PROJECT REPORT

Smart Inventory Management System (SIMS)

SEP2-A17

Group no: 3

Constantin Cornea(260017)

Daniela Crucerescu (260110)

Dumitru Muntean (261824)

Marius Papa(261306)

Supervisors:

Troels Mortensen

Jens Cramer Alkjaersig

15 December 2017

Table of Contents

1.List of figures

Abstract

SIMS is a system that helps primarily small to medium shops to organize their warehouse, having the advantages of being simple and inexpensive. It started by taking in consideration how a warehouse is working and what are the problems and needs of a shop owner managing the warehouse. This information was translated into requirements in the analyses stage. Features like searching for a product by specific criteria, keeping information about suppliers, orders and batches, keeping track off all products -the need for a safe data storage went into the implementation phase. Meeting those needs and several others identified in the talk with target clients helps the shop owner to ensure a more secure and competitive place for his business on the market. SIMS is developed using Java as the programming language. It follows the client-server architecture. All data is stored into a data base to reduce the risk of data loss. A login with a user name and password feature was unnecessary for SIMS. Therefor was not implemented. After identifying the requirements and implementing them accordingly, tests were performed to guarantee the good and consistent workflow of the system. SIMS can support changes and improvements if the growing market askes for such thing.

Introduction

Managing a shop can have a lot of challenges. One of the most important is keeping track of goods that are coming in and going out of the shop. If the shop is growing, knowing exactly what is left in the warehouse, what needs to be ordered or the expire date for all or for a specific branch can be difficult or even impossible to be done efficiently.

Most of today supermarkets are using pieces of software that are doing just that and even more. Those programs are usually expensive therefore smaller shops can not afford to have it.

SIMS is coming to meet the needs if those shops primarily. Is a simple and inexpensive solution for shops that are growing and have to manage more and more products but don't have the budget for expensive systems.

SIMS is holding information about all the products that can be found in the shop. The user can add new products, search for a specific item or delete one with a simple click. The system is keeping track of products letting the user know when and what has to be ordered or which items are about to expire and should go to discount.

Ordering can also be done whit SIMS. This system is the perfect manager assistant helping small to medium shops to reduce the loos of money, having a better organization and become competitive on the market.

It also reduces the risk of data loss in case of hardware failure by saving and retrieving data in a Database.

More of what SIMS can do is presented in the requirements list that follows.

SIMS(Smart Inventory Management System)

Analysis

Requirements

1) As an administrator I would like to insert new products into the systems with the following information:
   - barcode;
   - Name
   - Description
   - Price
   - Distributor
   - The minimum amount - the limit when system puts the product in order list.
   - Expired date
   - Default number of items to be ordered
   - Amount of items received
2) As an administrator I would like to register received products in the system:
   Steps:
   -Enter Barcode
   - Press Search
   If product is found:
   - enter amount and expired date
   Else:
   - pop up message "Product Not Found" with two options
     - add new item
     -cancel
   - Check all the information

3) As an administrator I would like to see all the products:
   - Barcode
   - Batch Number
   - Distributor
   - Name
   - Description
   - Expired Date
   - Price
   - Amount of items left
4) As an administrator I would like to search for a particular product by:
   - Barcode
   - Name
   - Supplier
   - Key words description
5) As a user I would like to be notified when the stack is low on specific products:
   -P.O
   -The amount is set from the beginning in "Add New Item" Section

6) As a *cashier* I would like to insert:
   - Barcode of items and the number of products to be sold
7) As *a cashier* I would like the system to create automatically an invoice of total price when the byer is done
8) As a *cashier* I would like to remove an products from the invoice
9) As an administrator I would like to edit product information:
   - Name
   - Description
   - Minimum Amount
   - Amount
   - Supplier
   - Amount to be ordered
   - Price
10) As an *administrator* I would like to remove a product from the list
11) As an *administrator* I would like to remove a list of products.
12) As an *administrator* I would like to be able to order products:
    -Check the list with the items to be ordered
    -Add the new item in the list
    -Change the default number of items to be ordered
    -Remove an item from the list
    -The order is complete when the button order is pressed

13) As an *administrator* I would like to be notified when the products are close to expired date

UseCase Diagram

The following picture represents the Use Case Diagram. The diagram illustrates the use cases how the administrator interacts with the system.

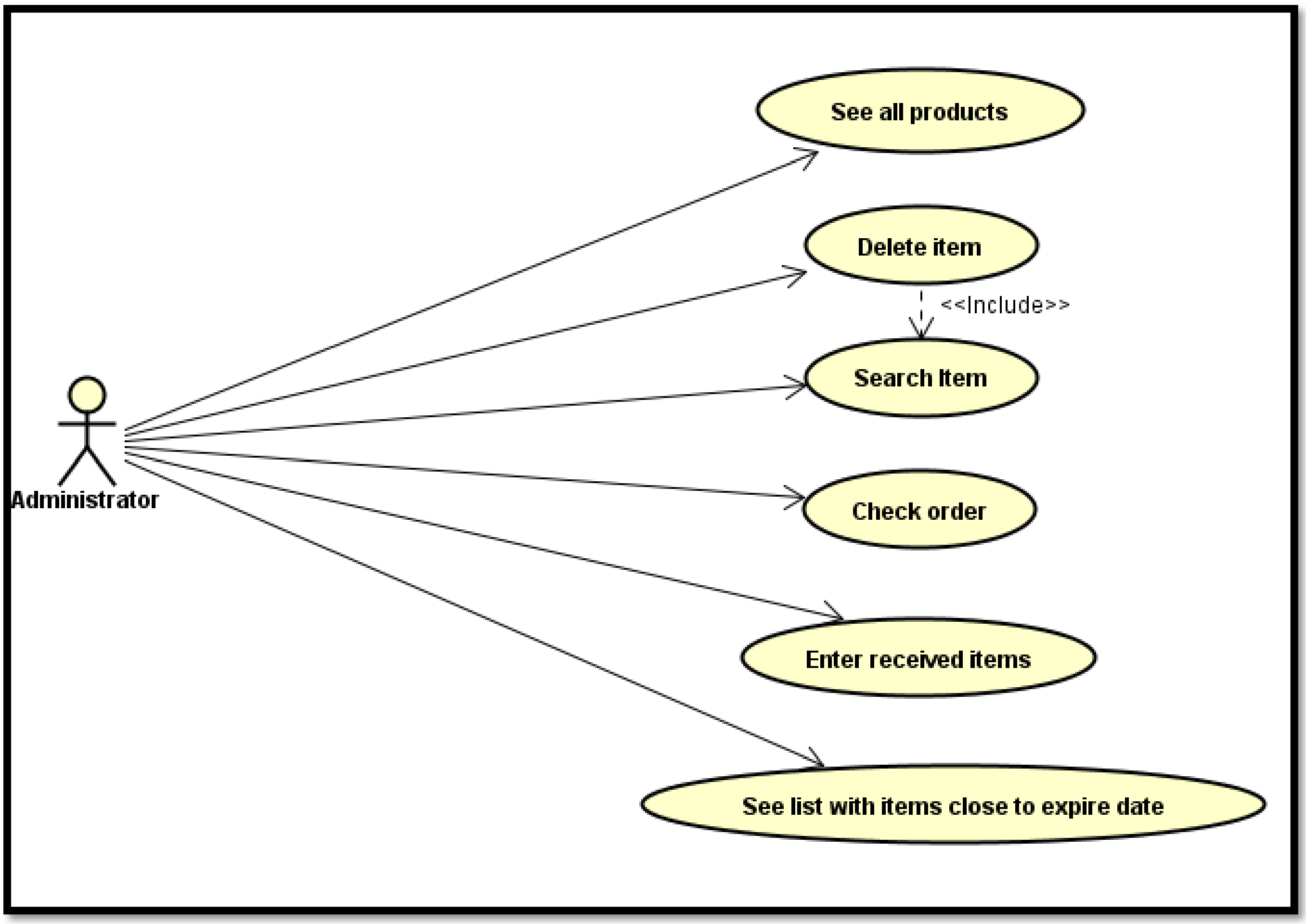


Figure 1 Use Case Diagram for Administrator

The next picture shows how the Cashier is interacting with the system.

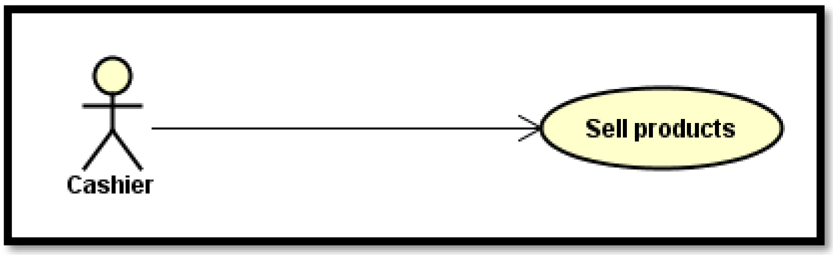
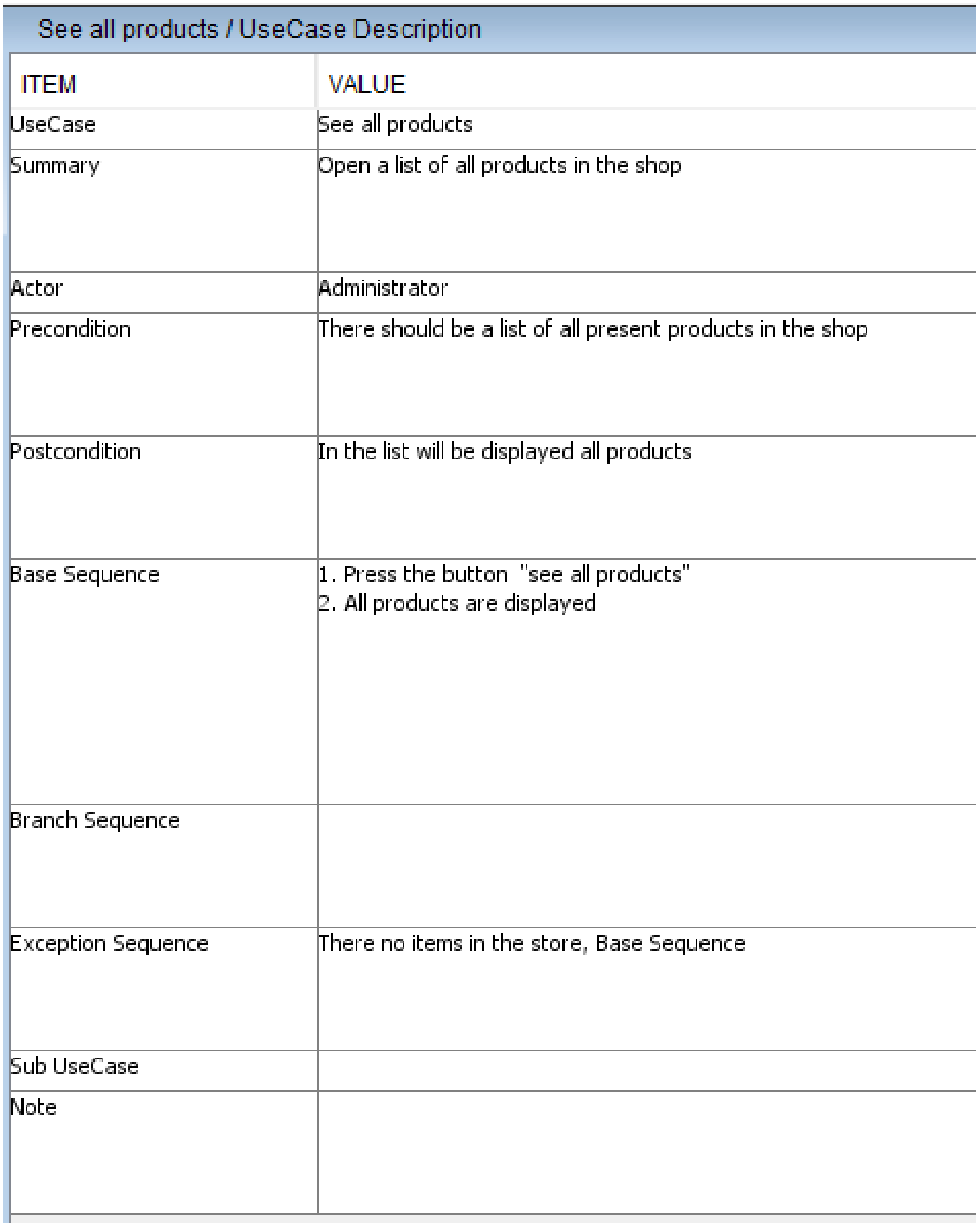


Figure 2: Use Case Diagram for Cashier

UseCase Description



See all products / UseCase Description

ITEM	VALUE
UseCase	See all products
Summary	Open a list of all products in the shop
Actor	Administrator
Precondition	There should be a list of all present products in the shop
Postcondition	In the list will be displayed all products
Base Sequence	1. Press the button "see all products" 2. All products are displayed
Branch Sequence	
Exception Sequence	There no items in the store, Base Sequence
Sub UseCase	
Note	

Figure 3 Use Case Description

The rest of use case description you can see in the Appendix " A".

Activity Diagrams

The *Figure 4* below illustrates the workflow step by step of procedure that Administrator of the Market have to follow to see all products in his store. Firstly, he should press the button “All products”. If the system has some products inside they will be displayed on the screen. Otherwise, the system will pop up a message “There are no products”.

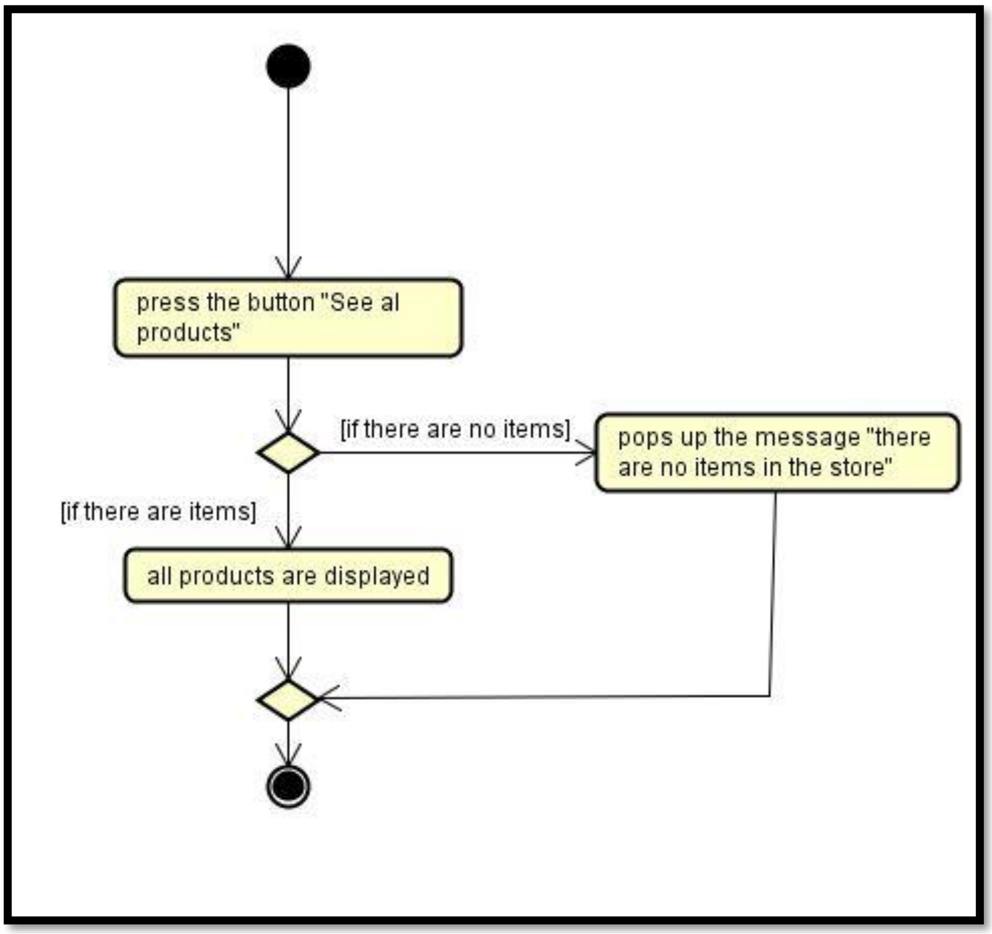

press the button "See al products"
[if there are no items]
pops up the message "there are no items in the store"
[if there are items]
all products are displayed

Figure 4 Activity Diagram for See all products

The rest of UseCase you can see in Appendix “ E“.

Design

In the picture (*Figure 5)* is designed the whole class diagram of the system. There is one server and to clients. The classes that are related to the server a blue-colored. The classes related to the administrator part are green-colored. The cashier classes are orange colored. The shared classes are yellow-colored. RMI is used to make the connection between the server and the client. The connection is marked with a blue wi-fi icon in the diagram. Database is used to store all the information of the system. JDBC was used to make the connection between the server and the database.

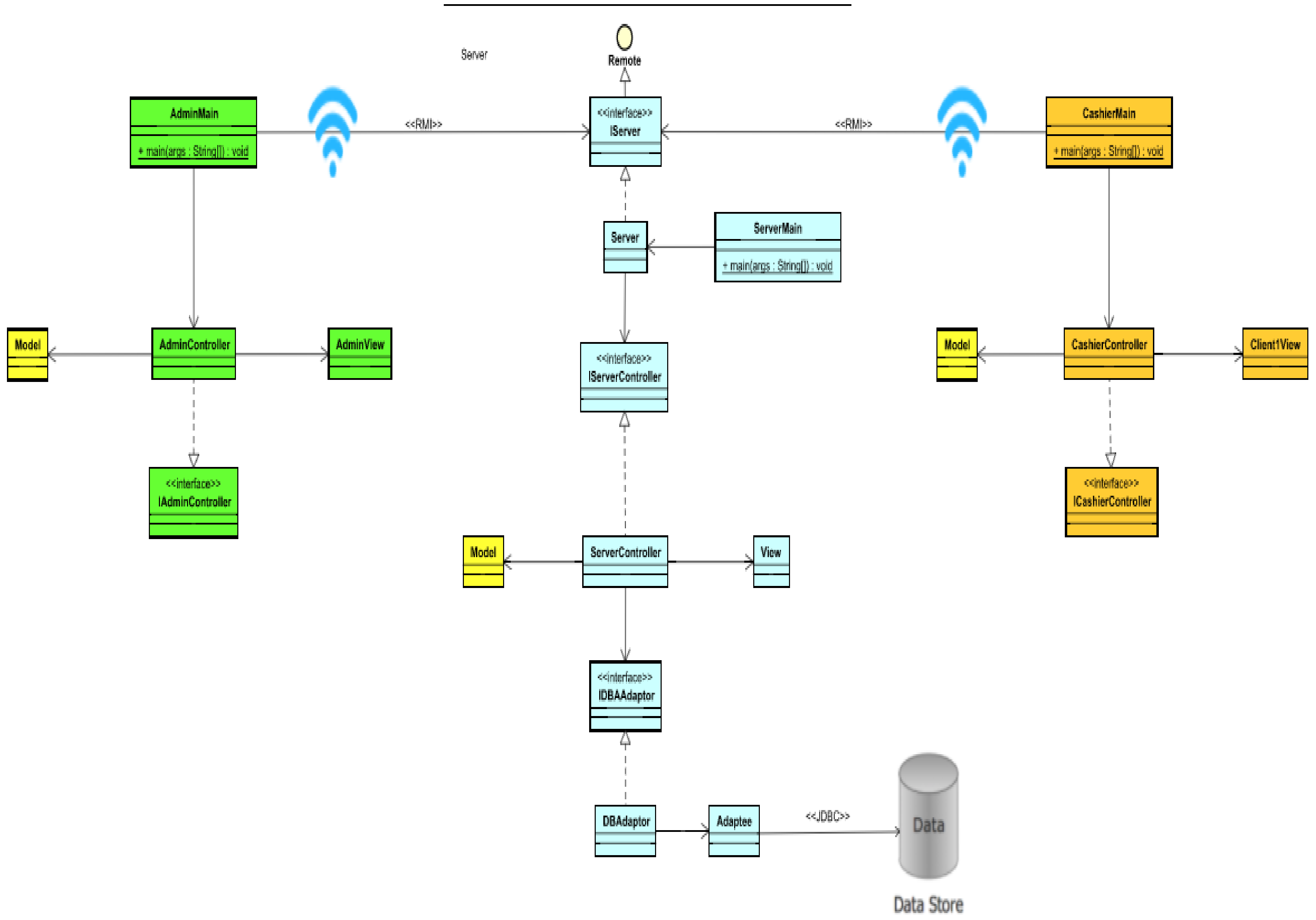


Figure 5 Class Diagram

1.Model

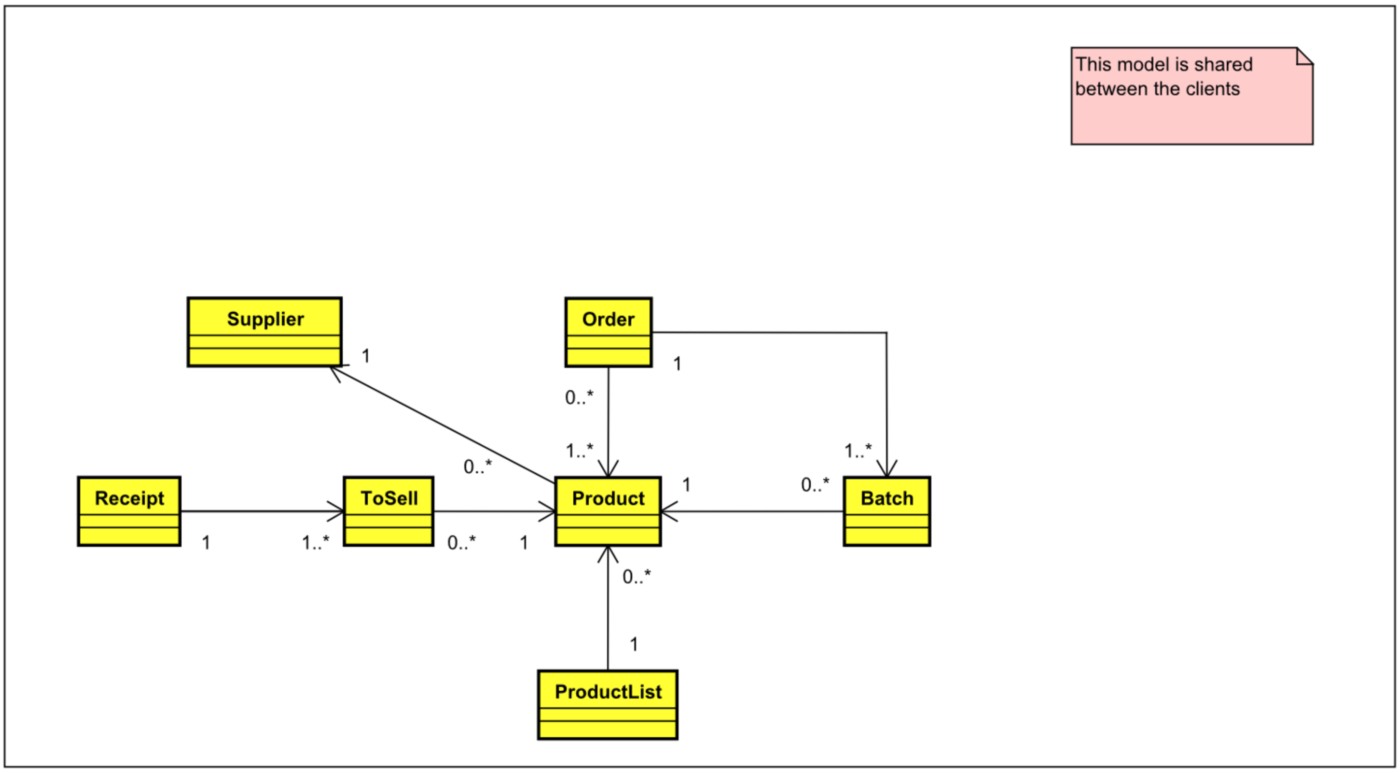


Figure 6 Model Class Diagram

The entire system was designed based on the MVC (Model – View – Controller) pattern. In this way there are three important parts:

- Model represents the data that the system is working with;(Figure 6)
- View is responsible to display the data from the model, so the user can easily see it when needed;
- Controller is used to handle the input from the user and convert it into some entities to modify the model;

Both clients and server follow this pattern to make a clean separation between graphical user interface and the model.

In the picture (Figure 7) bellow is displayed the MVC design in the server side.

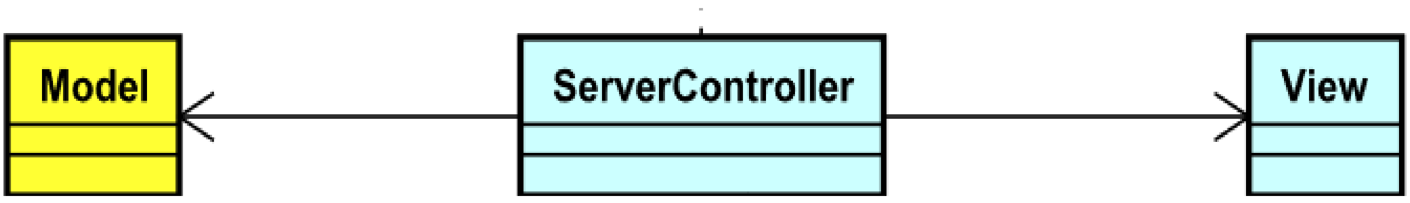


Figure 7 MVC Design Pattern

One of the core importance of the system is to be able to store and read data from the database. Adaptor pattern was used to make the connection between two programming languages: Java and PostgreSQL.

The server is connected to the database. Therefore, adaptor pattern is used to make the communication between database and Java possible. Figure 8 shows how it was designed. IDBAAdator interface is implemented by DBAAdaptor class. It is responsible to take the message from Java and translate it into SQL statement, so the database can execute it. Adaptee class is responsible to run SQL statements made by adaptor and also to store the database address, user and password.

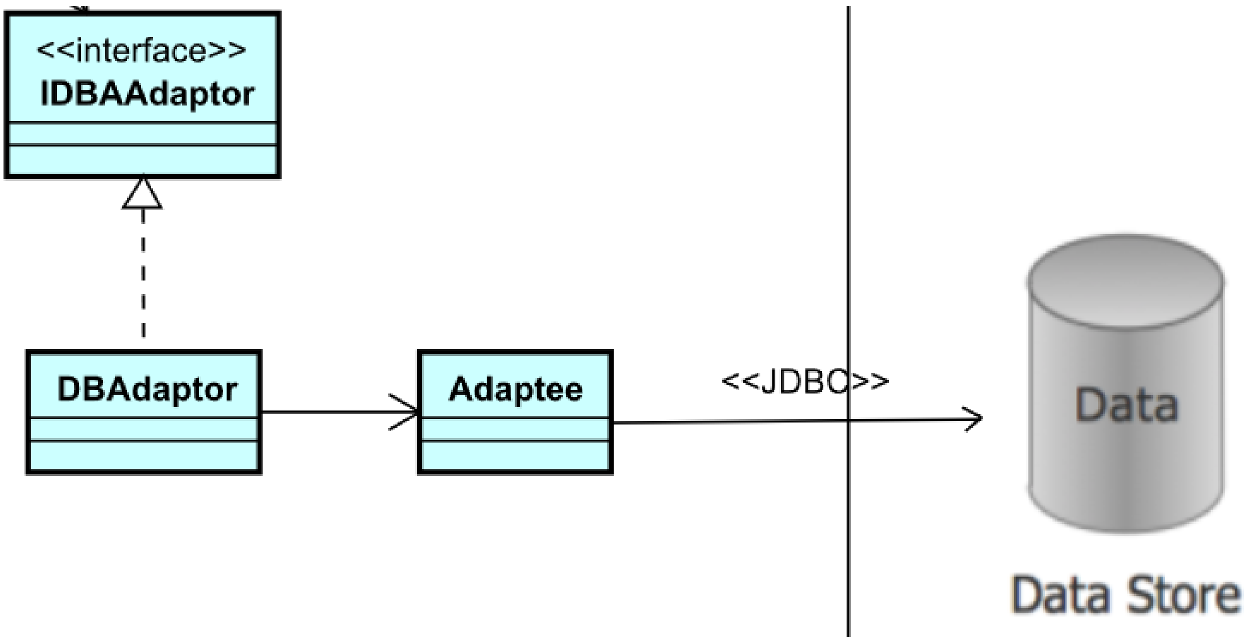


Figure 8 Adaptor Design Pattern

Flyweight design pattern was used to minimize the amount of memory used by the system. It is used in the view part of the both clients whenever it is needed to display panels that contains information about product. ProductPanelFactory class is responsible to create and store panels once they are created. Afterwards, products panel are retrieved from product panel factory every time the view wants to display them.

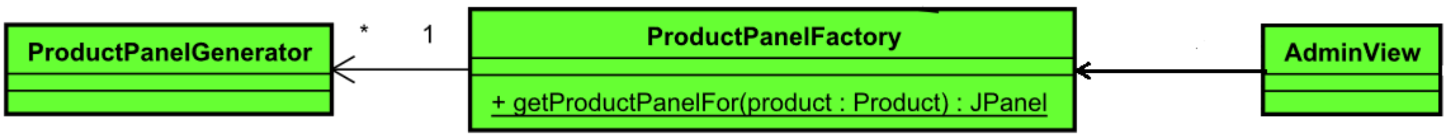


Figure 9 Flyweight Design Pattern

Singleton design pattern is used to monitor the logs on the server side. It is responsible to create a single log file to register all the log lines in. In the picture (Figure 10) singleton pattern is displayed.

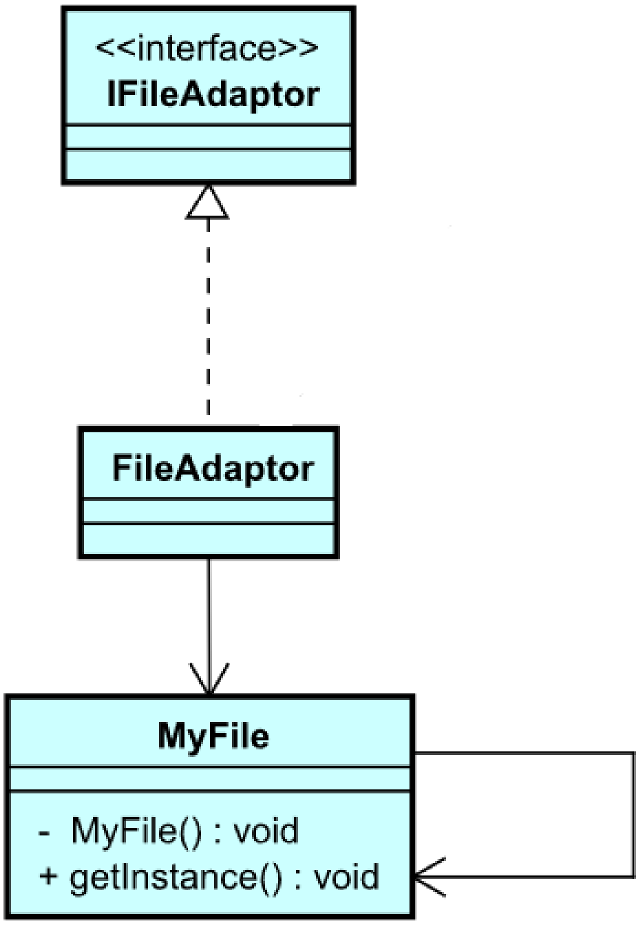


Figure 10 Singleton Design Pattern

For more detailed diagram please check Appendix C

Observer design pattern is used on the cashier side. The main purpose of it is to update the view once there are some changes in the model. In the figure (Figure 11) is displayed how it was designed.

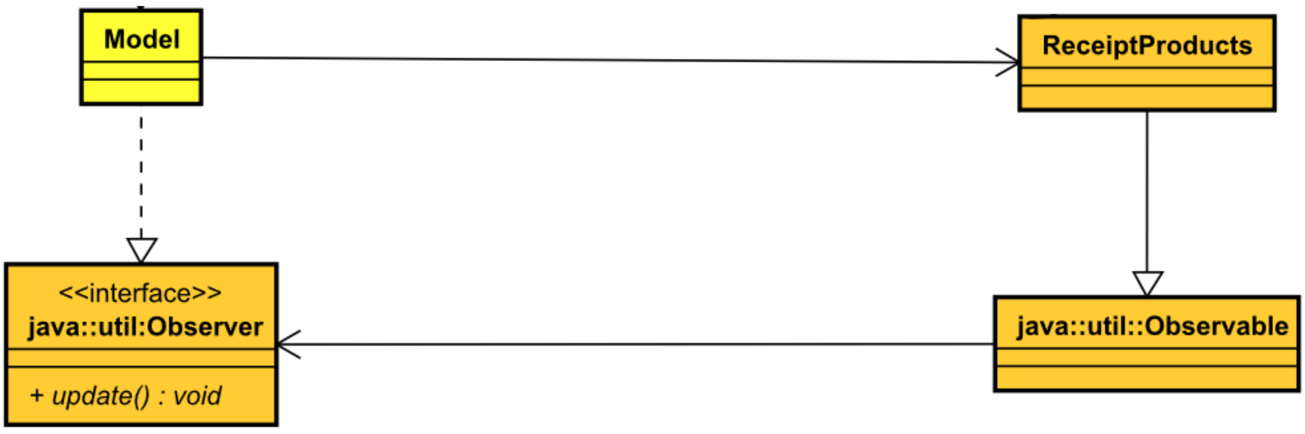


Figure 11 Observer Design Pattern

In the EER diagram (*Figure 12)* bellow is represented how the information is structure in the database. Is consists of six table that keeps truck of the products that comes in and goes out of the system.

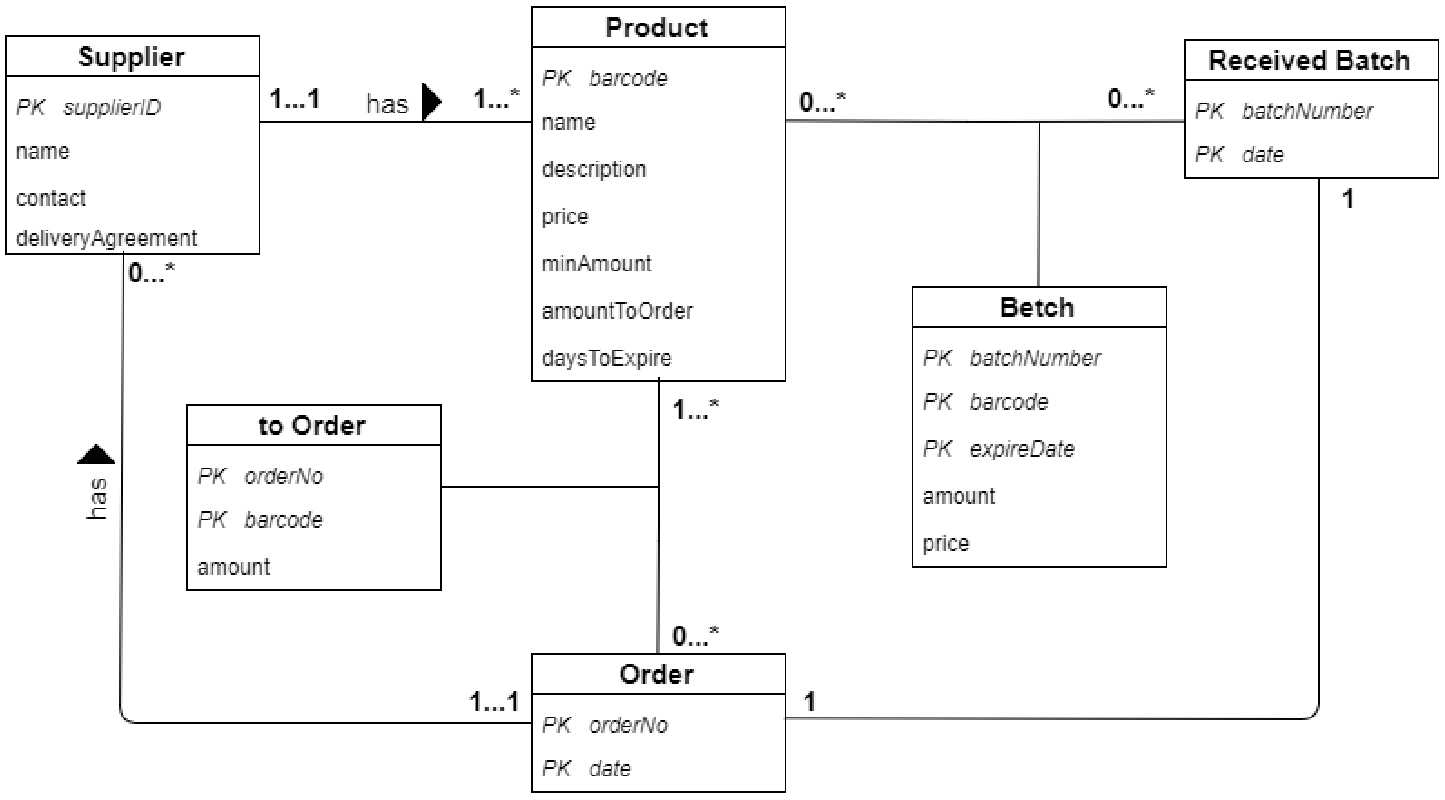


Figure 12 EER Diagram

Based on the above EER diagram *Figure 12* logical mapping was created:

Supplier(<u>supplierId,</u> name, contact, deliveryAgreement)

Product(<u>barcode</u>, name, description, price, minAmount, amountToOrder, daysToExpire, supplierId)

ReceivedBatch(<u>batchNumber</u>, <u>date</u>, orderNo)

toOrder(<u>orderNo</u>, <u>barcode</u>, amount)

Order(<u>orderNo</u>, date)

Batch(<u>batchNumber, barcode</u>, expireDate, amount, price)

<u>For more detailed UML diagrams please check Appendix “C”</u>

2.Client-Server Protocol

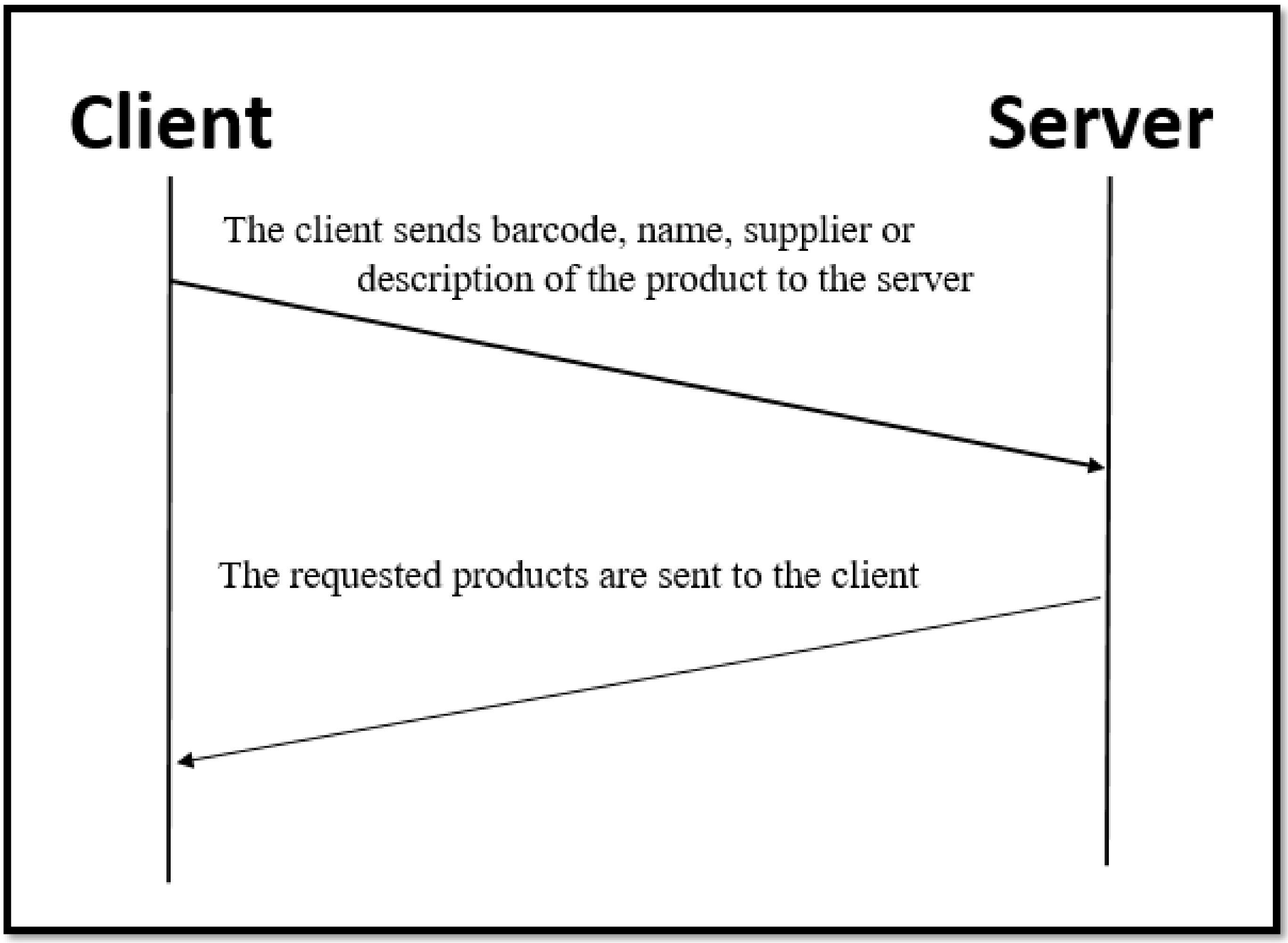

Client
Server
The client sends barcode, name, supplier or description of the product to the server
The requested products are sent to the client

Figure 13 Client Server Protocol

The client/server architecture protocol is displayed in the picture above. It represents how the client interacts with the server. When a client needs products to be displayed, it sends a request with the information related to the products it is looking for. It can get products based on their barcode, name, supplier or key words from description. The client can also make a combination of them to narrow down the search result. The server processes the request and replies with a list of products that matches the given information.

3. Graphic User Interface

The GUI is implemented for both clients: administrator and cashier.

In the picture (*Figure 14)* below the start page of the administrator of the market is displayed. It contains a few buttons and some text fields that makes the system to be functional and easy to use in the same time.

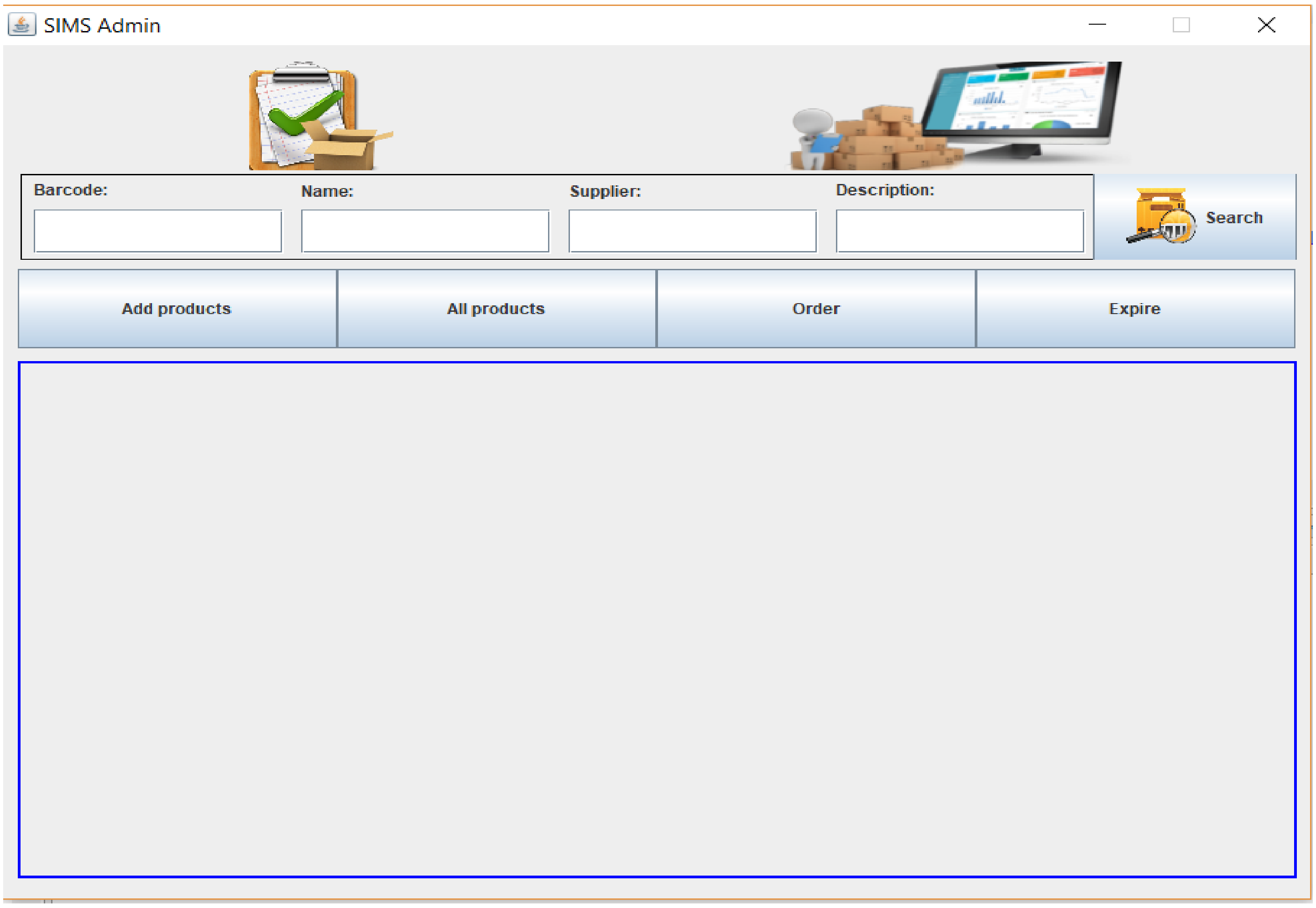


Figure 14 GUI Main Page

The main page of the administrator makes functional the following user's stories:

- As an administrator I would like to insert new items into the systems;
- As an administrator I would like to see all the items;
- As an administrator I would like to search for product by barcode, name, supplier, key words;

In the picture (*Figure 15)* bellow if relevant for the one of the most important user's stories:

- As an administrator I would like to insert new items into the systems;

All the fields should be completed in order to add whether a new product in warehouse or to update the amount of an existing one.

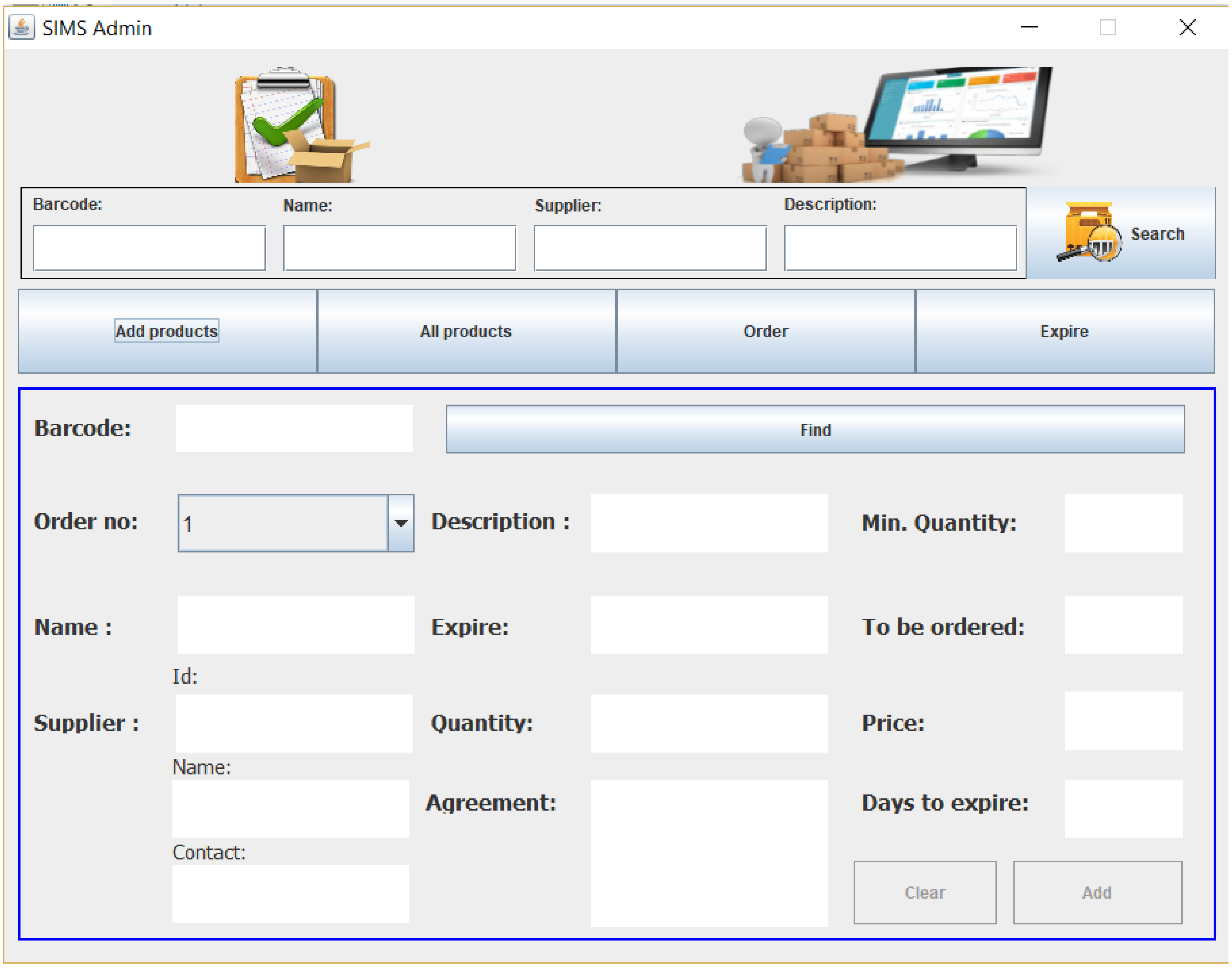
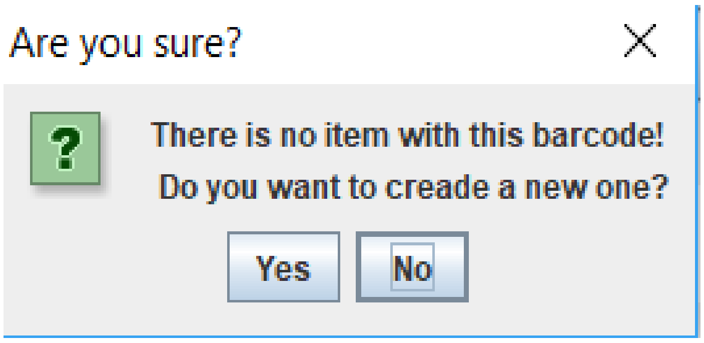


Figure 15 GUI Add products page

A pop-up message is displayed when a user wants to add an item which is not in the warehouse. The figure (*Figure 16)* shows how it looks like.

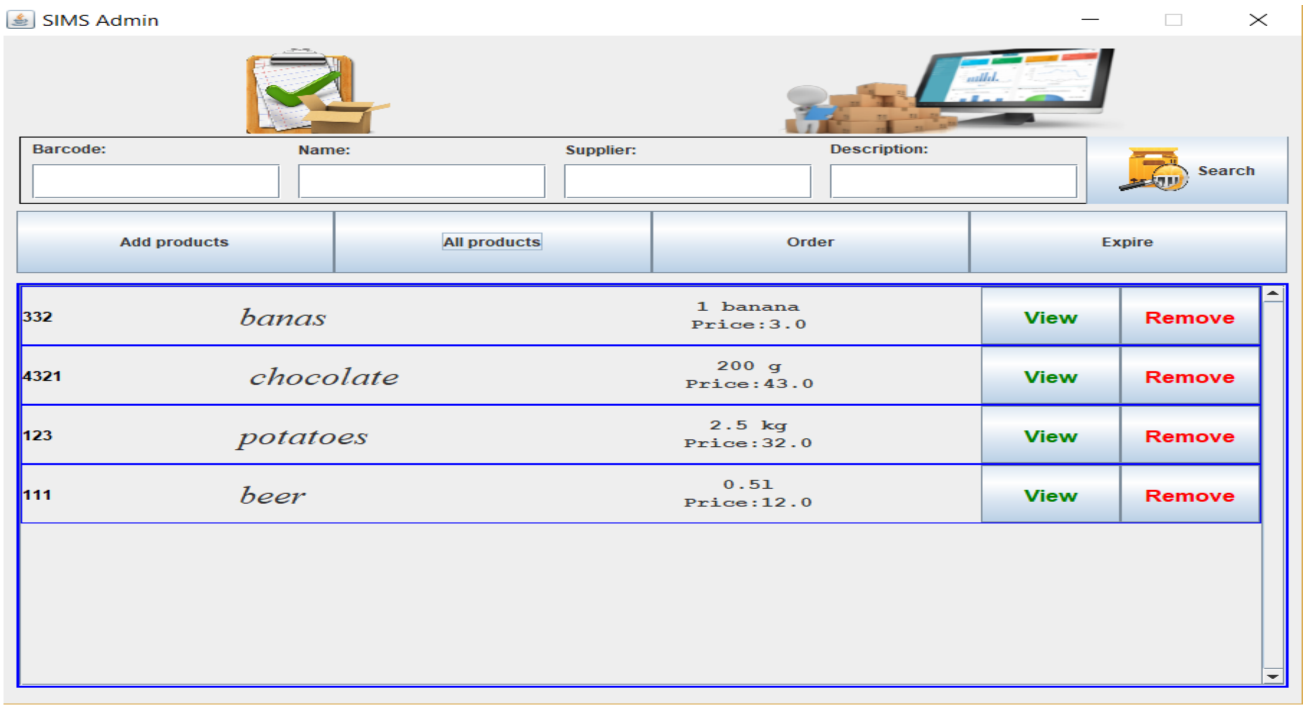

Are you sure?

There is no item with this barcode!
Do you want to creade a new one?

Figure 16 GUI Pop Up message

In the next picture *Figure 16* “All products page is displayed”. It makes functional the following user’s story:

- As an administrator I would like to see all the items;



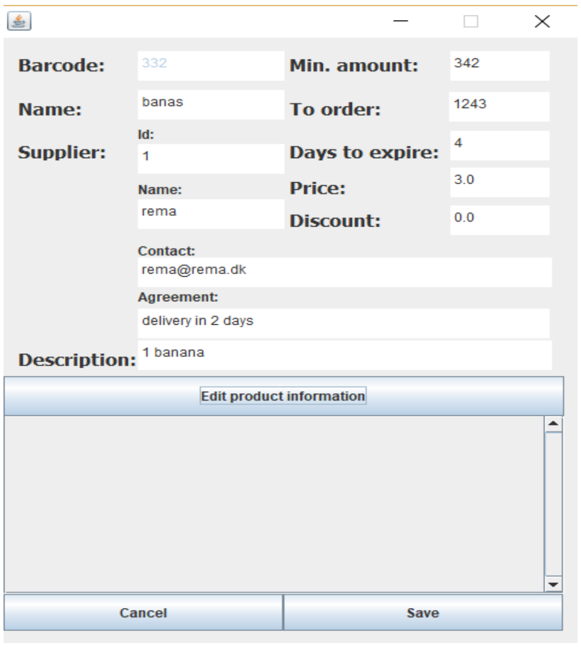
332	banas	1 banana Price:3.0	View	Remove
4321	chocolate	200 g Price:43.0	View	Remove
123	potatoes	2.5 kg Price:32.0	View	Remove
111	beer	0.5l Price:12.0	View	Remove

Figure 17 GUI All products page

In the same way the next user story is performed:

- As an administrator I would like to search for a particular product by barcode, name, supplier, key words;

In the picture *Figure 18* bellow is displayed how the user can see all the information about a particular product.



Barcode: 332
Name: banas
Supplier: Id: 1 Name: rema Contact: rema@rema.dk Agreement: delivery in 2 days
Min. amount: 342
To order: 1243
Days to expire: 4
Price: 3.0
Discount: 0.0
Description: 1 banana

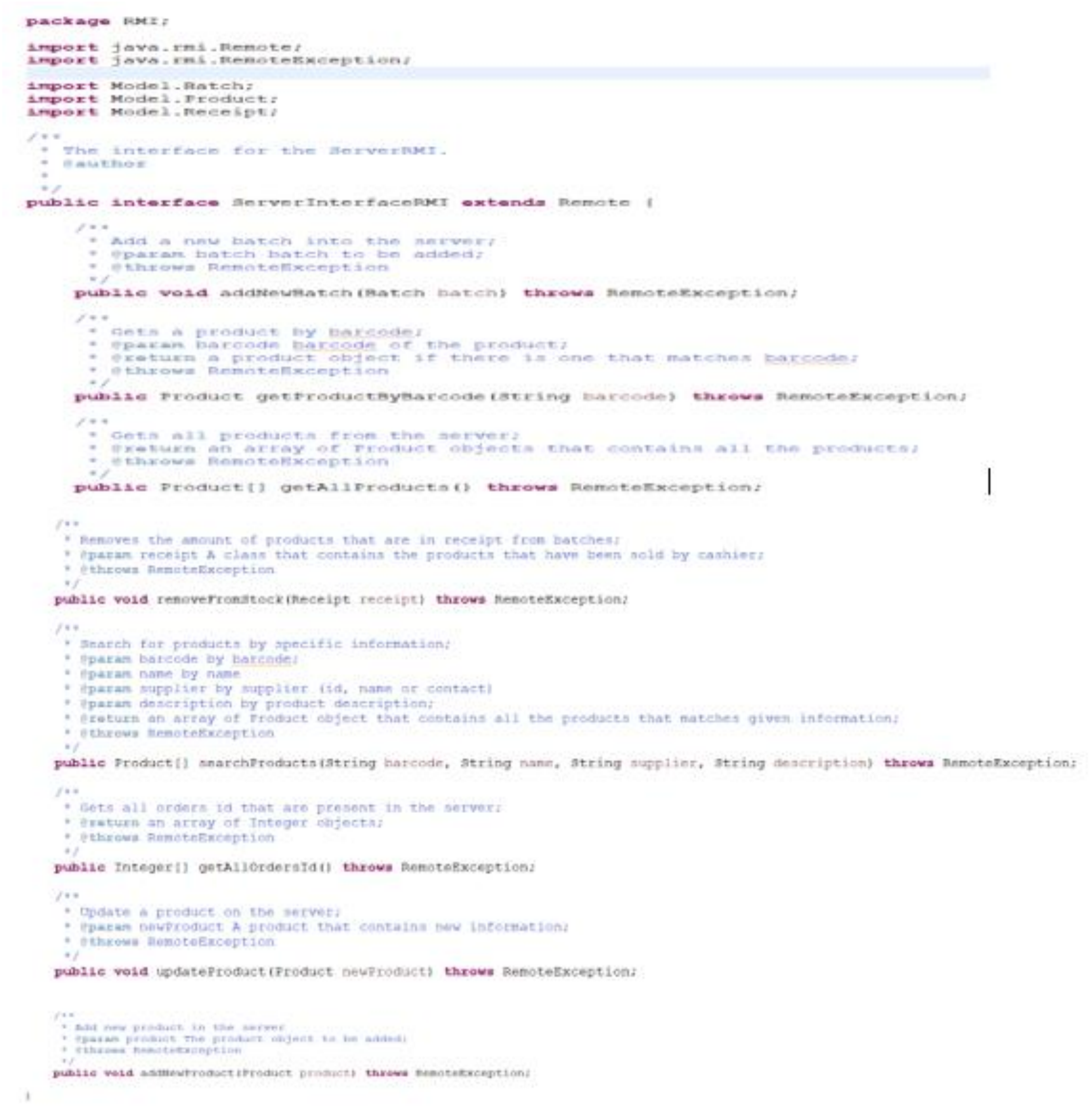
It is also possible here to edit some information about the product

Figure 18 GUI View product window

Implementation

Implementation is the phase where the ideas from analysis and design are transformed into real things. It was crucial to make sure that all the tasks are implemented in terms of importance.

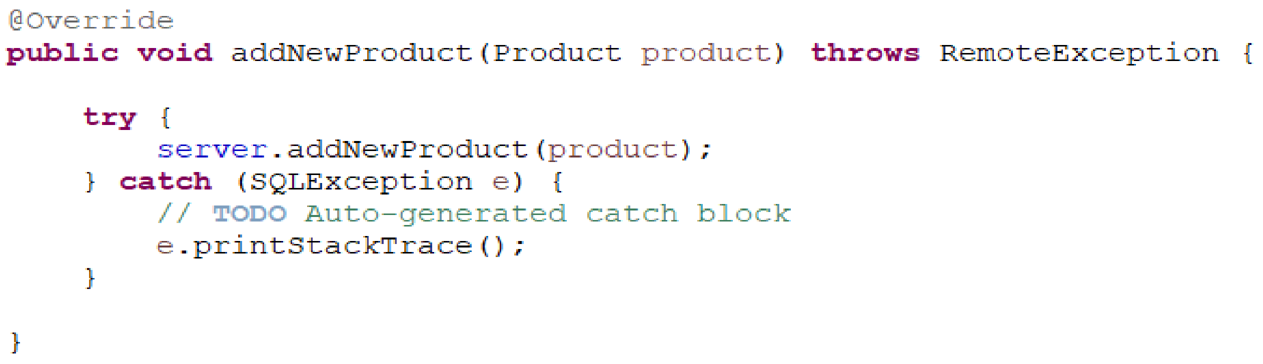
One of the core features of the system is the client-server architecture. RMI was implemented to accomplish this task. In the picture (*Figure 19)* bellow is the ServerInterfaceRMI. The interface that is used by the clients to access the methods which are on the remote server. All the method throws RemoteException.



```java
package RMI;

import java.rmi.Remote;
import java.rmi.RemoteException;

import Model.Batch;
import Model.Product;
import Model.Receipt;

/**
 * The interface for the ServerRMI.
 * @author
 *
 */
public interface ServerInterfaceRMI extends Remote {

    /**
     * Add a new batch into the server;
     * @param batch batch to be added;
     * @throws RemoteException
     */
    public void addNewBatch(Batch batch) throws RemoteException;

    /**
     * Gets a product by barcode;
     * @param barcode barcode of the product;
     * @return a product object if there is one that matches barcode;
     * @throws RemoteException
     */
    public Product getProductByBarcode(String barcode) throws RemoteException;

    /**
     * Gets all products from the server;
     * @return an array of Product objects that contains all the products;
     * @throws RemoteException
     */
    public Product[] getAllProducts() throws RemoteException;

    /**
     * Removes the amount of products that are in receipt from batches;
     * @param receipt A class that contains the products that have been sold by cashier;
     * @throws RemoteException
     */
    public void removeFromStock(Receipt receipt) throws RemoteException;

    /**
     * Search for products by specific information;
     * @param barcode by barcode;
     * @param name by name
     * @param supplier by supplier (id, name or contact)
     * @param description by product description;
     * @return an array of Product object that contains all the products that matches given information;
     * @throws RemoteException
     */
    public Product[] searchProducts(String barcode, String name, String supplier, String description) throws RemoteException;

    /**
     * Gets all orders id that are present in the server;
     * @return an array of Integer objects;
     * @throws RemoteException
     */
    public Integer[] getAllOrdersId() throws RemoteException;

    /**
     * Update a product on the server;
     * @param newProduct A product that contains new information;
     * @throws RemoteException
     */
    public void updateProduct(Product newProduct) throws RemoteException;


    /**
     * Add new product in the server
     * @param product The product object to be added;
     * @throws RemoteException
     */
    public void addNewProduct(Product product) throws RemoteException;

}
```

Figure 19 Code snippet 1: Server RMI class

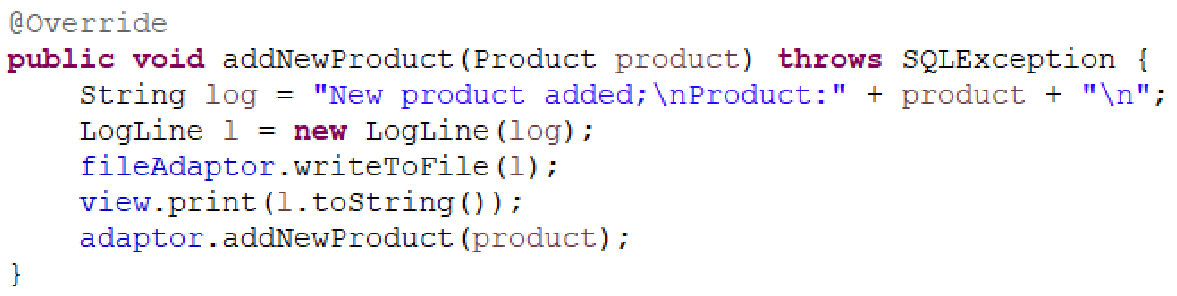
The methods from the interface are implemented by ServerRMI class. In the figure (*Figure 20)* bellow is displayed how addNewProduct() function was implemented.



```
@Override
public void addNewProduct(Product product) throws RemoteException {

    try {
        server.addNewProduct(product);
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

Figure 20 Code snippet 2: Add new product method

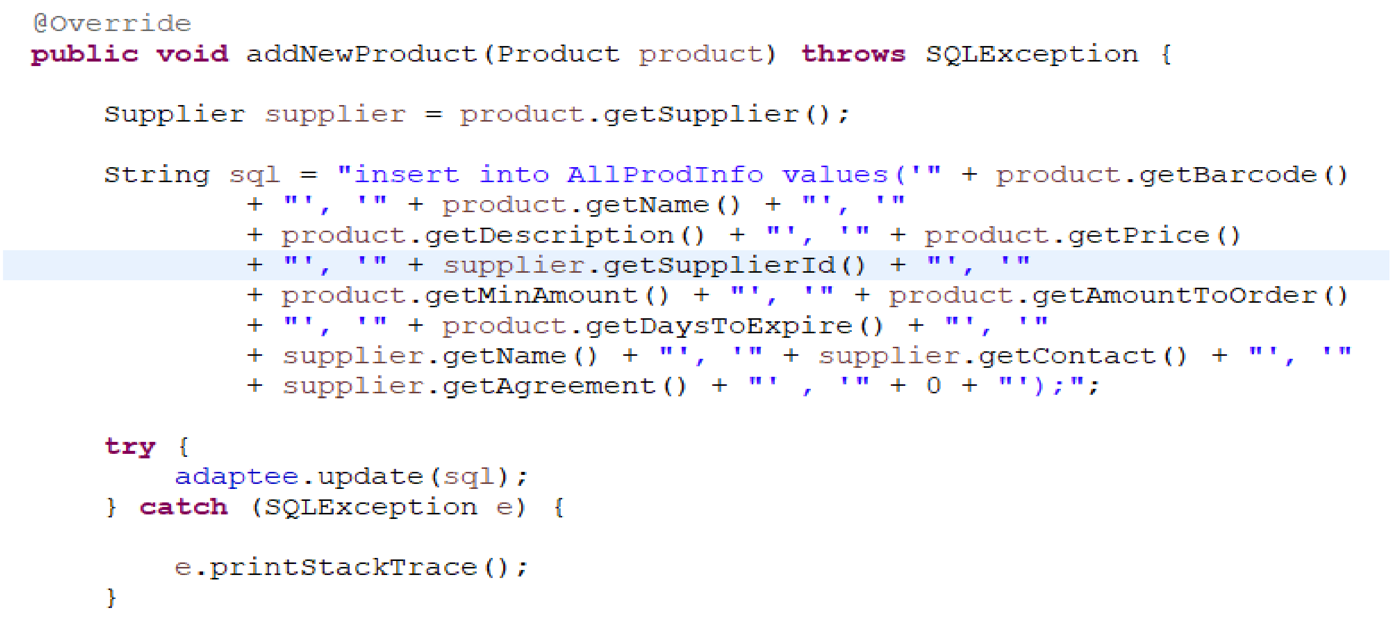
Server follows MVC design pattern. Therefore, all the functionality is performed by the controller. In the picture (*Figure 21)* bellow can be easily noticed how controller add a new product into the database.



```
@Override
public void addNewProduct(Product product) throws SQLException {
    String log = "New product added;\nProduct:" + product + "\n";
    LogLine l = new LogLine(log);
    fileAdaptor.writeToFile(l);
    view.print(l.toString());
    adaptor.addNewProduct(product);
}
```

Figure 21 Code snippet 3

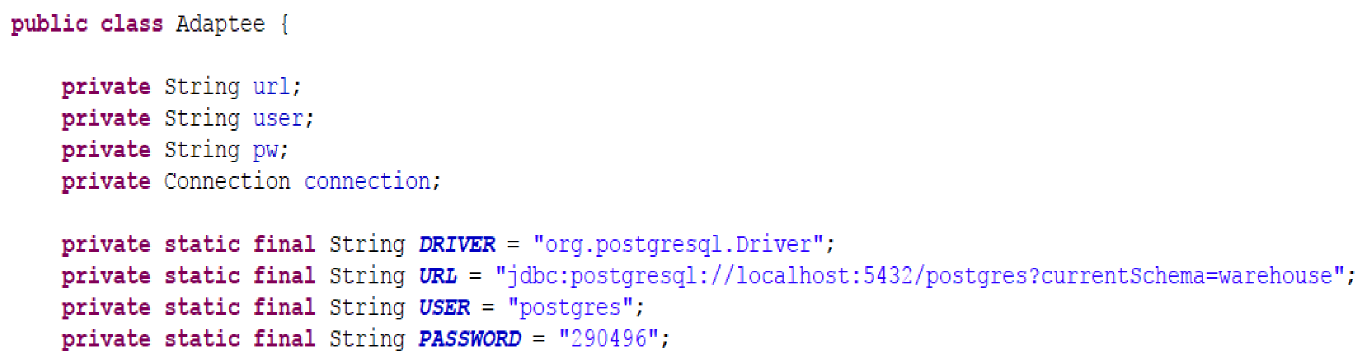
To store the object in the database, the server controller has an adaptor that translates the object into an SQL statement, so the database can execute it. In the picture (*Figure 22)* bellow is shown how it is done.



```
@Override
public void addNewProduct(Product product) throws SQLException {

    Supplier supplier = product.getSupplier();

    String sql = "insert into AllProdInfo values('" + product.getBarcode()
            + "', '" + product.getName() + "', '"
            + product.getDescription() + "', '" + product.getPrice()
            + "', '" + supplier.getSupplierId() + "', '"
            + product.getMinAmount() + "', '" + product.getAmountToOrder()
            + "', '" + product.getDaysToExpire() + "', '"
            + supplier.getName() + "', '" + supplier.getContact() + "', '"
            + supplier.getAgreement() + "' , '" + 0 + "');";

    try {
        adaptee.update(sql);
    } catch (SQLException e) {

        e.printStackTrace();
    }
```

Figure 22 Code snippet 4

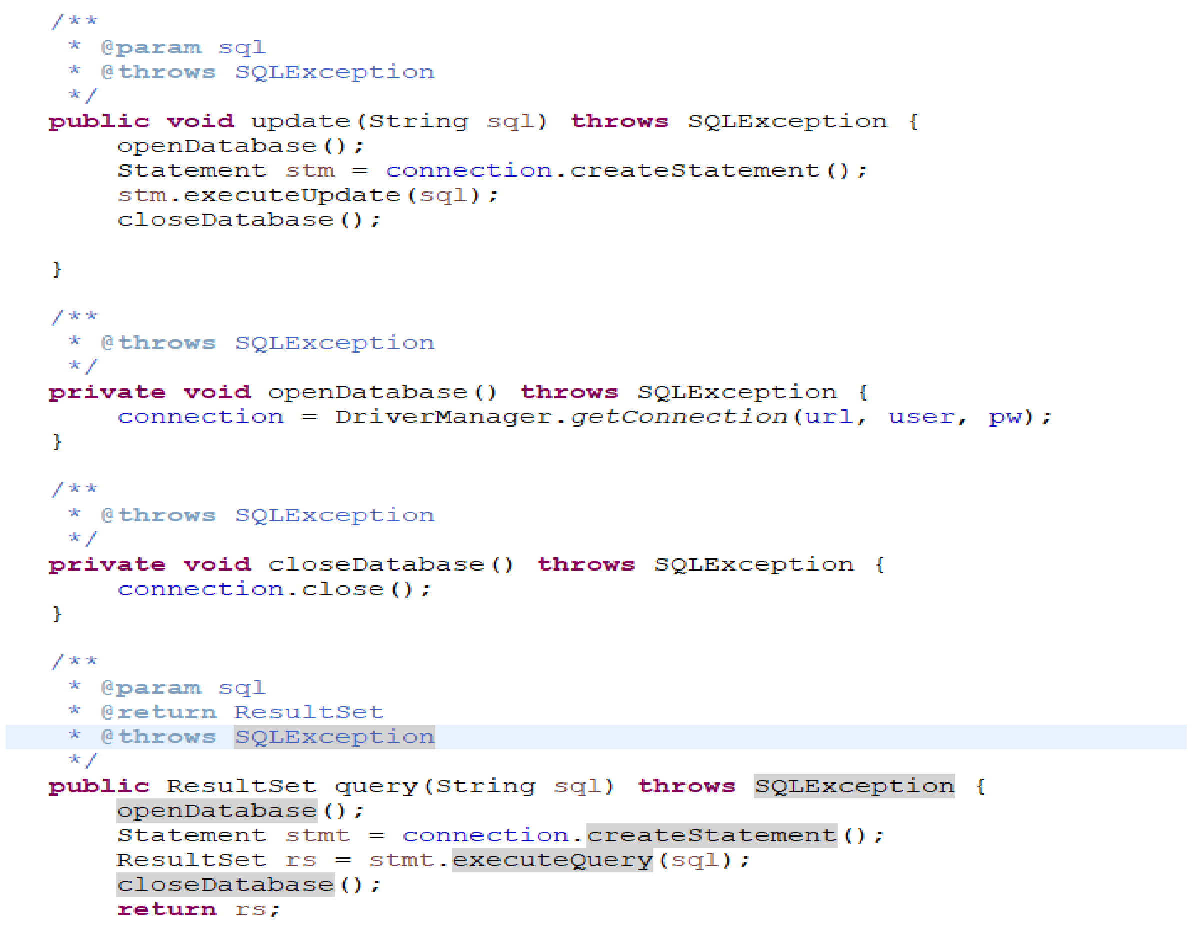
The adaptor has an Adaptee class that is responsible to take SQL statements generated by the adaptor and to run in the database. It is also responsible to store url, driver, user, and password of the database (*Figure 23).*



```
public class Adaptee {

    private String url;
    private String user;
    private String pw;
    private Connection connection;

    private static final String DRIVER = "org.postgresql.Driver";
    private static final String URL = "jdbc:postgresql://localhost:5432/postgres?currentSchema=warehouse";
    private static final String USER = "postgres";
    private static final String PASSWORD = "290496";
```

Figure 23 Code snippet 5

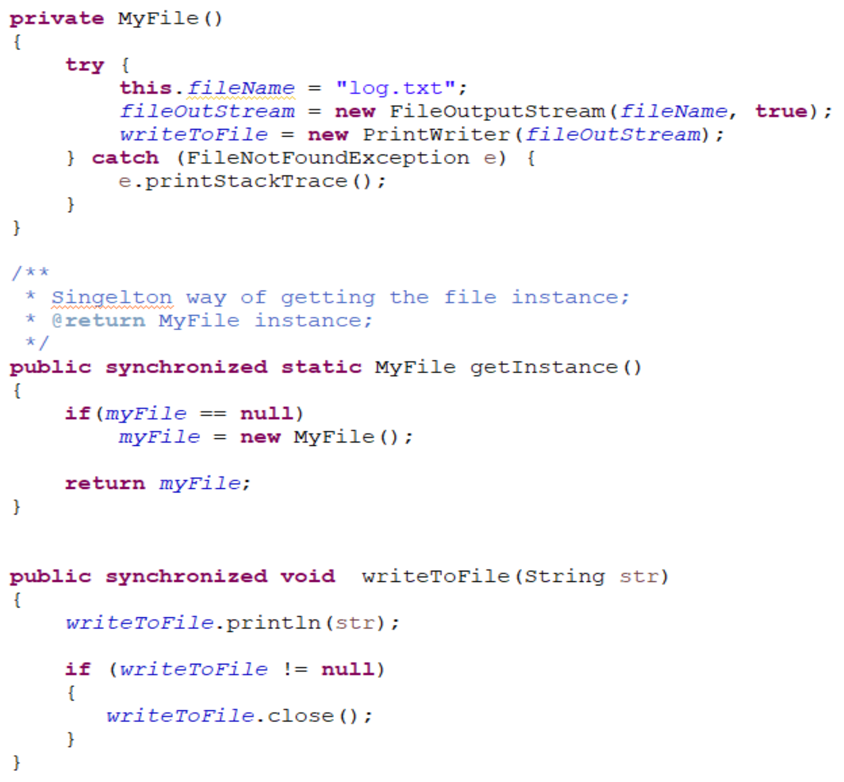
In order to run the sql statement query() and update() methods are called. If an update is required, adaptee open the database, run the sql and close the database after. If it is a query, it is doing basically the same steps, but at the end return the result in a ResultSet class. In the figure *Figure 24* bellow it is clearly seen how it was implemented.



```
/**
 * @param sql
 * @throws SQLException
 */
public void update(String sql) throws SQLException {
    openDatabase();
    Statement stm = connection.createStatement();
    stm.executeUpdate(sql);
    closeDatabase();

}

/**
 * @throws SQLException
 */
private void openDatabase() throws SQLException {
    connection = DriverManager.getConnection(url, user, pw);
}

/**
 * @throws SQLException
 */
private void closeDatabase() throws SQLException {
    connection.close();
}

/**
 * @param sql
 * @return ResultSet
 * @throws SQLException
 */
public ResultSet query(String sql) throws SQLException {
    openDatabase();
    Statement stmt = connection.createStatement();
    ResultSet rs = stmt.executeQuery(sql);
    closeDatabase();
    return rs;
```

Figure 24 Code snippet 6

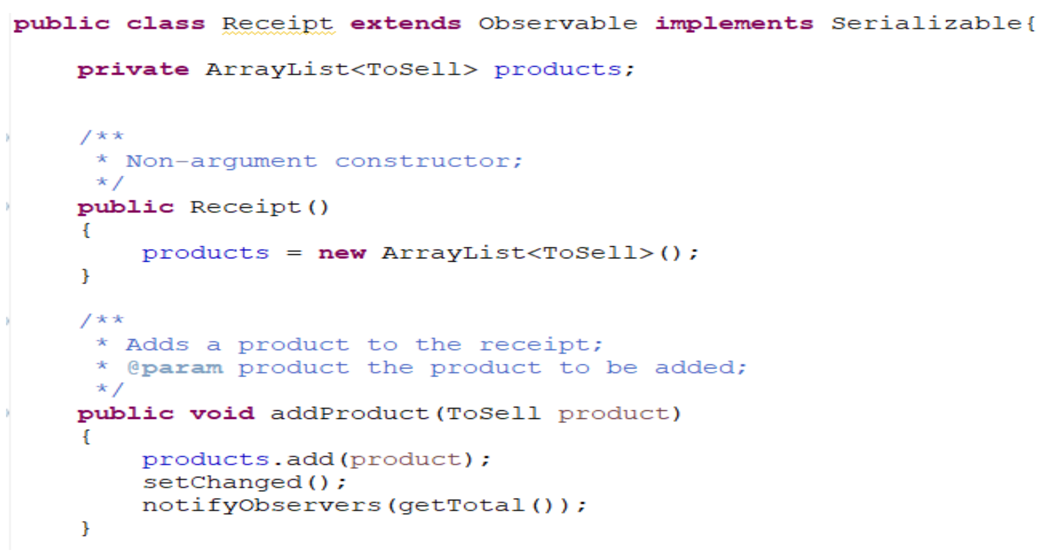
Singleton pattern (*Figure 25)* is used to write all the logs into a text file. It was the most suitable way to do this since it takes care to create an object just one time. In this way, the system doesn’t write every log in a different file but uses just one to store all of them.



```
private MyFile()
{
    try {
        this.fileName = "log.txt";
        fileOutStream = new FileOutputStream(fileName, true);
        writeToFile = new PrintWriter(fileOutStream);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}

/**
 * Singelton way of getting the file instance;
 * @return MyFile instance;
 */
public synchronized static MyFile getInstance()
{
    if(myFile == null)
        myFile = new MyFile();

    return myFile;
}


public synchronized void  writeToFile(String str)
{
    writeToFile.println(str);

    if (writeToFile != null)
    {
       writeToFile.close();
    }
}
```

Figure 25 Code snippet 7

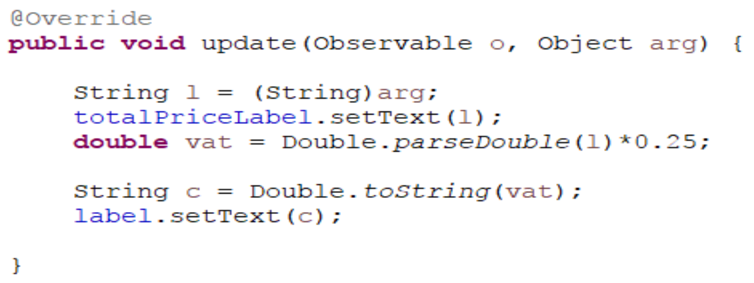
Observer design pattern was the best way to keep the view updated every time there are some changes done in the model. To implement this, the model class extends Observable class and the view implements the observer. The model notifies all observers once there are some changes. Bellow is displayed how it was implemented.



```
public class Receipt extends Observable implements Serializable{

    private ArrayList<ToSell> products;


    /**
     * Non-argument constructor;
     */
    public Receipt()
    {
        products = new ArrayList<ToSell>();
    }

    /**
     * Adds a product to the receipt;
     * @param product the product to be added;
     */
    public void addProduct(ToSell product)
    {
        products.add(product);
        setChanged();
        notifyObservers(getTotal());
    }
```

Figure 26 Code snippet 8

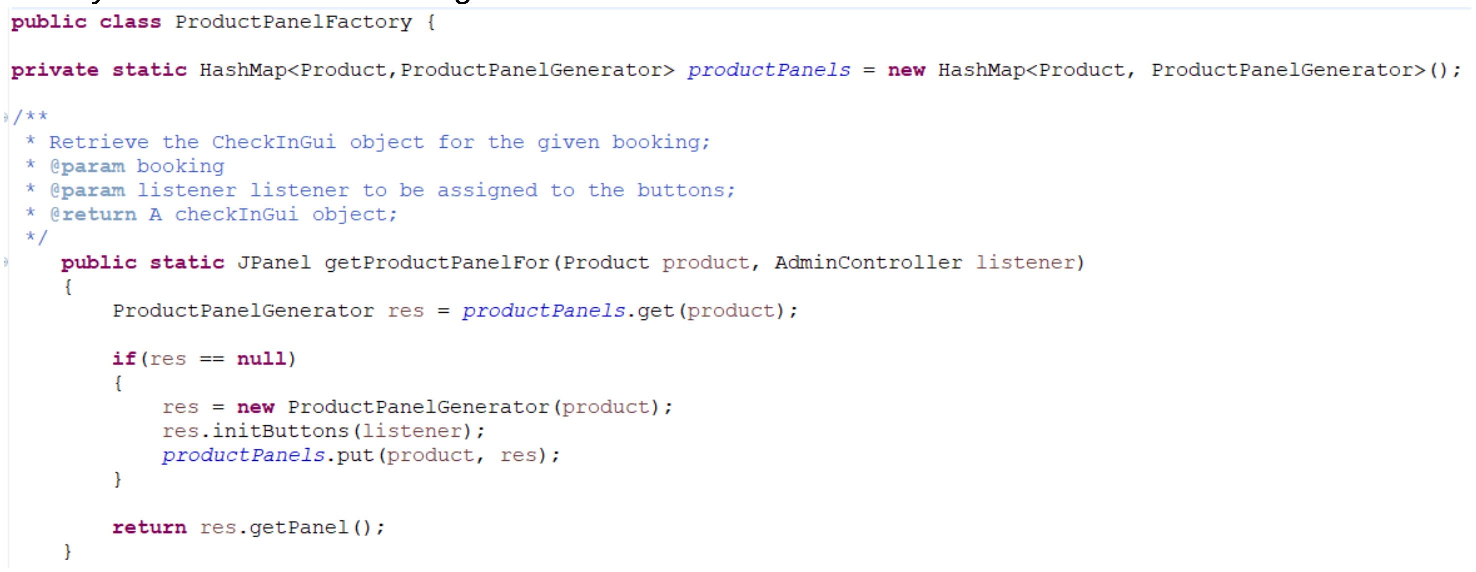
In the observer class update method is called each time there is any changes in the observable.



```java
@Override
public void update(Observable o, Object arg) {

     String l = (String)arg;
     totalPriceLabel.setText(l);
     double vat = Double.parseDouble(l)*0.25;

     String c = Double.toString(vat);
     label.setText(c);

}
```

Figure 27 Code snippet 9 Observers update method

Flyweight design pattern is implemented in the administrator part. It is used to generate panels that displays information about products. Using this design pattern was the best solution to minimize the amount of data used to run the software. To implements this, a factory class must be done *Figure 28*.

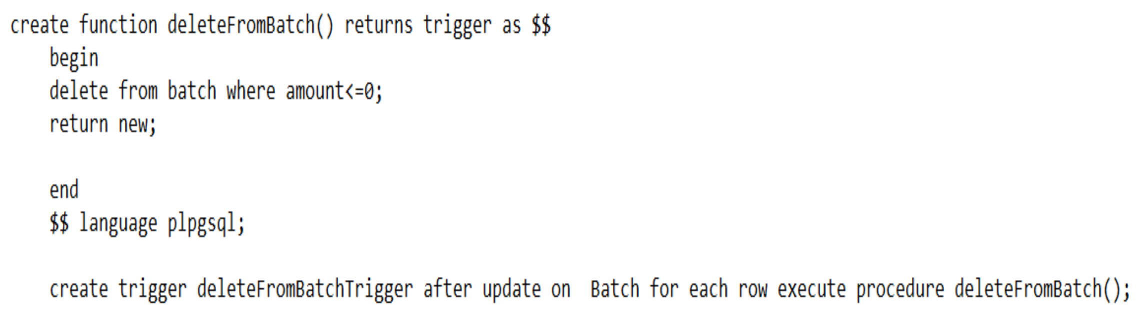


```java
public class ProductPanelFactory {

private static HashMap<Product,ProductPanelGenerator> productPanels = new HashMap<Product, ProductPanelGenerator>();

/**
 * Retrieve the CheckInGui object for the given booking;
 * @param booking
 * @param listener listener to be assigned to the buttons;
 * @return A checkInGui object;
 */
    public static JPanel getProductPanelFor(Product product, AdminController listener)
    {
        ProductPanelGenerator res = productPanels.get(product);

        if(res == null)
        {
            res = new ProductPanelGenerator(product);
            res.initButtons(listener);
            productPanels.put(product, res);
        }

        return res.getPanel();
    }
```

Figure 28 Code snippet 10. Flyweight pattern

<u>The rest of Java code you can see in Appendix “ A“</u>

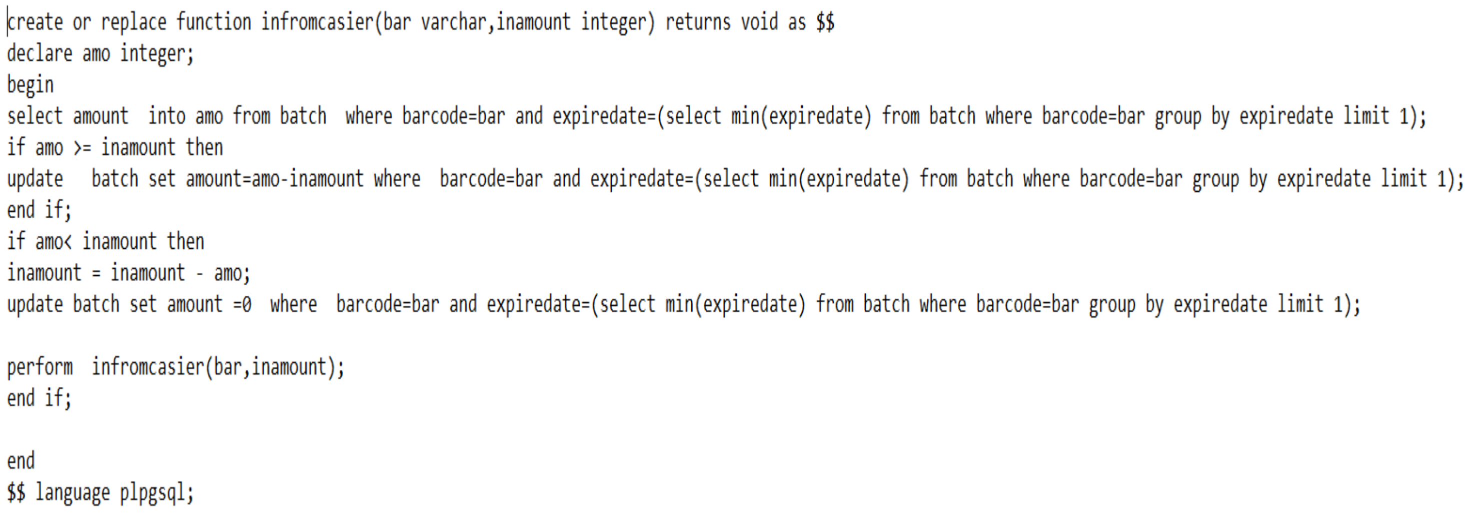
Having a good structured data base makes the system be more accurate. There are several tables to organize and store the data, so it can be easily accessed by the server. One of the most important functionalities of the data base is a trigger that takes care of deleting a batch when the amount is equal with zero. In the picture bellow is the demonstration how it was created.



```sql
create function deleteFromBatch() returns trigger as $$
    begin
    delete from batch where amount<=0;
    return new;

    end
    $$ language plpgsql;

    create trigger deleteFromBatchTrigger after update on  Batch for each row execute procedure deleteFromBatch();
```

Figure 29 Code snippet 11:Delete from batch when amount = 0

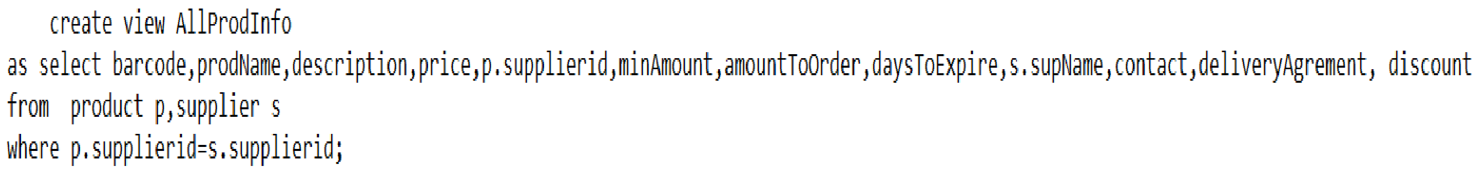
In addition, a clever idea was to implement a function that is responsible to remove the amount from batches when the products a sold. In the picture (*Figure 30)* is shown how the function is constructed.



```
create or replace function infromcasier(bar varchar,inamount integer) returns void as $$
declare amo integer;
begin
select amount  into amo from batch  where barcode=bar and expiredate=(select min(expiredate) from batch where barcode=bar group by expiredate limit 1);
if amo >= inamount then
update   batch set amount=amo-inamount where  barcode=bar and expiredate=(select min(expiredate) from batch where barcode=bar group by expiredate limit 1);
end if;
if amo< inamount then
inamount = inamount - amo;
update batch set amount =0  where  barcode=bar and expiredate=(select min(expiredate) from batch where barcode=bar group by expiredate limit 1);

perform  infromcasier(bar,inamount);
end if;

end
$$ language plpgsql;
```

Figure 30 Code snippet 12

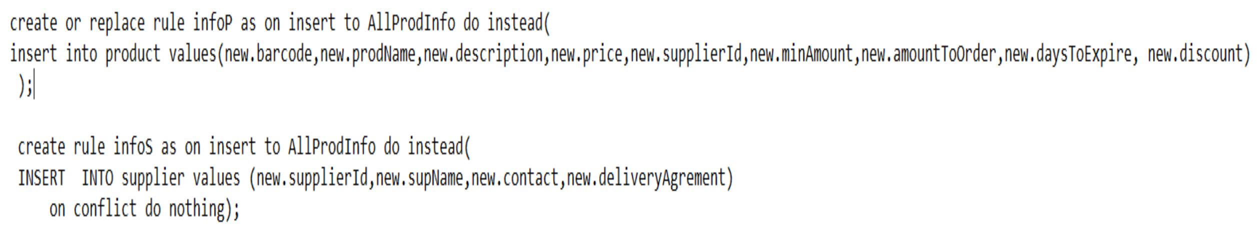
A good experience was to implement views that will combine information from different tables to be displayed to the user. Moreover, in this project the views were used the other way around. User inserts the information into a view and the system will split that information into different tables. To do that, rules were used. In (*Figure 31)* is displayed how the view was created.



```
    create view AllProdInfo
as select barcode,prodName,description,price,p.supplierid,minAmount,amountToOrder,daysToExpire,s.supName,contact,deliveryAgrement, discount
from  product p,supplier s
where p.supplierid=s.supplierid;
```

Figure 31 Code snippet 13 : AllProdInfo view

In the picture bellow is displayed how the rules were implemented.



```
create or replace rule infoP as on insert to AllProdInfo do instead(
insert into product values(new.barcode,new.prodName,new.description,new.price,new.supplierId,new.minAmount,new.amountToOrder,new.daysToExpire, new.discount)
 );

 create rule infoS as on insert to AllProdInfo do instead(
 INSERT  INTO supplier values (new.supplierId,new.supName,new.contact,new.deliveryAgrement)
     on conflict do nothing);
```

Figure 32 Code snippet 14

<u>The rest of PostgreSQL implementation you can see in Appendix “ B“</u>

Testing and results

Test Documentation

1) See all products
   - Pressing "All products" button opens the page with all items.
   - Pressing "All products" opens a new page
   - Pressing "View" it opens a window with the information about item
   - Pressing "Remove" it will remove the item from the system

   Expectation: All products are displayed

   Error: None
2) Add products
   - Pressing "Add products" opens the page with all necessary fields to be completed to add a new product into the system
   - In the Barcode text field, the user inserts the barcode of the product the user is looking for
   - Press "Find" button
   - If the barcode of the product is not found the system pops up with the message "There is no product with this barcode. Do you want to add a new product?"
   - If the user press "yes" then all fields should be completed.
     Expectation: The product it's added in the system
     Error: System also saves any kind of characters in some of text fields without checking if is a valid input in some other text fields invalid inputs is crashing the system
     If the amount field is taking letters from the input then the system crashes
3) Search item
   - Enter the Barcode or / and
   - Enter the Name and /or
   - Enter the Supplier and/or
   - Enter the Description
   - Press "Search Button"
     Expectation: The searched product is found
     Error: None
4) Order
   - Pressing the "Order" button is displaying the new page
   - Enter the barcode
   - pressing the "add product"
   - The product is added
   - After adding the all needed products, the user should press the "place order button"

Expectation1: The found product by Barcode is found and added in the system

Expectation2: The order is placed

Error: None

Results

1	As an administrator I want to access from multiple computers	Working
2	As an administrator I would like to insert new items into the systems	Working
3	As an administrator I would like to see all the items	Working
4	As an administrator I would like to search for a particular product by barcode, name, supplier,key words	Working
5	As a cashier I would like to insert:Barcode of items and the number of products to be sold	Working
6	As an administrator I would like to be able to order items	Not working
7	As an administrator I would like to edit product information	Working
8	As an administrator I would like to remove an item from the list	Working
9	As a cashier I would like to remove an item from the invoice	Working
10	As an administrator I would like to remove a list of items	Not working
11	As a cashier I would like the system to create automatically an invoice of total price when the byer is done	Not working
12	As an administrator I would like to be notifed when items are close to expired	Not working

Figure 33 Results

Conclusion

The goal of this project is to offer shop owners a simple, low cost solution for inventory management. It started with a list of requirements gathered from clients. The implementation doesn't include all that was requested in the beginning. Due to the limited amount of time to complete this project some features were left aside, which, by themselves, can create the subject of a future development project of the system.

The features were implemented in terms of priority. Overall, the most important ones were implemented during this project, so the system reached its core goal. Adding a new product, searching through the database for a product and/or supplier information, editing and keeping an updated stock info are features that SIMS includes.

Features left aside were useful but not necessarily a must for the system purpose. In fact, having the requirement ''remove an item from the list'' implemented makes requirements like ''remove a list of items'' seem more like a ''user friendly'' feature than a functionality related feature.

Much of the important ''work'' of the system is done automatically. SIMS is keeping track of the stock, counting products, deleting empty batches and creating new orders when the stock is low automatically.

Overall SIMS is a helpful assistant and it takes much of the responsibility of the warehouse manager 's shoulders. Having that said, the team that created SIMS thinks that the project is a success and reaches it goal.

References

Connolly, T.M. and Begg, C.E., 2010. *Database systems : a practical approach to design, implementation, and management*. 5th ed. Harlow, Essex, England: Addison-Wesley.

Gaddis, T., 2015. *Starting out with Java : early objects*. Fifth5. ed ed. Upper Saddle River, NJ: Pearson.

Kurose, J.F. and Ross, K.W., 2010. *Computer networking : a top-down approach*. 5th ed. Boston: Addison-Wesley.

Larman, C., 2005. *Applying UML and patterns : an introduction to object-oriented analysis and design and iterative development*. 3rd ed. Upper Saddle River, N.J.: Prentice Hall Professional Technical Reference.

Appendices