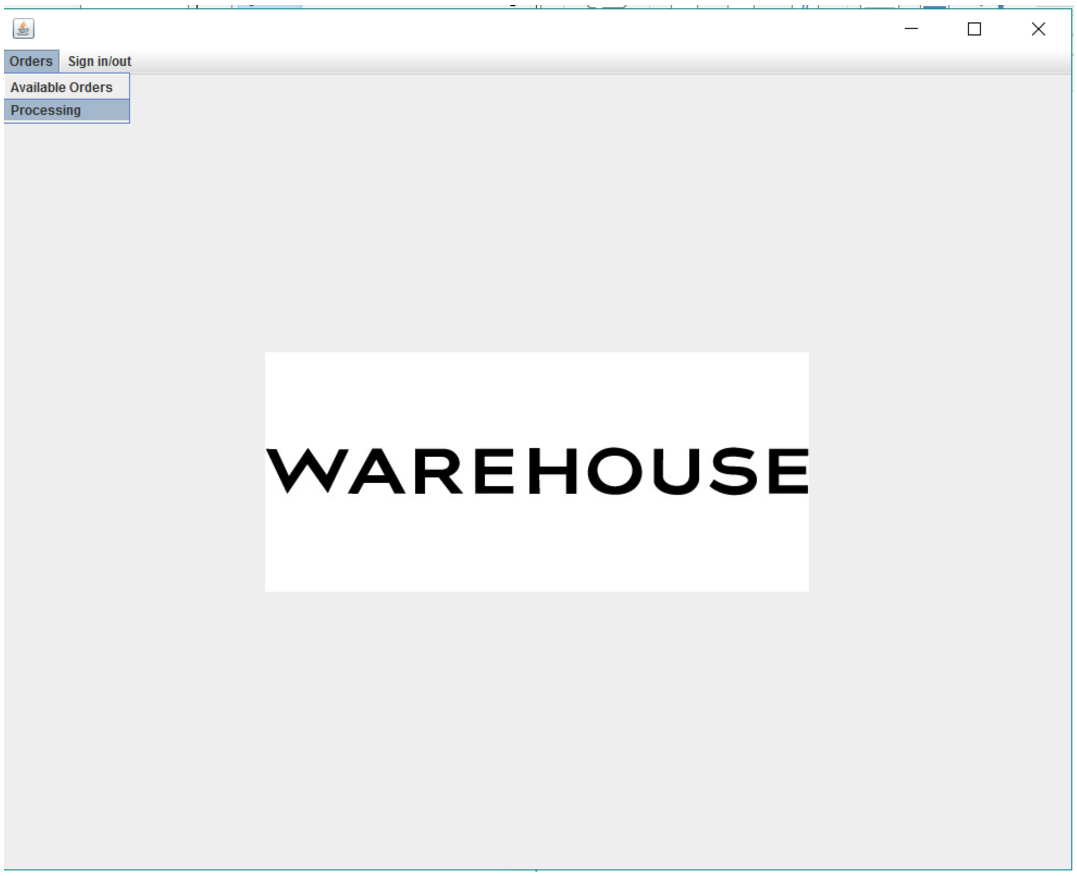
The screenshot above presents the way the menu bar is organized for the administrator.

Once connected as a customer there is only one option given, there is no menu bar and the GUI starts directly. This view was created to be easier for the customer to see what they have selected and what is added to the order. The figure below shows the view and how it is split into two, on the left is the list of all the times in a certain category and to the right is the basket where all the items are added by the customer.

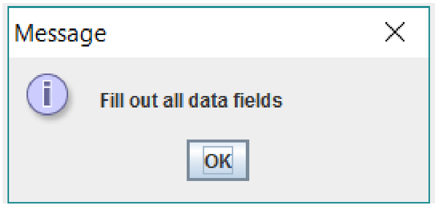


Select	Name	ID	Items pers box	Price per box	Number of boxes	Total price
☐	Lemon	698204	40	160.0		
✓	Apple	412563	60	180.0	7	1260.0 dkk
☐	Banana	965874	40	150.0		
☐	Apricot	759351	80	240.0		
☐	Grape	456782	20	370.0		
☐	Avocado	668654	40	490.0		
☐	Cherry	753951	10	350.0		
☐	Clementine	700844	60	210.0		

The picker side is handling two main features, checking in or out of the system, and getting a list of orders and starting the first one. After starting an order the picker has the possibility to choose from two buttons with different functionality. The order can be called incomplete or finished based on the situation.



The GUI was created in the way that if there is invalid information inputted by the user there will always be pop up warning messages saying what has gone wrong.

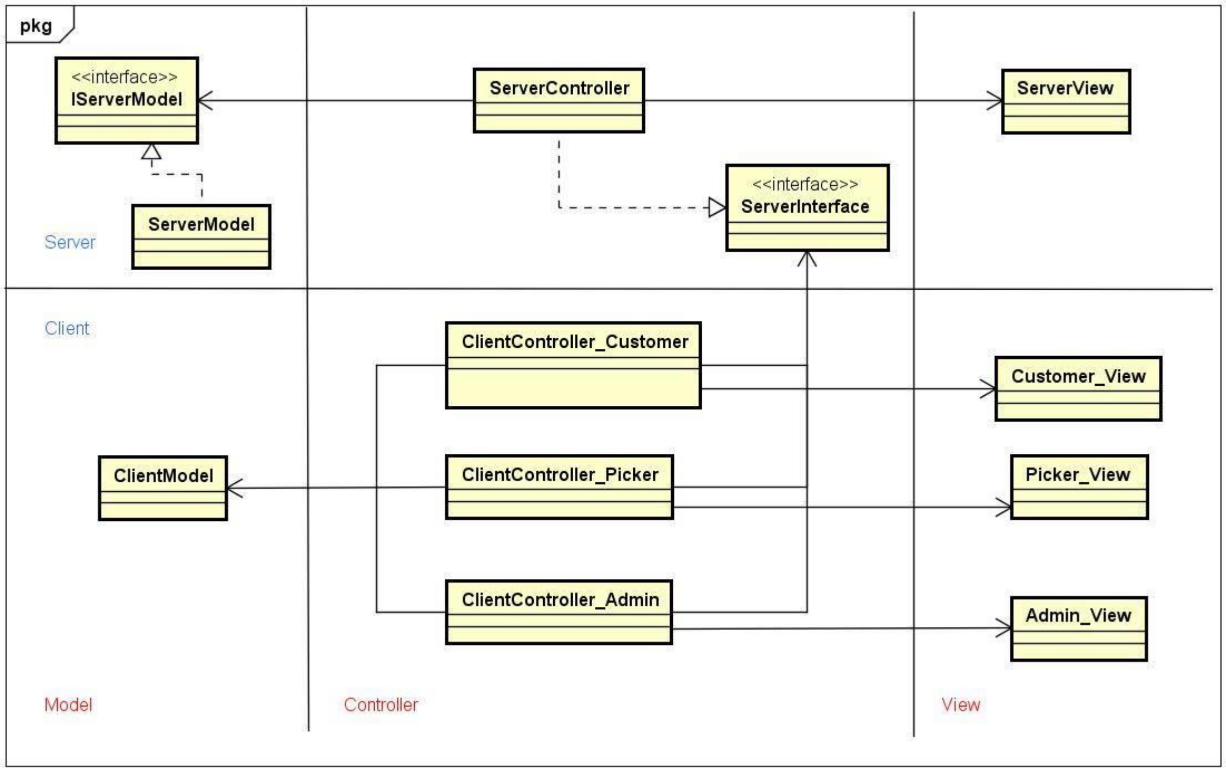


Fill out all data fields

Design Patterns

Model View Controller

UML Class Diagram



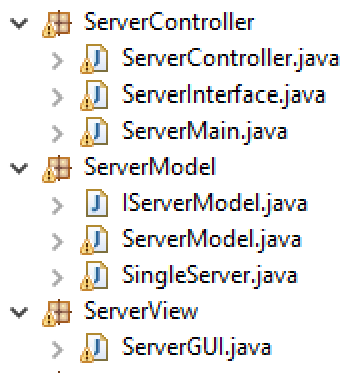
Intent

The purpose of having an MVC (Model View Controller) pattern is to organize the classes in a way to help the developer better understand the functionalities behind the classes. For example, the model should be dealing with the data, whether it is regarding a database or a file system. The controller should have the actual functionality with implemented methods that would be used in the system. The controller has access to the view and model which are used to access the methods implemented in those two classes/packages. The view should only deal with the interface of the system, whether it is a graphical user interface or just text in the console. This way of splitting the code makes it easier for further improvements or changes to the code.

Structure

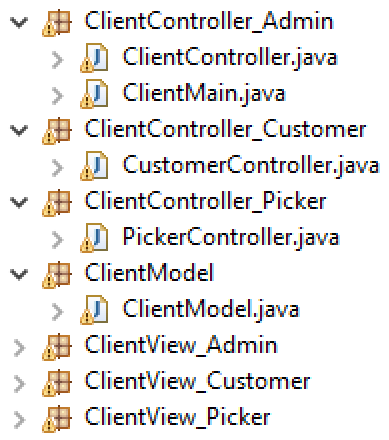
In our assignment, we have an MVC implemented on the server side and an MVC implemented on the client side. The client in our system can be one of the following: Administrator, Customer, or Picker; therefore, we have three different views for every client type and three different client controllers. The three types of client share the same model.

For the server side, we have three different packages one for ServerModel, one for ServerController, and one with ServerView. In our ServerModel package, we have three classes IServerModel, ServerModel, and SingleServer. The ServerModel uses Mydatabase class to communicate with the server. The ServerModel implements the interface IServerModel. The purpose of the interface is to allow the adapter design pattern do its job in case changing the database to be files. SingleServer is used in order to implement the Singleton Pattern and start up the server.



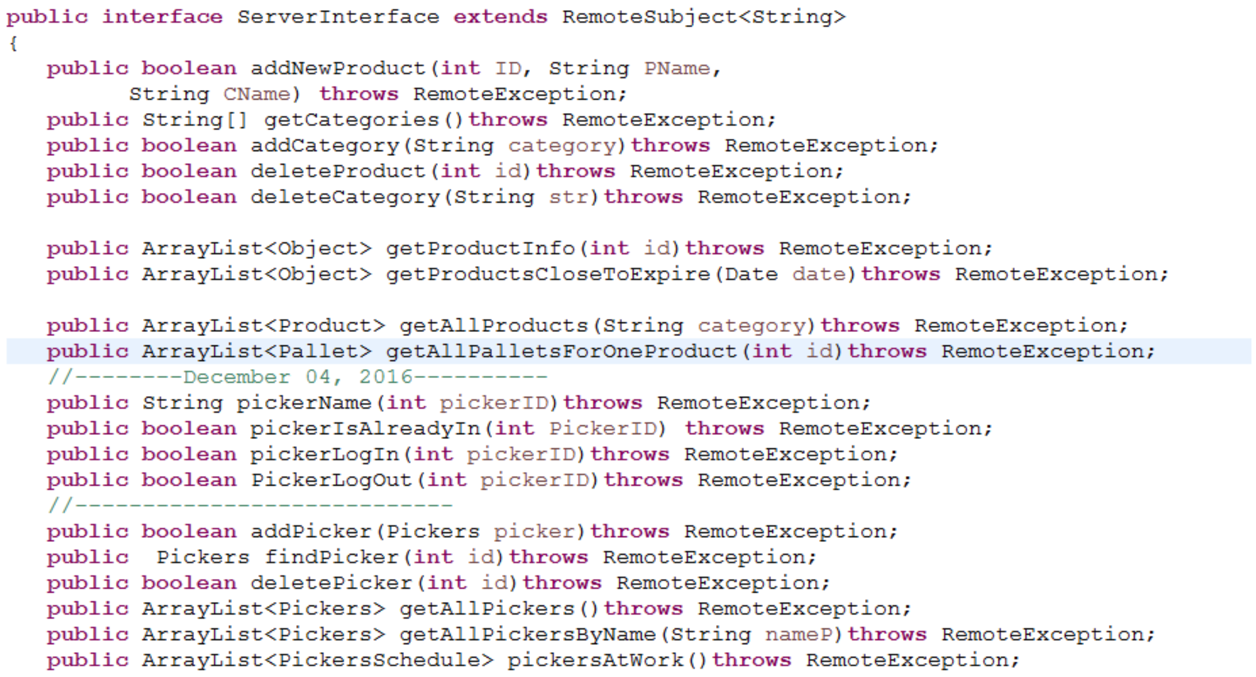
```
ServerController
    ServerController.java
    ServerInterface.java
    ServerMain.java
ServerModel
    IServerModel.java
    ServerModel.java
    SingleServer.java
ServerView
    ServerGUI.java
```

The ServerController package consists of the ServerInterface, ServerController and ServerMain classes. The ServerInterface includes all the methods that the server controller should be implementing. ServerController implements the ServerInterface and thus includes all the methods declared in the ServerInterface class. The ServerController has access to the ServerModel and ServerView which is how it implements the methods it needs to. The ServerView package consists of only one class which is the ServerGUI. The ServerGUI shows a message that the server is successfully running.



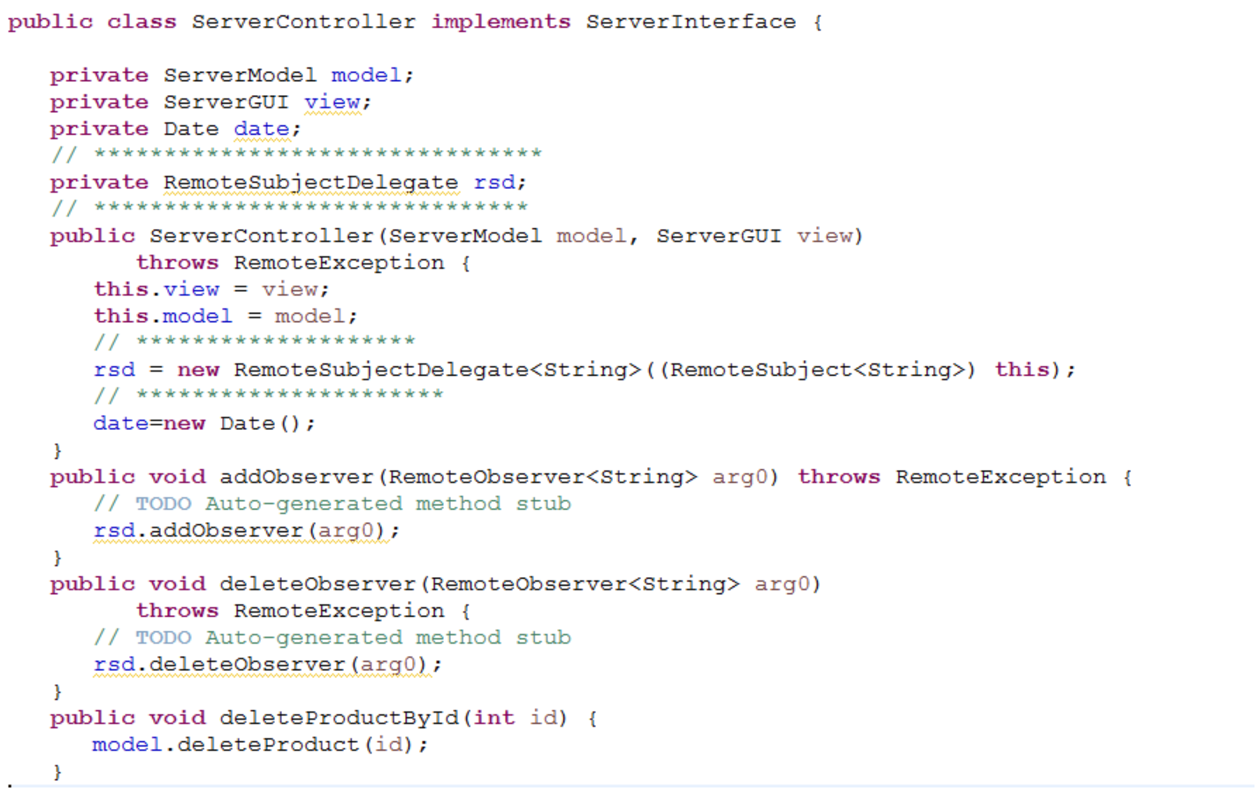
```
ClientController_Admin
    ClientController.java
    ClientMain.java
ClientController_Customer
    CustomerController.java
ClientController_Picker
    PickerController.java
ClientModel
    ClientModel.java
ClientView_Admin
ClientView_Customer
ClientView_Picker
```

The MVC on the client side consist of 7 packages. Client Model which has ClientModel class and this class is shared for every type of user. Since we have Administrator, Customer and picker, we made three different packages of view: ClientView_Admin, ClientView_Customer, ClientView_Picker and every package has some classes for GUI. For every type of client there is a controller which will connect the model with the view and communicate with the server controller through the instance of the server interface. Upon starting of the program there is a login page that will appear. Then, an appropriate instance of the model and view are created and passed to the creation of the controller based on the name and password the user provided.



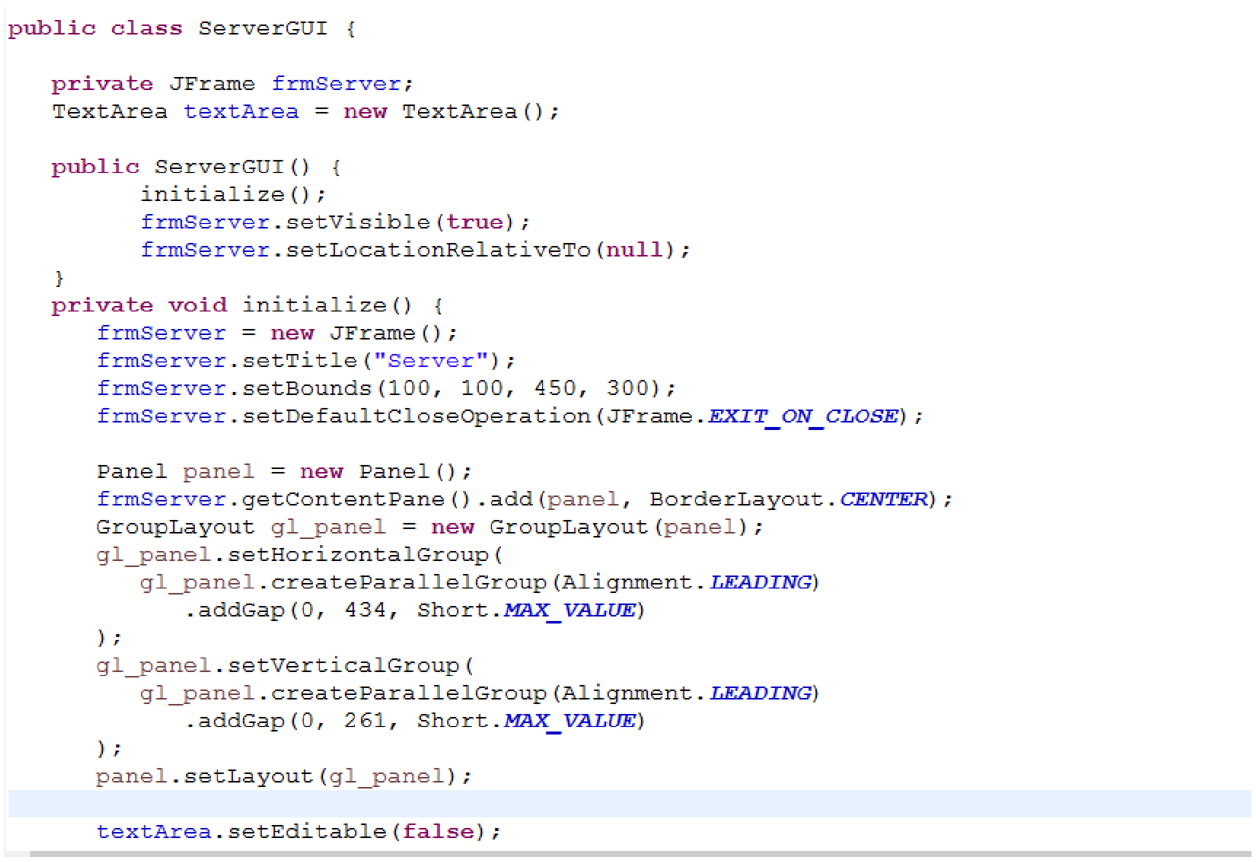
```
public interface ServerInterface extends RemoteSubject<String>
{
   public boolean addNewProduct(int ID, String PName,
         String CName) throws RemoteException;
   public String[] getCategories()throws RemoteException;
   public boolean addCategory(String category)throws RemoteException;
   public boolean deleteProduct(int id)throws RemoteException;
   public boolean deleteCategory(String str)throws RemoteException;

   public ArrayList<Object> getProductInfo(int id)throws RemoteException;
   public ArrayList<Object> getProductsCloseToExpire(Date date)throws RemoteException;

   public ArrayList<Product> getAllProducts(String category)throws RemoteException;
   public ArrayList<Pallet> getAllPalletsForOneProduct(int id)throws RemoteException;
   //--------December 04, 2016----------
   public String pickerName(int pickerID)throws RemoteException;
   public boolean pickerIsAlreadyIn(int PickerID) throws RemoteException;
   public boolean pickerLogIn(int pickerID)throws RemoteException;
   public boolean PickerLogOut(int pickerID)throws RemoteException;
   //----------------------------
   public boolean addPicker(Pickers picker)throws RemoteException;
   public  Pickers findPicker(int id)throws RemoteException;
   public boolean deletePicker(int id)throws RemoteException;
   public ArrayList<Pickers> getAllPickers()throws RemoteException;
   public ArrayList<Pickers> getAllPickersByName(String nameP)throws RemoteException;
   public ArrayList<PickersSchedule> pickersAtWork()throws RemoteException;
```

Code snippet from the server interface. It has the signature of some methods to be implemented in the server controller. It represents the functionality of the server.



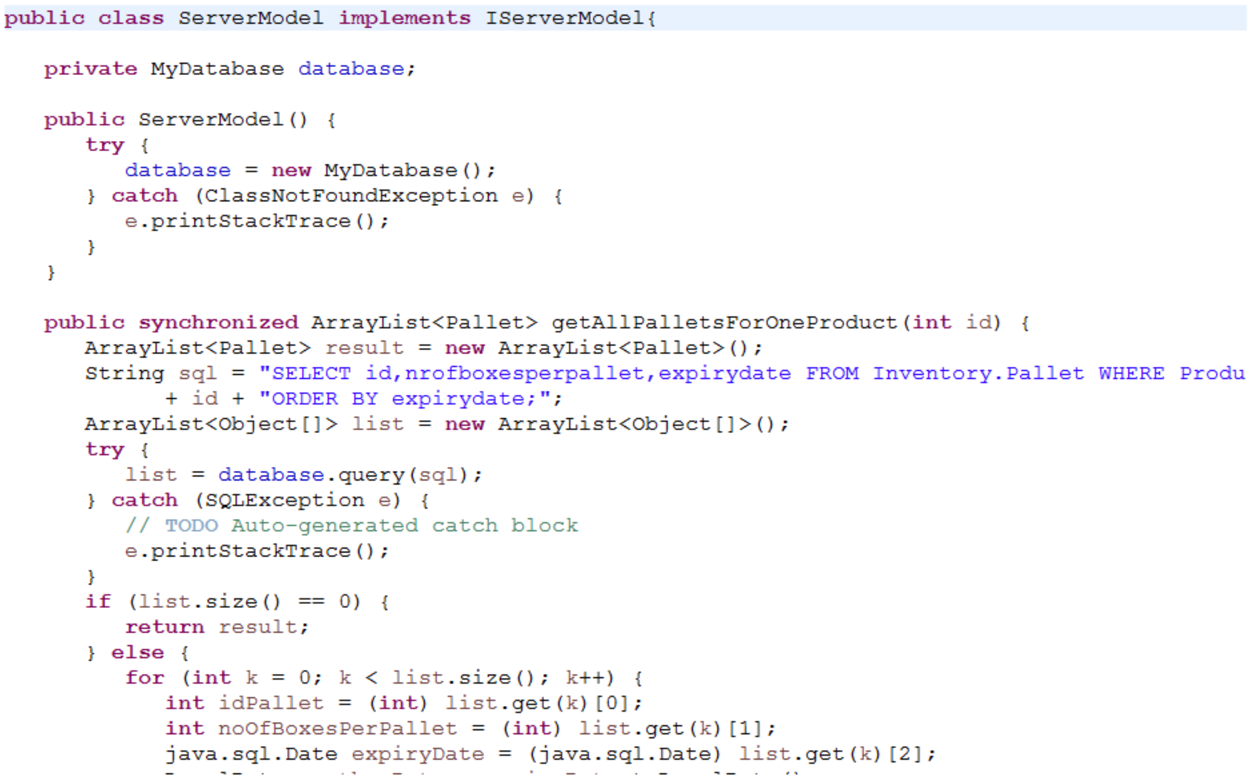
```java
public class ServerController implements ServerInterface {

   private ServerModel model;
   private ServerGUI view;
   private Date date;
   // ********************************
   private RemoteSubjectDelegate rsd;
   // *******************************
   public ServerController(ServerModel model, ServerGUI view)
         throws RemoteException {
      this.view = view;
      this.model = model;
      // ********************
      rsd = new RemoteSubjectDelegate<String>((RemoteSubject<String>) this);
      // **********************
      date=new Date();
   }
   public void addObserver(RemoteObserver<String> arg0) throws RemoteException {
      // TODO Auto-generated method stub
      rsd.addObserver(arg0);
   }
   public void deleteObserver(RemoteObserver<String> arg0)
         throws RemoteException {
      // TODO Auto-generated method stub
      rsd.deleteObserver(arg0);
   }
   public void deleteProductById(int id) {
      model.deleteProduct(id);
   }
```

The code above is from the server controller implementing the server interface and its methods.



```java
public class ServerGUI {

   private JFrame frmServer;
   TextArea textArea = new TextArea();

   public ServerGUI() {
         initialize();
         frmServer.setVisible(true);
         frmServer.setLocationRelativeTo(null);
   }
   private void initialize() {
      frmServer = new JFrame();
      frmServer.setTitle("Server");
      frmServer.setBounds(100, 100, 450, 300);
      frmServer.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      Panel panel = new Panel();
      frmServer.getContentPane().add(panel, BorderLayout.CENTER);
      GroupLayout gl_panel = new GroupLayout(panel);
      gl_panel.setHorizontalGroup(
         gl_panel.createParallelGroup(Alignment.LEADING)
            .addGap(0, 434, Short.MAX_VALUE)
      );
      gl_panel.setVerticalGroup(
         gl_panel.createParallelGroup(Alignment.LEADING)
            .addGap(0, 261, Short.MAX_VALUE)
      );
      panel.setLayout(gl_panel);

      textArea.setEditable(false);
```

Above is code snippet from the simple GUI of the server to show its status.

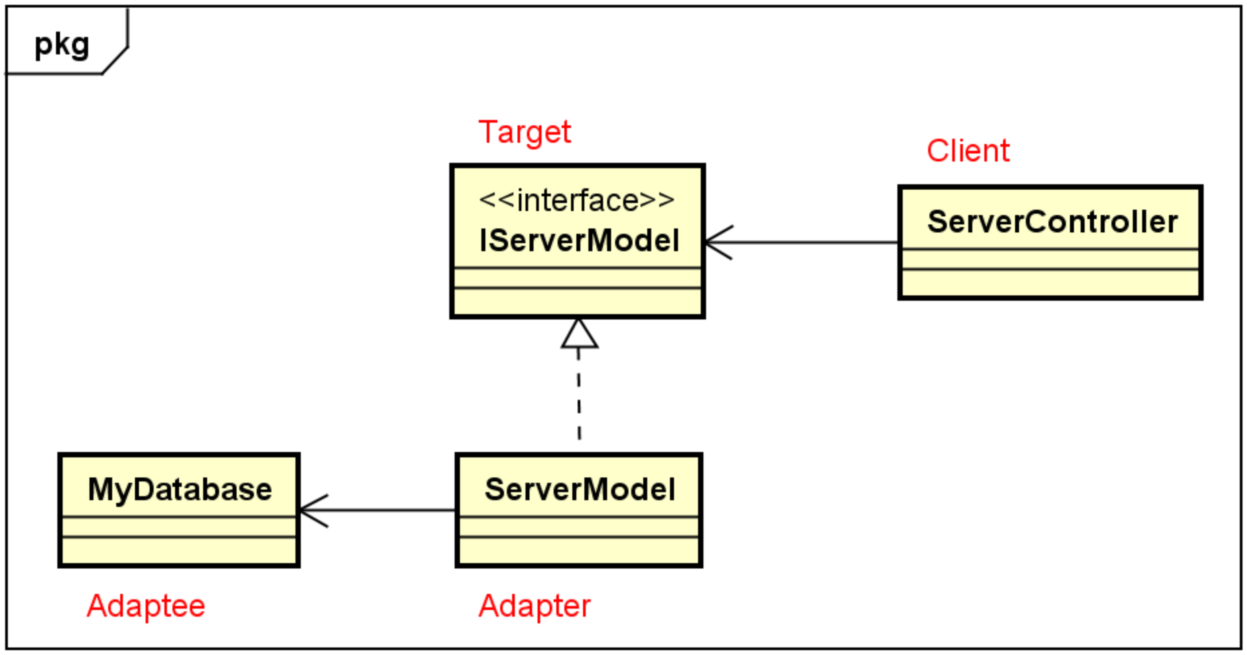


```
public class ServerModel implements IServerModel{

   private MyDatabase database;

   public ServerModel() {
      try {
         database = new MyDatabase();
      } catch (ClassNotFoundException e) {
         e.printStackTrace();
      }
   }

   public synchronized ArrayList<Pallet> getAllPalletsForOneProduct(int id) {
      ArrayList<Pallet> result = new ArrayList<Pallet>();
      String sql = "SELECT id,nrofboxesperpallet,expirydate FROM Inventory.Pallet WHERE Produ
            + id + "ORDER BY expirydate;";
      ArrayList<Object[]> list = new ArrayList<Object[]>();
      try {
         list = database.query(sql);
      } catch (SQLException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
      if (list.size() == 0) {
         return result;
      } else {
         for (int k = 0; k < list.size(); k++) {
            int idPallet = (int) list.get(k)[0];
            int noOfBoxesPerPallet = (int) list.get(k)[1];
            java.sql.Date expiryDate = (java.sql.Date) list.get(k)[2];
```

Above is code snippet from the serverModel which implements the interface IServerModel and has instance of Mydatabase. The model is responsible for communicating with the database.

Adapter

UML Class Diagram

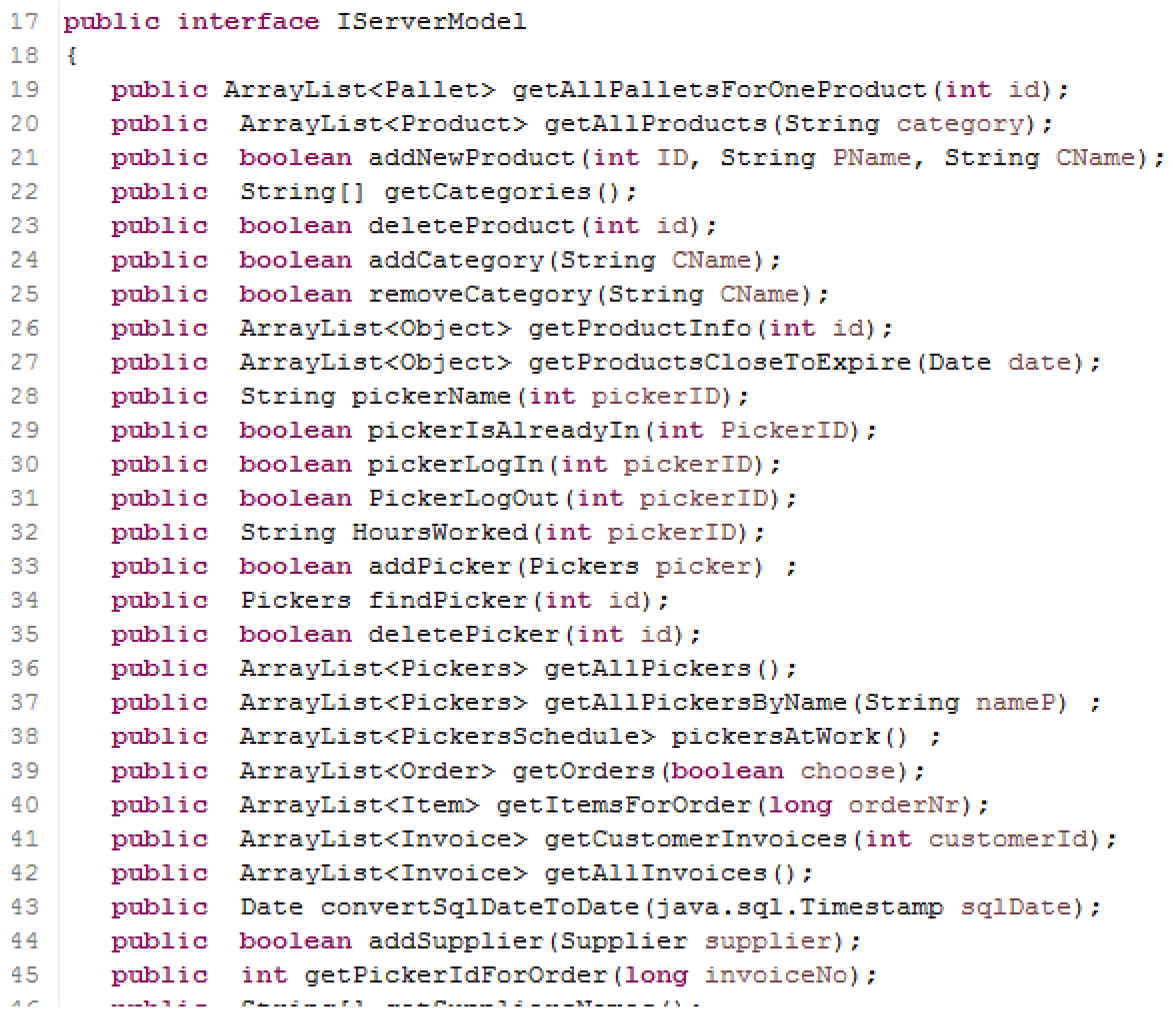


Intent

An interface class is made for the serverModel in case of changing the structure of the program to read from a file instead of reading from database.

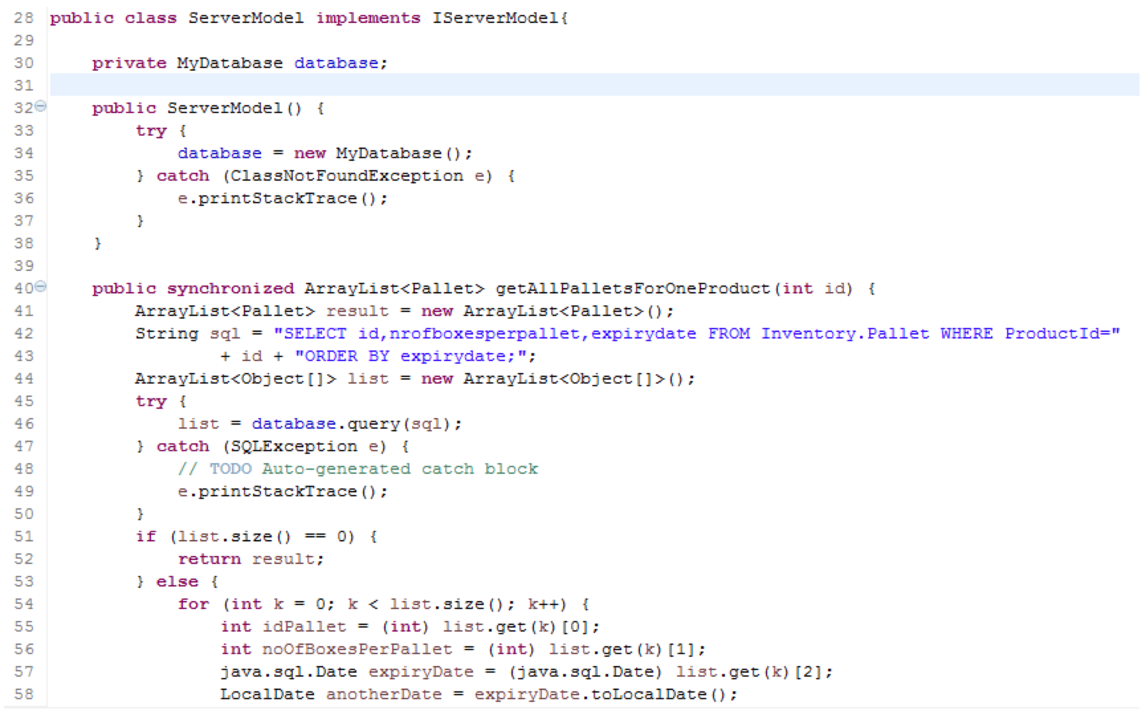
Structure

In our program we have the ServerModel is the adapter that communicate the changes with the database. The ServerModel implements the methods of the interface IServerModel. The server controller receive an instance of the IServerModel and uses the methods available in the interface. Changing the ServerModel to read from file, the ServerController will be able to call the same methods and will not be affected.



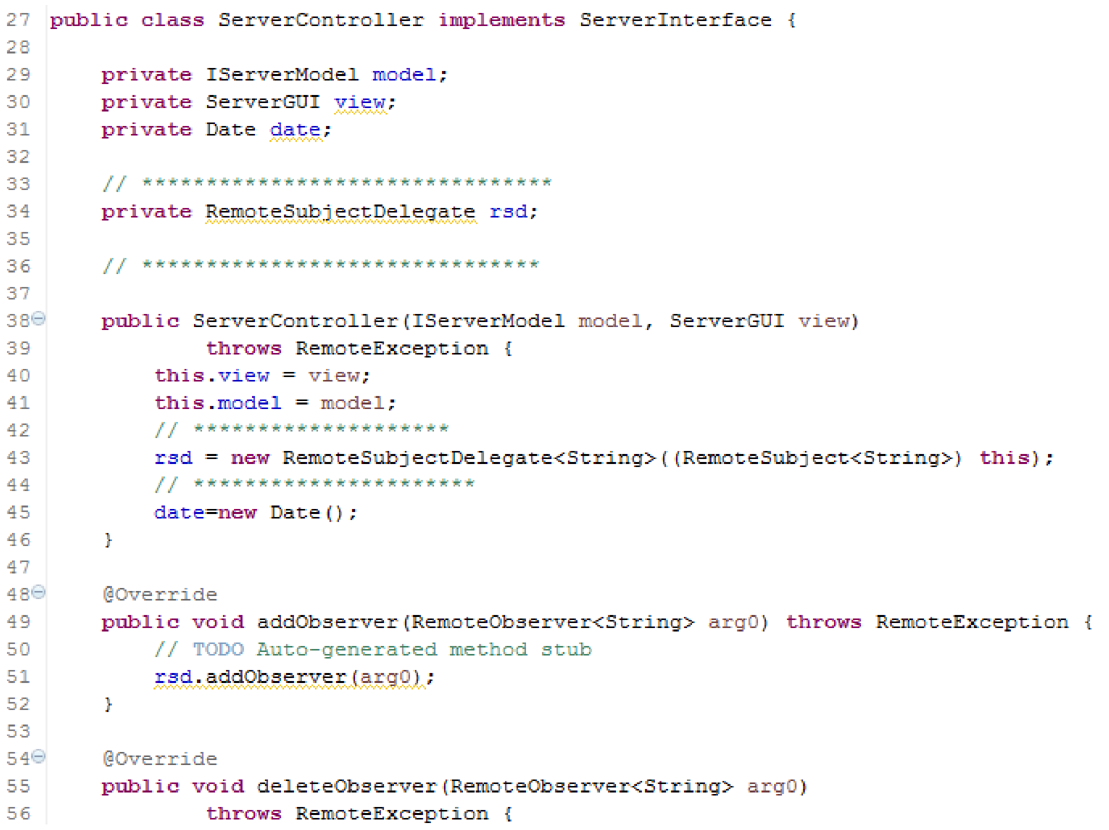
```
public interface IServerModel
{
   public ArrayList<Pallet> getAllPalletsForOneProduct(int id);
   public  ArrayList<Product> getAllProducts(String category);
   public  boolean addNewProduct(int ID, String PName, String CName);
   public  String[] getCategories();
   public  boolean deleteProduct(int id);
   public  boolean addCategory(String CName);
   public  boolean removeCategory(String CName);
   public  ArrayList<Object> getProductInfo(int id);
   public  ArrayList<Object> getProductsCloseToExpire(Date date);
   public  String pickerName(int pickerID);
   public  boolean pickerIsAlreadyIn(int PickerID);
   public  boolean pickerLogIn(int pickerID);
   public  boolean PickerLogOut(int pickerID);
   public  String HoursWorked(int pickerID);
   public  boolean addPicker(Pickers picker) ;
   public  Pickers findPicker(int id);
   public  boolean deletePicker(int id);
   public  ArrayList<Pickers> getAllPickers();
   public  ArrayList<Pickers> getAllPickersByName(String nameP) ;
   public  ArrayList<PickersSchedule> pickersAtWork() ;
   public  ArrayList<Order> getOrders(boolean choose);
   public  ArrayList<Item> getItemsForOrder(long orderNr);
   public  ArrayList<Invoice> getCustomerInvoices(int customerId);
   public  ArrayList<Invoice> getAllInvoices();
   public  Date convertSqlDateToDate(java.sql.Timestamp sqlDate);
   public  boolean addSupplier(Supplier supplier);
   public  int getPickerIdForOrder(long invoiceNo);
```

Above is a code snippet from the IServerModel. Those methods are available for the ServerController and the ServerModel has to implement it.



```java
public class ServerModel implements IServerModel{

    private MyDatabase database;

    public ServerModel() {
        try {
            database = new MyDatabase();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public synchronized ArrayList<Pallet> getAllPalletsForOneProduct(int id) {
        ArrayList<Pallet> result = new ArrayList<Pallet>();
        String sql = "SELECT id,nrofboxesperpallet,expirydate FROM Inventory.Pallet WHERE ProductId="
                + id + "ORDER BY expirydate;";
        ArrayList<Object[]> list = new ArrayList<Object[]>();
        try {
            list = database.query(sql);
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        if (list.size() == 0) {
            return result;
        } else {
            for (int k = 0; k < list.size(); k++) {
                int idPallet = (int) list.get(k)[0];
                int noOfBoxesPerPallet = (int) list.get(k)[1];
                java.sql.Date expiryDate = (java.sql.Date) list.get(k)[2];
                LocalDate anotherDate = expiryDate.toLocalDate();
```

Above is a code Snippet from the ServerModel. It implements the IServerModel.It has instance of MyDatabase (Adaptee) to read and write the changes to the database.

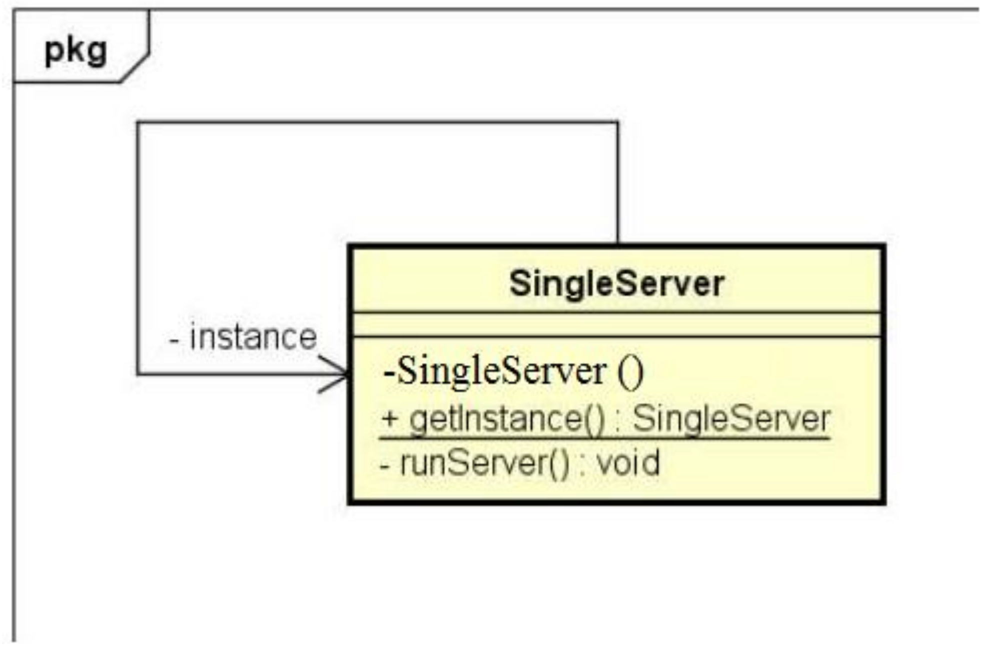


```java
public class ServerController implements ServerInterface {

    private IServerModel model;
    private ServerGUI view;
    private Date date;

    // ********************************
    private RemoteSubjectDelegate rsd;

    // ********************************

    public ServerController(IServerModel model, ServerGUI view)
            throws RemoteException {
        this.view = view;
        this.model = model;
        // ********************
        rsd = new RemoteSubjectDelegate<String>((RemoteSubject<String>) this);
        // **********************
        date=new Date();
    }

    @Override
    public void addObserver(RemoteObserver<String> arg0) throws RemoteException {
        // TODO Auto-generated method stub
        rsd.addObserver(arg0);
    }

    @Override
    public void deleteObserver(RemoteObserver<String> arg0)
            throws RemoteException {
```

Code snippet from the ServerController receive the instance of the IServerModel

Singleton

UML Class Diagram



Intent

The purpose of using singleton is to ensure that a class has only one instance, and provide a global point of access to it.

Structure

The singleton pattern is useful when we need to make sure we only have a single instance of the server; in our case, the SingleServer class. We have created a SingleServer class that runs the server through a singleton class. SingleServer class has a private constructor and a static instance of itself. This class provides a static method to get its static instance to be used elsewhere. The main class ServerMain, will use SingleServer class to get only one instance of SingleServer. In our case, we also have a private method within the SingleServer class to run the server and then we call it from the constructor. We figured that if we did not have it in the constructor, then the main would have an instance of the singleton class and can call the run method several times, which defies the purpose of a Singleton pattern. Therefore, we decided that the best way to solve it would be to change the method to be private and call it through the SingleServer constructor.

```java
public class SingleServer {
   private static SingleServer instance;
   private SingleServer()
   {
      runServer();
   }
   public static SingleServer getInstance(){
      if (instance == null)
       {
         instance = new SingleServer();
       }

      return instance;
   }
   private void runServer(){
      try{
         Registry reg=LocateRegistry.createRegistry(1099);
         ServerGUI view=new ServerGUI();
         ServerModel model=new ServerModel();
         ServerInterface server=new ServerController(model, view);
         UnicastRemoteObject.exportObject(server, 0);
         Naming.rebind("Warehouse",server);
         view.showMessage("The server is Started....");
      }
      catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

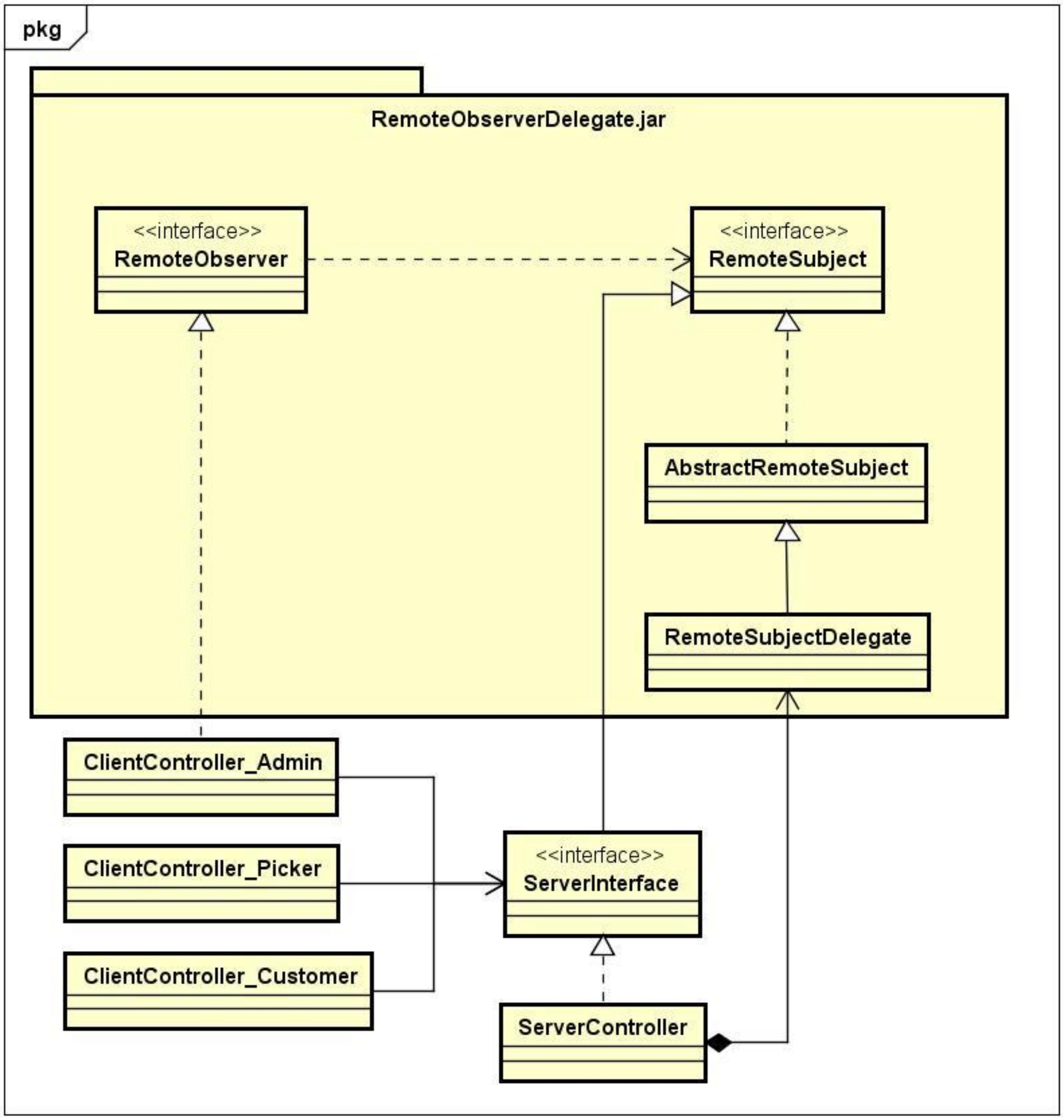
The singleton class with the private static instance, private constructor and the static method. Also extra method to run the server.

```java
public class ServerMain
{
    public static void main(String[] args)
    {
        SingleServer server = SingleServer.getInstance();
    }
}
```

Snippet from ServerMain class which use the static method to get the running server.

Remote Observer

UML Class Diagram



Intent

Since there are three types of clients in our warehouse management system (Admin, Picker, Customer), and it can be more than one of the same kind, It is very important to update every client with the changes happening.

Problem

A customer is making an order and looking at the products of the chosen category to select them and specify the amount. The administrator registers new pallets of new product which should be available at the same moment to that customer. Without that design pattern, the customer will be ordering this product from somewhere else.

Structure

The pattern has two parts: a subject being observed (the server) and observers (clients) observing state change in the subject. The subject keeps a list of observers and when the subject's state changes the observers are being notified making a loop and calling method Update for each observer.

-The RemoteObserverDelegate.jar file has to be included in the project.

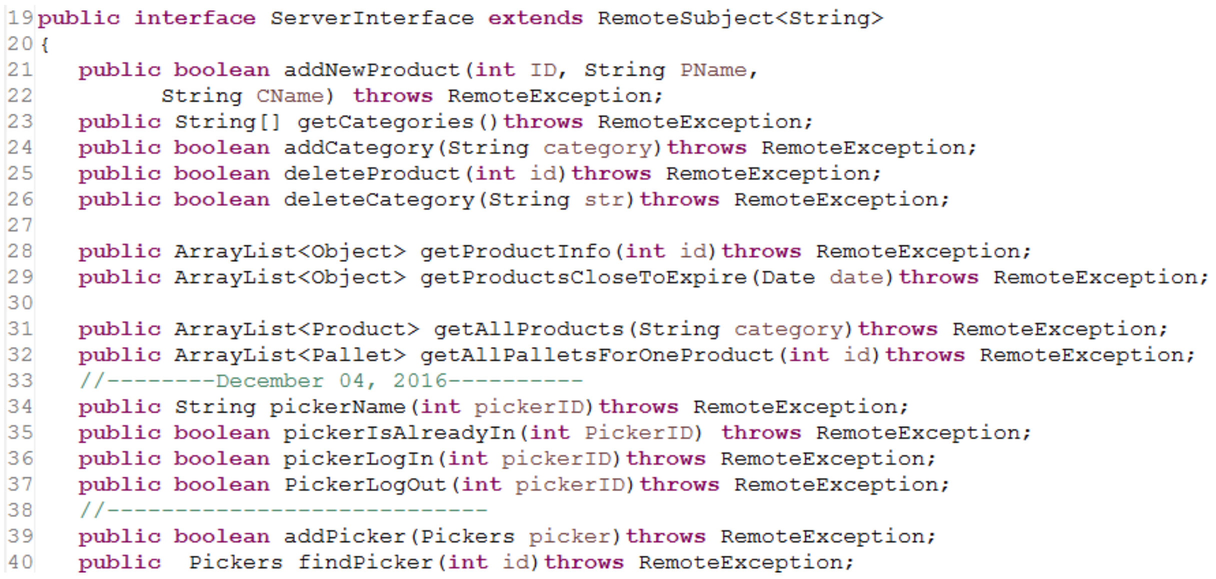
-The server interface extends the RemoteSubject class in its interface.

-The serverController has an instance of the RemoteSubjectDelegate. The server controller implements the server interface. The server controller must implement the inherited methods addObserver and deleteObserver. The instance of the RemoteSubjectDelegate can be used in those two methods to delegate the work from the server to the abstractRemoteSubject class.

- The instance of the RemoteSubjectDelegate can also be used in any method that makes change on the server to notify observers that a change has happened. The notifyObservers uses threads to communicate the change efficiently.

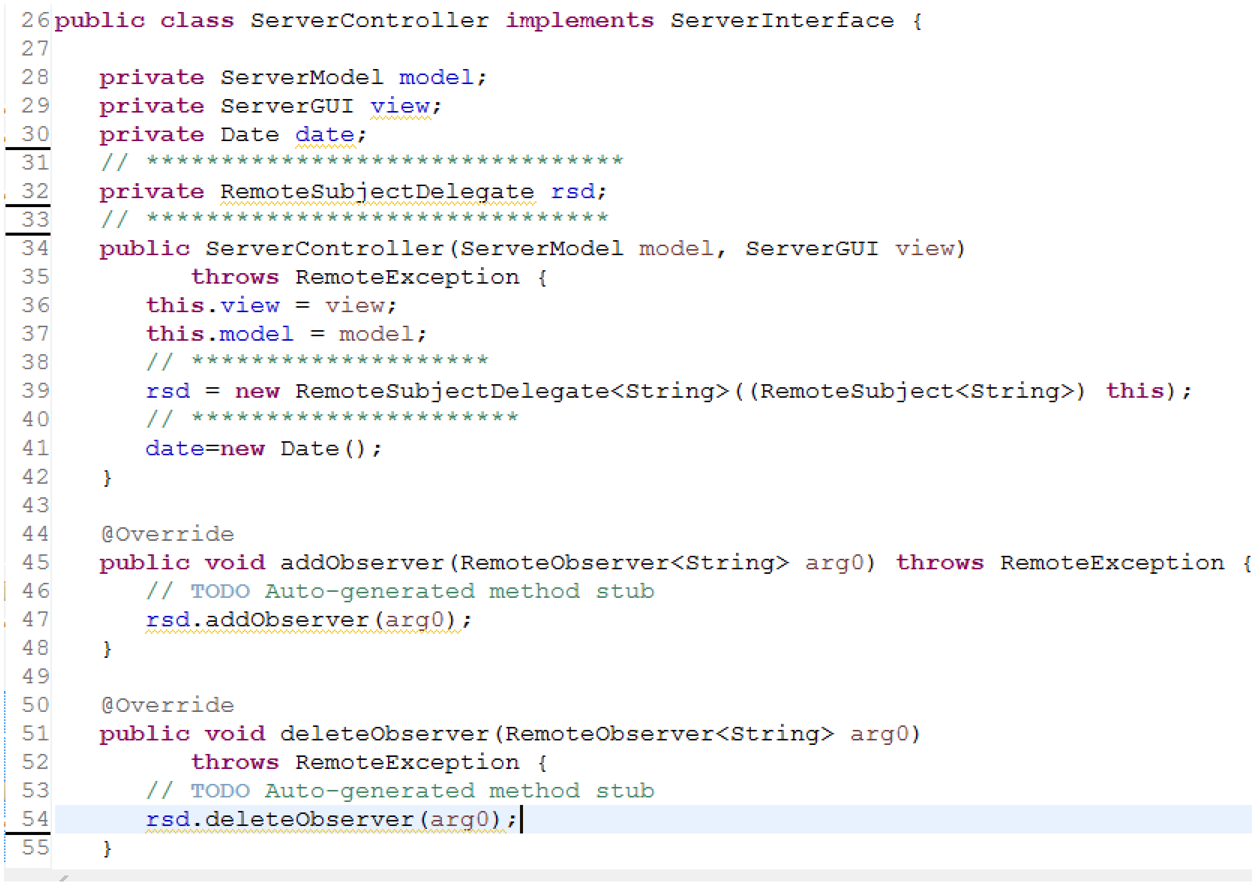
-Every client has an instance of the server interface to call the available methods for the client.

-Every client implements the RemoteObserver interface with its method update to carry on the changes that happened in the server.



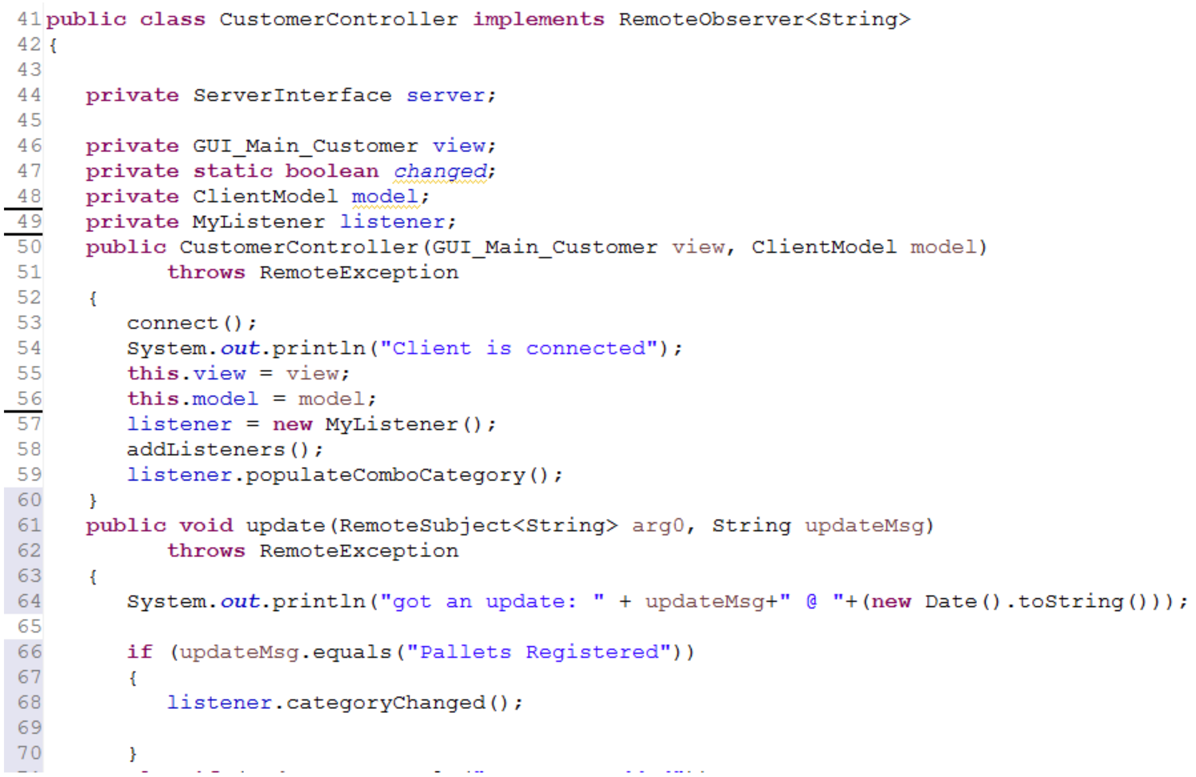
```java
public interface ServerInterface extends RemoteSubject<String>
{
   public boolean addNewProduct(int ID, String PName,
         String CName) throws RemoteException;
   public String[] getCategories()throws RemoteException;
   public boolean addCategory(String category)throws RemoteException;
   public boolean deleteProduct(int id)throws RemoteException;
   public boolean deleteCategory(String str)throws RemoteException;

   public ArrayList<Object> getProductInfo(int id)throws RemoteException;
   public ArrayList<Object> getProductsCloseToExpire(Date date)throws RemoteException;

   public ArrayList<Product> getAllProducts(String category)throws RemoteException;
   public ArrayList<Pallet> getAllPalletsForOneProduct(int id)throws RemoteException;
   //--------December 04, 2016----------
   public String pickerName(int pickerID)throws RemoteException;
   public boolean pickerIsAlreadyIn(int PickerID) throws RemoteException;
   public boolean pickerLogIn(int pickerID)throws RemoteException;
   public boolean PickerLogOut(int pickerID)throws RemoteException;
   //---------------------------
   public boolean addPicker(Pickers picker)throws RemoteException;
   public  Pickers findPicker(int id)throws RemoteException;
```

Code snippet shows the ServerInterface extends the RemoteSubject and have the signature of the methods to be available for the clients.



```java
public class ServerController implements ServerInterface {

    private ServerModel model;
    private ServerGUI view;
    private Date date;
    // ********************************
    private RemoteSubjectDelegate rsd;
    // *******************************
    public ServerController(ServerModel model, ServerGUI view)
            throws RemoteException {
        this.view = view;
        this.model = model;
        // ********************
        rsd = new RemoteSubjectDelegate<String>((RemoteSubject<String>) this);
        // **********************
        date=new Date();
    }

    @Override
    public void addObserver(RemoteObserver<String> arg0) throws RemoteException {
        // TODO Auto-generated method stub
        rsd.addObserver(arg0);
    }

    @Override
    public void deleteObserver(RemoteObserver<String> arg0)
            throws RemoteException {
        // TODO Auto-generated method stub
        rsd.deleteObserver(arg0);
    }
```

ServerController implements the ServerInterface which means also implementing the methods of the RemoteSubject: addObserver and deleteObserver. An instance of the RemoteSubjectDelegate is used to call add observers, delete observers and call the notify observer methods.

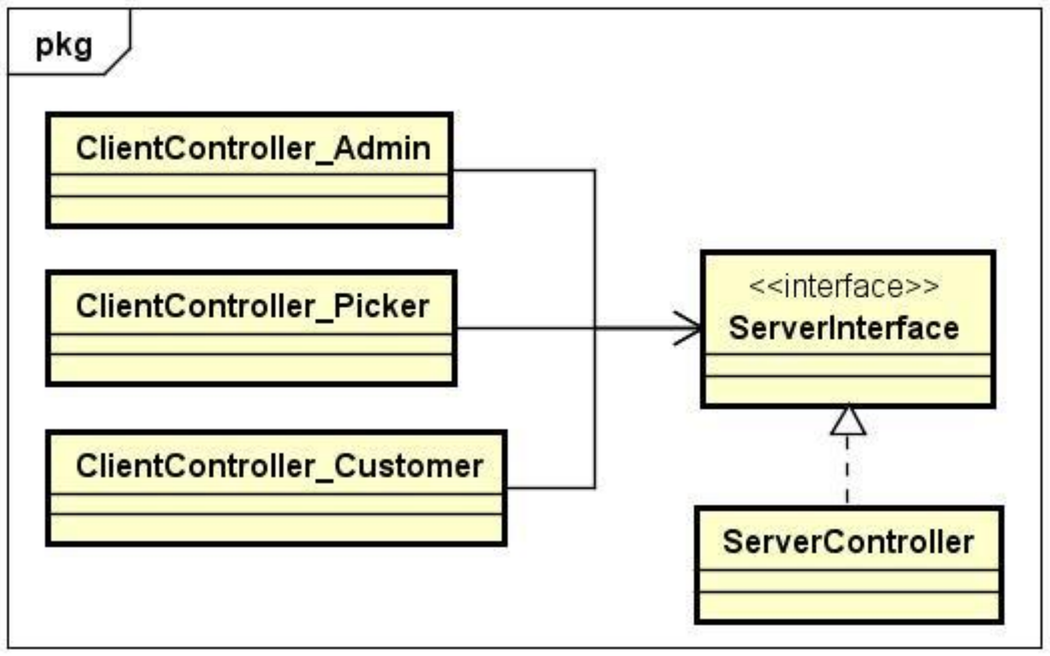


```
public class CustomerController implements RemoteObserver<String>
{

   private ServerInterface server;

   private GUI_Main_Customer view;
   private static boolean changed;
   private ClientModel model;
   private MyListener listener;
   public CustomerController(GUI_Main_Customer view, ClientModel model)
         throws RemoteException
   {
      connect();
      System.out.println("Client is connected");
      this.view = view;
      this.model = model;
      listener = new MyListener();
      addListeners();
      listener.populateComboCategory();
   }
   public void update(RemoteSubject<String> arg0, String updateMsg)
         throws RemoteException
   {
      System.out.println("got an update: " + updateMsg+" @ "+(new Date().toString()));

      if (updateMsg.equals("Pallets Registered"))
      {
         listener.categoryChanged();

      }
```

Code snippet from one of the clientControllers implments the RemoteObserver interface and its update method. It has instance of the serverInterface to call methods on the server.

Client/Server using RMI

UML Class Diagram



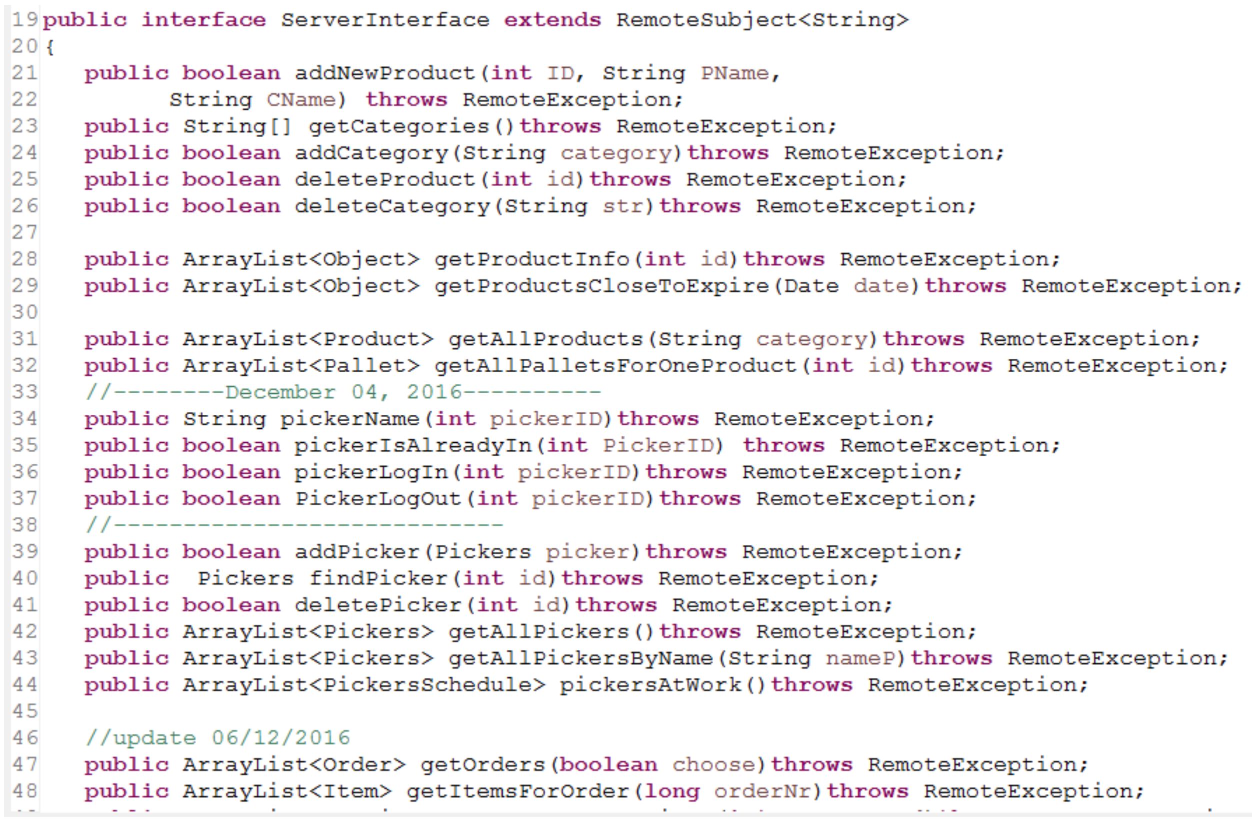
Intent

To connect our different clients (Admin, Picker, Customer) to the server and invoke the available methods for the clients.

Structure

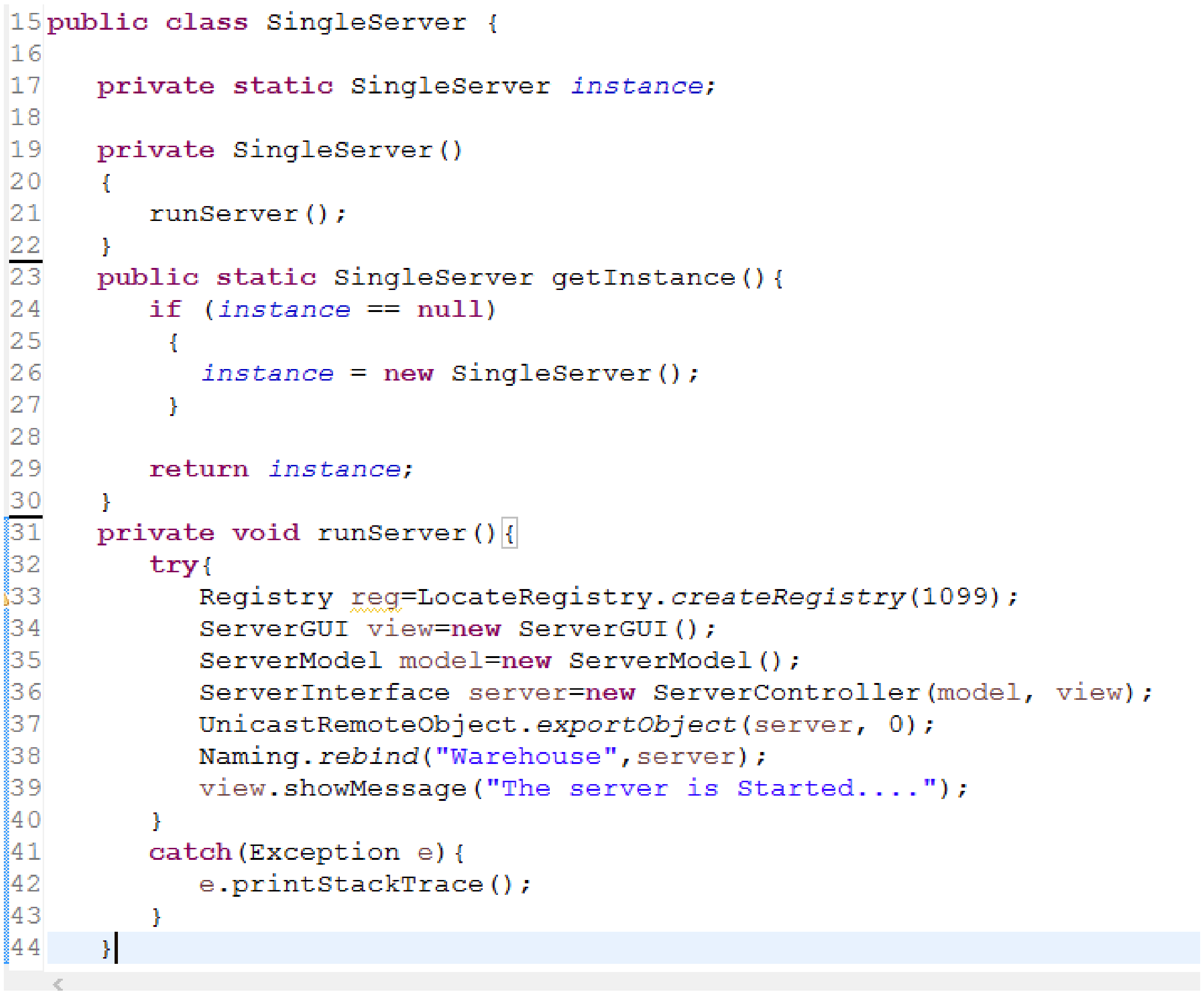
The RMI is used to allow the client controller call methods on the server side. The server has an interface. In the interface, there are all the methods that the client is allowed to call using an instance of the server Interface on the client side. In the server side the server is registered in the registry and give it a name, so the client can look the server up and bind to it using the name, port number and the IP address of the server.

For our assignment, we used RMI to build a client/server system using the Singleton pattern to ensure we have only one instance of the server running at a time. We have a ServerInterface which defines the behavior of the server. The ServerController class implements the ServerInterface and thus it is where the functionality is. In our case, we have a separate Singleton class that is called SingleServer which includes the RMI connection, creating the registry, naming the server and having an instance of the ServerModel, ServerView, and the ServerInterface. The ClientController, on the other hand, has an instance of the ServerInterface so it can access the methods regarding the different lists it needs that are read by and stored on the server side. The ClientController has a private method that connects to the server using the name given to the server so the client can look up the server using its name.



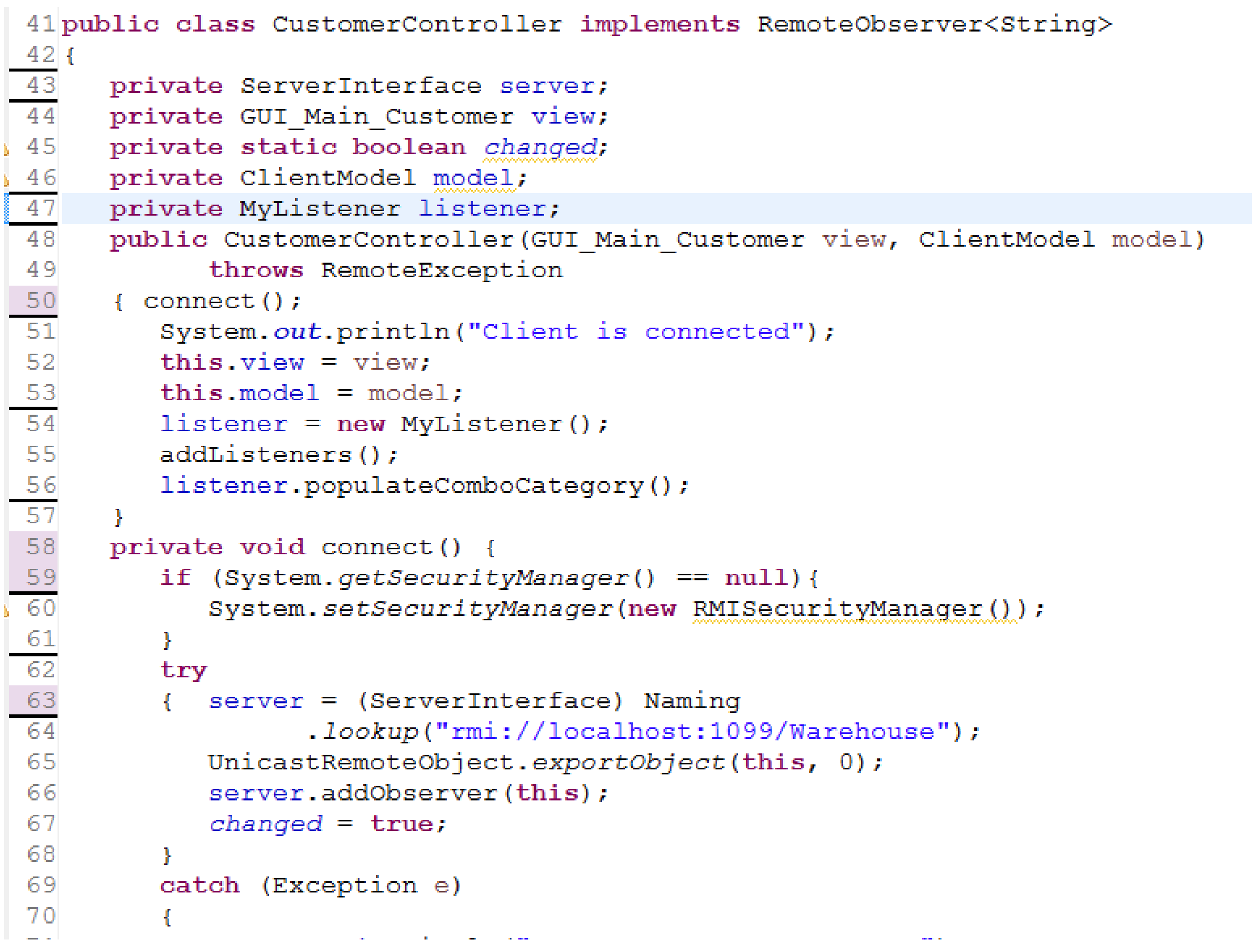
```java
public interface ServerInterface extends RemoteSubject<String>
{
   public boolean addNewProduct(int ID, String PName,
         String CName)  throws RemoteException;
   public String[] getCategories()throws RemoteException;
   public boolean addCategory(String category)throws RemoteException;
   public boolean deleteProduct(int id)throws RemoteException;
   public boolean deleteCategory(String str)throws RemoteException;

   public ArrayList<Object> getProductInfo(int id)throws RemoteException;
   public ArrayList<Object> getProductsCloseToExpire(Date date)throws RemoteException;

   public ArrayList<Product> getAllProducts(String category)throws RemoteException;
   public ArrayList<Pallet> getAllPalletsForOneProduct(int id)throws RemoteException;
   //--------December 04, 2016----------
   public String pickerName(int pickerID)throws RemoteException;
   public boolean pickerIsAlreadyIn(int PickerID)  throws RemoteException;
   public boolean pickerLogIn(int pickerID)throws RemoteException;
   public boolean PickerLogOut(int pickerID)throws RemoteException;
   //----------------------------
   public boolean addPicker(Pickers picker)throws RemoteException;
   public  Pickers findPicker(int id)throws RemoteException;
   public boolean deletePicker(int id)throws RemoteException;
   public ArrayList<Pickers> getAllPickers()throws RemoteException;
   public ArrayList<Pickers> getAllPickersByName(String nameP)throws RemoteException;
   public ArrayList<PickersSchedule> pickersAtWork()throws RemoteException;

   //update 06/12/2016
   public ArrayList<Order> getOrders(boolean choose)throws RemoteException;
   public ArrayList<Item> getItemsForOrder(long orderNr)throws RemoteException;
```

Code example from the server interface shows some methods that the client is allowed to call.



```java
public class SingleServer {

    private static SingleServer instance;

    private SingleServer()
    {
        runServer();
    }
    public static SingleServer getInstance(){
        if (instance == null)
         {
            instance = new SingleServer();
         }

        return instance;
    }
    private void runServer(){
        try{
            Registry reg=LocateRegistry.createRegistry(1099);
            ServerGUI view=new ServerGUI();
            ServerModel model=new ServerModel();
            ServerInterface server=new ServerController(model, view);
            UnicastRemoteObject.exportObject(server, 0);
            Naming.rebind("Warehouse",server);
            view.showMessage("The server is Started....");
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
```

This is the private method that exists in the SingleServer class which is then called in the constructor of the class. A registry is created with the assigned port number. Then the server is being named using Naming.rebind.



```java
public class CustomerController implements RemoteObserver<String>
{
    private ServerInterface server;
    private GUI_Main_Customer view;
    private static boolean changed;
    private ClientModel model;
    private MyListener listener;
    public CustomerController(GUI_Main_Customer view, ClientModel model)
          throws RemoteException
    { connect();
       System.out.println("Client is connected");
       this.view = view;
       this.model = model;
       listener = new MyListener();
       addListeners();
       listener.populateComboCategory();
    }
    private void connect() {
       if (System.getSecurityManager() == null){
          System.setSecurityManager(new RMISecurityManager());
       }
       try
       {  server = (ServerInterface) Naming
                .lookup("rmi://localhost:1099/Warehouse");
          UnicastRemoteObject.exportObject(this, 0);
          server.addObserver(this);
          changed = true;
       }
       catch (Exception e)
       {
```

Customer controller has instance of the server interface used to call methods on the server. It connect to the server by looking up the server using its IP, port number and name.

Test

Product Backlog Items	Implemented	Not Implemented
Log In	✓	
Admin - Manage Products	✓	
Admin - Invoice History	✓	
Admin - Check Orders	✓	
Admin - Manage Pickers	✓	
Admin - Receiving Goods	✓	
Admin - Manage Suppliers	✓	
Admin - Manage Customers	✓	
Admin - Manage Stocks	✓	
Picker – Check In/Check Out	✓	
Picker – Start Order	✓	
Picker – Finish/Incomplete an Order	✓	
Customer – Make an Order	✓	

System – automatically check the amount left in the picking positions, find another pallet in the storage, and add it to the picking position.		✕
System – filter the items in the order based on the picking positions.		✕

Result

The system was created in order to make it easier for a small warehouse business to manage their inventory, the suppliers, customers and pickers. The system covers most of the requirements that were listed in the product backlog. Two of the requirements were not fulfilled because the product owner decided that it is less important to have those features. The requirements were regarding picking positions and taking care of automatically checking the amount at the picking position.

The system consists of two applications, a client and a server. The client part has three different types of users: administrator, picker and customer. The difference between the users is made while logging into the system. Each user has different functionality for example: once logged in as an administrator more functionality is given, such as:

- Manage products
- Manage suppliers
- Manage customers
- Manage pickers
- Manage stocks
- Receiving goods
- Check orders
- List of invoices

The picker has the possibility to check in or check out of the system and to start the first order from the list of orders. If the order is not finished and the picker needs to check out, he has the option to call the order incomplete.

The only functionality that the customer has is to create an order and send it to the server. All the changes are made and sent back to the server. Depending on the functionality, the client either reads from the database or changes something and sends it back to the database. The server establishes connections to the database whenever a request has been made. By the end, it is a fully functional program, which covers all high priority requirements.

Discussion

As a group we are proud of the entire system we created but we are most proud of being able to implement a few design patterns and making them work together. With implementing the MVC correctly, the system can be easily modified whether it is adding features or even changing the interface. The separation of the classes and packages makes it easy to read and understand.

We kept coming up with new ideas all the time but due to time constraints, we had to keep some of our ideas on a side and focus on the main points of the system. Even by the end of the implementation of the project, we were still thinking of features that could be added to the system to enhance it and make it more complex and we believe it is not difficult to add more features to our system.

We wanted to also implement triggers for the database such as having a count of the number of products per category but due to time constraint we did not get to implement it and that is something we are not very proud of.

Overall, we are very proud to be able to implement most of our ideas and have them function properly.

Conclusion

The aim of this project was to create a warehouse management system that is using a client – server system and saves the data to a database. The connection between the client and the server is made using RMI. For the user to be able to manipulate the system a GUI was created. All the requirements listed in the product backlog are functional. Each feature of the system is described in details in the use case descriptions and activity diagrams which can be found in the appendices B and C respectively. The entire project proved to be working as expected, and more features can be easily added in the future.

We followed our Product Owner's requests when taking into consideration what exactly we should be implementing. Our Product Owner wanted a well-structured and easy to use system which keeps track of the inventory, the goods going into and out of the warehouse, the customers being able to make an order, and dealing with a bit of functionality on the picker side. A lot of effort was put in, in order to satisfy the Product Owner's needs. The system can now take care of the main functionalities of a warehouse management system. We have created three different frames, one for each user (administrator, picker, and customer) which are accessed using specific passwords. This was the way we thought would be best when separating all the functionality and make it easy to understand by end-users.

All actions are being saved to a database, so in case of emergency such as power outage or whatnot, the data can be found through the database. All in all, we met the original requirements and our Product Owner was quite satisfied with the system.

List of References

Java2s.com. (2016). *JList selection changed listener : List « Swing JFC « Java*. [online] Available at: http://www.java2s.com/Code/Java/Swing-JFC/JListselectionchangedlistener.htm [Accessed 6 Dec. 2016].

Seasite.niu.edu. (2016). *Adding and Deleting Items from a JList*. [online] Available at: http://www.seasite.niu.edu/cs580java/JList_Basics.htm [Accessed 7 Dec. 2016].

My.vertica.com. (2016). *SET DATESTYLE*. [online] Available at: https://my.vertica.com/docs/7.1.x/HTML/Content/Authoring/SQLReferenceManual/Statements/SET/SETDATESTYLE.htm [Accessed 9 Dec. 2016].

JFrame?, H. (2016). *How to get all elements inside a JFrame?*. [online] Stackoverflow.com. Available at: http://stackoverflow.com/questions/6495769/how-to-get-all-elements-inside-a-jframe [Accessed 8 Dec. 2016].

Methodsandtools.com. (2016). *Understanding the Unified Process (UP)*. [online] Available at: http://www.methodsandtools.com/archive/archive.php/archive.php?id=32 [Accessed 13 Dec. 2016].

Appendix

Appendix A – User Guide

Appendix A can be found in a separate pdf file titled as Appendix A – User Guide, within this folder.

Appendix B – Use Case Descriptions

Appendix B can be found in a separate pdf file titled as Appendix B – Use Case Descriptions, within this folder.

Appendix C – Activity Diagrams

Appendix C can be found in a separate pdf file titled as Appendix C – Activity Diagrams, within this folder.

Appendix D- Burndown Chart Excel Sheet

Appendix D can be found in a separate excel file titled as Appendix D – Burndown Chart Excel Sheet, within this folder.

Appendix E – Setting Up the System

Appendix E can be found in a separate excel file titled as Appendix E – Setting up the System, within this folder.