# Analysis and Design Document

for

Software Design – Assignment 1
WasteLess

Micheș Mihnea Bogdan
Group 34031

## Table of Contents

# 1. Assignment Request

Design and implement an application that helps users manage food waste.
Once a user is authenticated he can input grocery lists and see reports of how much food is wasted weekly and monthly. A grocery list item has a name and a quantity as well as a calorie value, purchase date, expiration date and consumption date.
The system also allows users to track goals and minimize waste by sending reminders if waste levels are too high based on ideal burndown rates.
The ideal burndown rate for 100 calories worth of groceries due to expire in 5 days is 20 calories worth of groceries per day.
The system should provide you with options to donate excess food to various local food charities and soup kitchens and notify you of them prior to item expiration.

**Requirements**

● Implement and test the application

● Commit the work you do on your Git repository. Do it iteratively as you progress, not all at once (this will incur a penalty on your final mark)

- ● Use any OOP language you like. Non-exhaustive: Python, C#, Java, Ruby, C/C++, JS+Typescript
- ● Use a client-server architecture
- ● Use an observer for sending notifications to users about donation options when item expiration is due
- ● The data will be stored in a database
- ● All the inputs of the application will be validated against invalid data before submitting the data and saving it in the database.

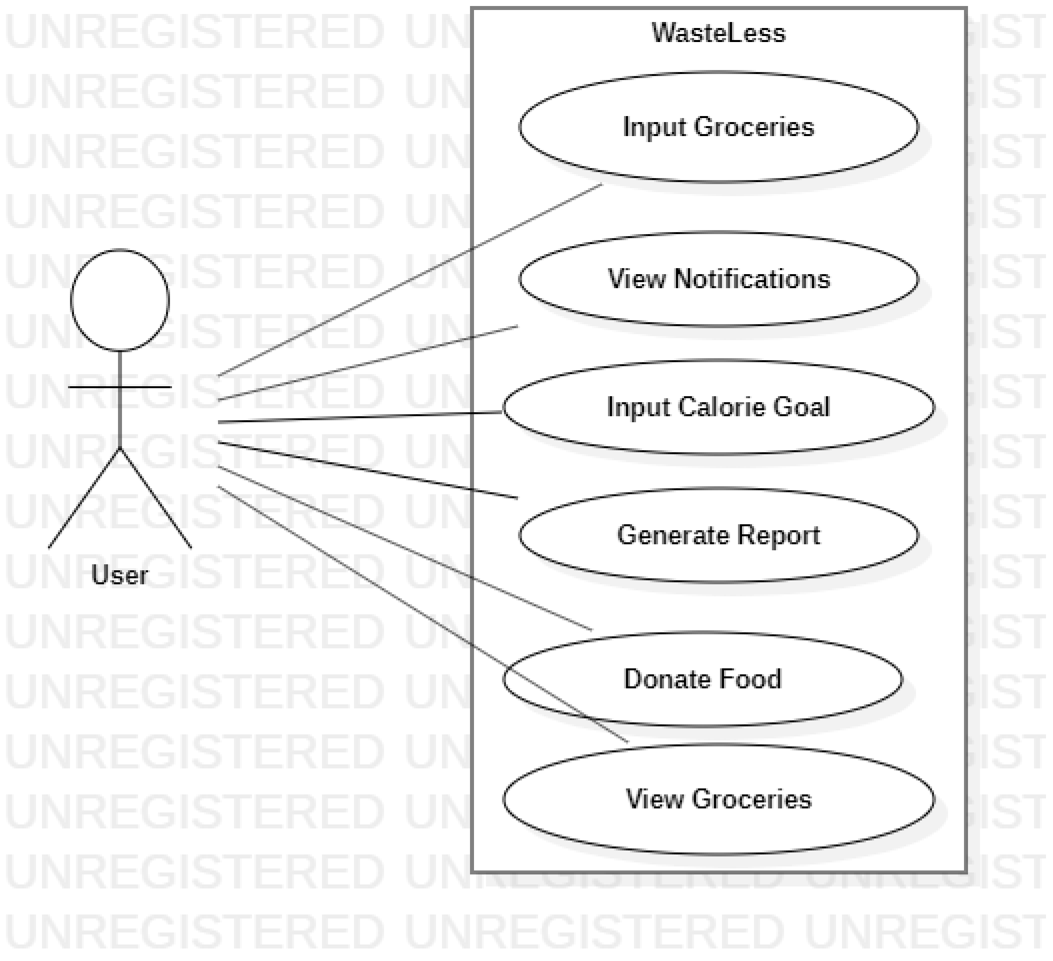
The main goal of this application is to minimize food waste. To do that, the input we need is the grocery lists and the items on them. Then, we calculate their ideal burndown rates, add them up and then see whether there is the possibility of wasting food based on the user's entered calorie target. Then we have to notify the user about the waste, and offer options to get rid of it.

Broken down into simpler tasks, we have to:

- Gather input from the user: grocery lists and calorie target
- Store the groceries in the database (a very simple database though)
- Compute the ideal burndown rates, and check whether there is a possibility of having waste
- Notify the user accordingly
- Allow the user to donate groceries (i.e. delete from database)
- Generate reports

# 2. Problem Analysis

## Use Cases



Use Case: Input Groceries

- Description: User inputs a grocery list and the system stores it in the database
- Actor: User
- Pre-condition: The database + server connection must exist.
- Post-condition: The items entered by the user are stored in the database.
- Main Success Scenario: User clicks the button corresponding to adding a new grocery lists, enters the required data for each item (such as name, quantity,..) and then submits the input.
- Extensions: In case of bad input, the user is notified of this and asked to retry.

Use Case: View Notifications

- Description: A new window appears, showing all the user's notifications.
- Actor: User

- Pre-condition: The database + server connection must exist.
- Post-condition: A table presenting the notifications is displayed.
- Main Success Scenario: User clicks the button corresponding to viewing their notifications. The system then checks for new notifications and, in case they exists, adds them to the database, and then queries the database in order to display all the notifications.
- Extensions: none.

Use Case: Input Calorie Goal

- Description: User changes the calorie goal (Default: 2000 calories)
- Actor: User
- Pre-condition: The new value must be valid (> 0).
- Post-condition: The system's calorie goal is updated.
- Main Success Scenario: User clicks the button corresponding to updating their calorie goal. A new window pops up, asking for input. After receiving this input, the system updates and refreshes the statistics regarding food waste.
- Extensions: In case of bad input, the user is notified, and asked to retry.

Use Case: Generate Report

- Description: User asks the system to generate a .txt file with the weekly / monthly waste report.
- Actor: User
- Pre-condition: A connection to the database + server must exist.
- Post-condition: A .txt file is generated with the report.
- Main Success Scenario: User clicks the button corresponding to generating a new report. The system then queries the database in order to generate a .txt file with the waste statistics.
- Extensions: none.

Use Case: Donate Food

- Description: User asks the system to donate the food that has 2 days left until expiration.
- Actor: User
- Pre-condition: A connection to the database + server must exist.
- Post-condition: Database entries corresponding to the items that are about to expire are deleted.
- Main Success Scenario: User clicks the button corresponding to donating the food. The system then displays the items that will be donated, asks for confirmation, and then deletes these items from the database.

- Extensions: none.
-

Use Case: View Groceries

- Description: A new window appears, showing all the user’s owned groceries.
- Actor: User
- Pre-condition: The database + server connection must exist.
- Post-condition: A table presenting the user’s groceries is displayed.
- Main Success Scenario: User clicks the button corresponding to viewing their groceries. The system then queries the database in order to display all the owned grocery items.
- Extensions: none.

Use scenarios are very basic: the user can input grocery lists, view their groceries, view notifications, generate reports or donate food. Each of these scenarios only requires a click from the user, except for the first which requires some data input.

Scenario: Input a new grocery list

Steps:

- User clicks the button corresponding to adding a new grocery list
- A new window pops up, asking for information about a grocery list item
- The user fills in the data, and then decides whether the list is done, or more items are required. In the latter case, the user clicks the button corresponding to entering the next item’s data and repeats this process until the list is done.
- The user the clicks the button corresponding to finishing entering data into the application, and then the grocery list is stored in the database.

As said above, the other use scenarios require only one click, and the system will handle the rest. The system should be able to fulfill these functions:

- Read and write from / to the database
- Determine the ideal burndown rates for the items
- Decide whether food waste is possible based on these rates and if so, generate a notification for the user
- Calculate the food waste statistics
- Generate a .txt report containing these statistics

# 3. Design

## 3.1. Design Decisions

I assumed that the food waste was determined by the difference between the ideal burndown rates for the groceries and the user's calorie goal. This was not mentioned in the requirements but it made the most sense to me.

The application will be developed using the Java OO language, in the Eclipse IDE.

The GUI of the application will be built using the WindowBuilder plugin for Eclipse IDE.

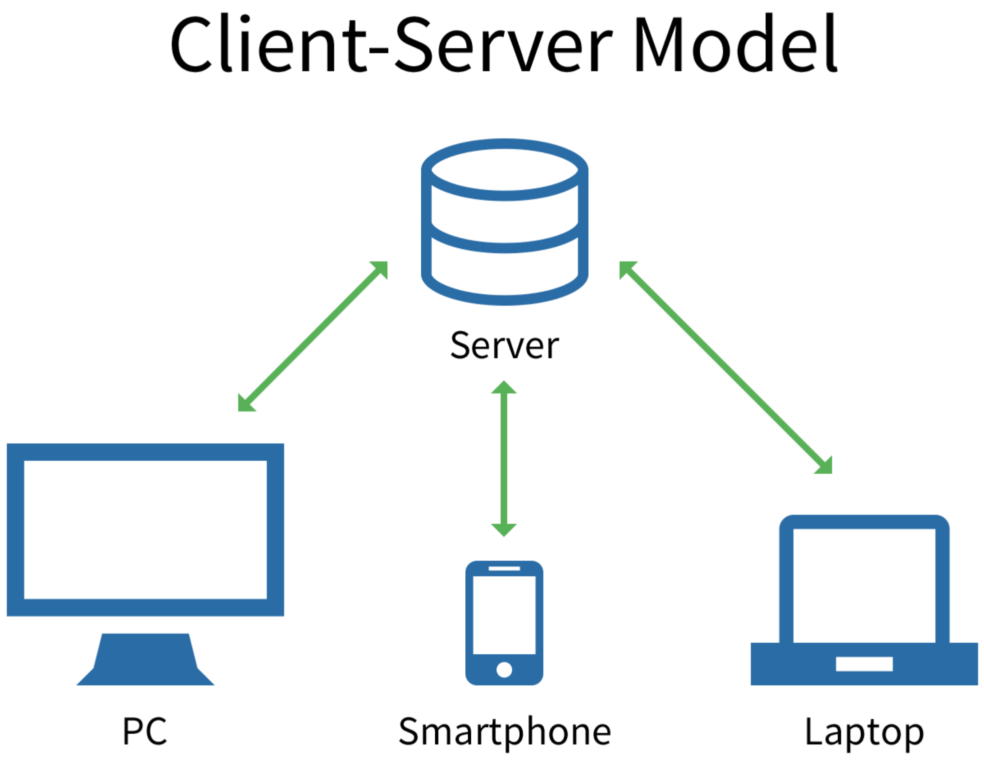
For the Dependency Injection, we will use Spring Boot (with Maven).

For the Object Relational Mapping, we will use Hibernate.

The database will be created using MySQL Workbench, and the connection between database and application server will be handled by Hibernate.

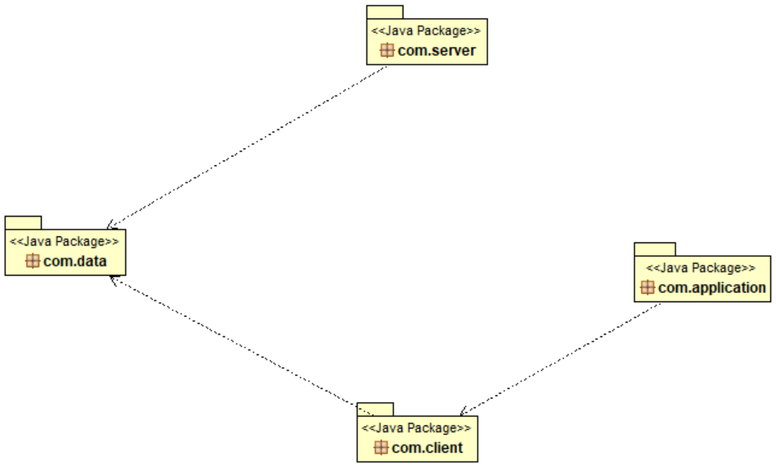
This application is built using a client – server architecture. This means that our application is split in two: the client side, and the server side. For this application, we will consider the client's side to consist only of the User Interface and the PDF Writer. The server will handle the business logic and the database accesses.

The client and server will communicate through sockets (this communication is bi-directional). The client will request data from the server, and the server will respond with said data. Note that the client has no knowledge of the Data package, as it only works with Strings in order to perform all its duties. This way, the server's Data package can change without influencing the client.



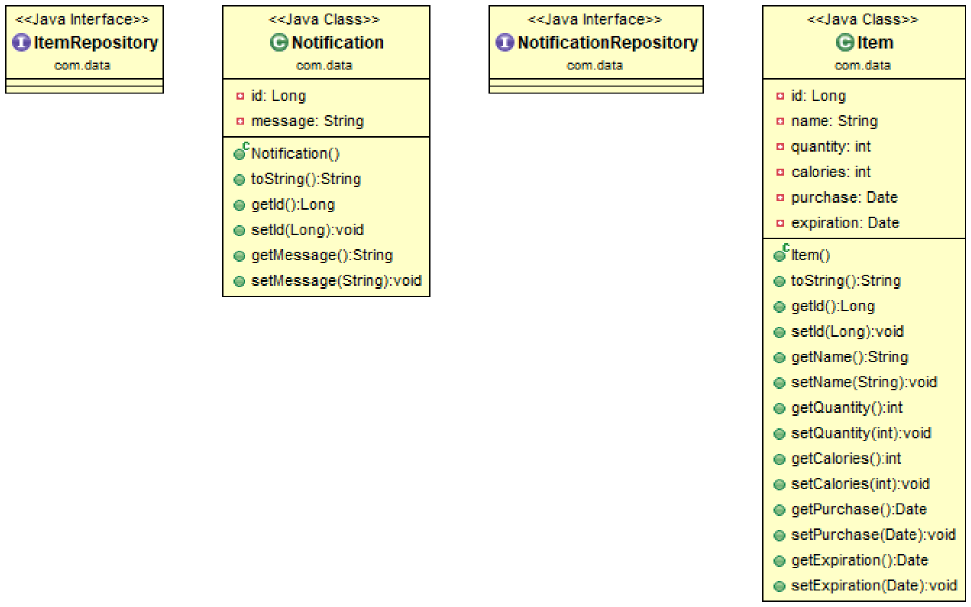
## 3.2. Diagrams

Package diagram:

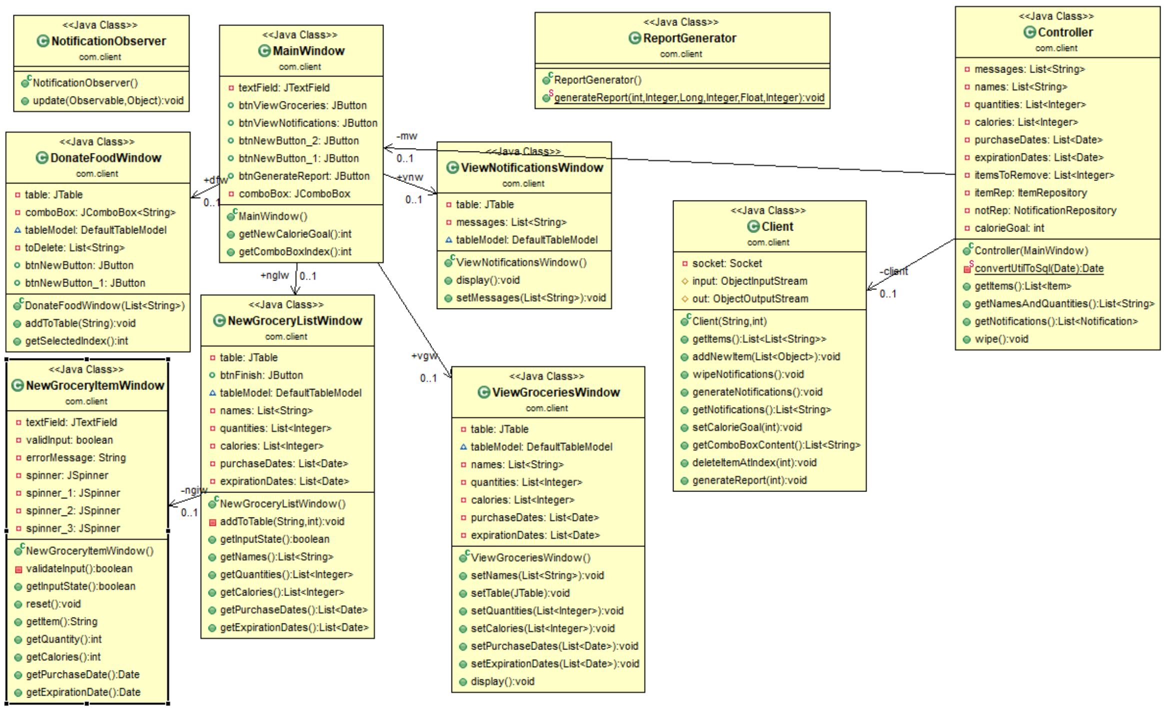


## Class diagrams:

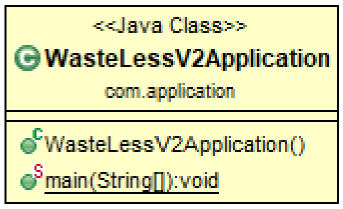
### Package Data:



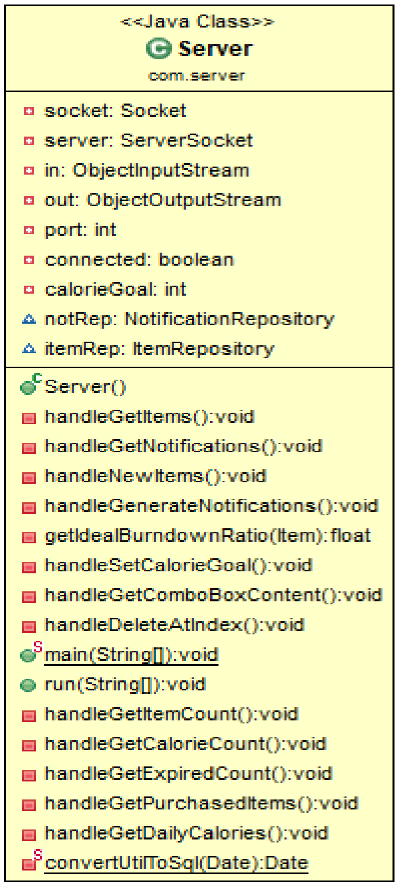
## Package Business:



## Package Presentation:



## Package Server:



<<Java Class>>
Server
com.server
socket: Socket
server: ServerSocket
in: ObjectInputStream
out: ObjectOutputStream
port: int
connected: boolean
calorieGoal: int
notRep: NotificationRepository
itemRep: ItemRepository
Server()
handleGetItems():void
handleGetNotifications():void
handleNewItems():void
handleGenerateNotifications():void
getIdealBurndownRatio(Item):float
handleSetCalorieGoal():void
handleGetComboBoxContent():void
handleDeleteAtIndex():void
main(String[]):void
run(String[]):void
handleGetItemCount():void
handleGetCalorieCount():void
handleGetExpiredCount():void
handleGetPurchasedItems():void
handleGetDailyCalories():void
convertUtilToSql(Date):Date

## 3.3. Handling the logic

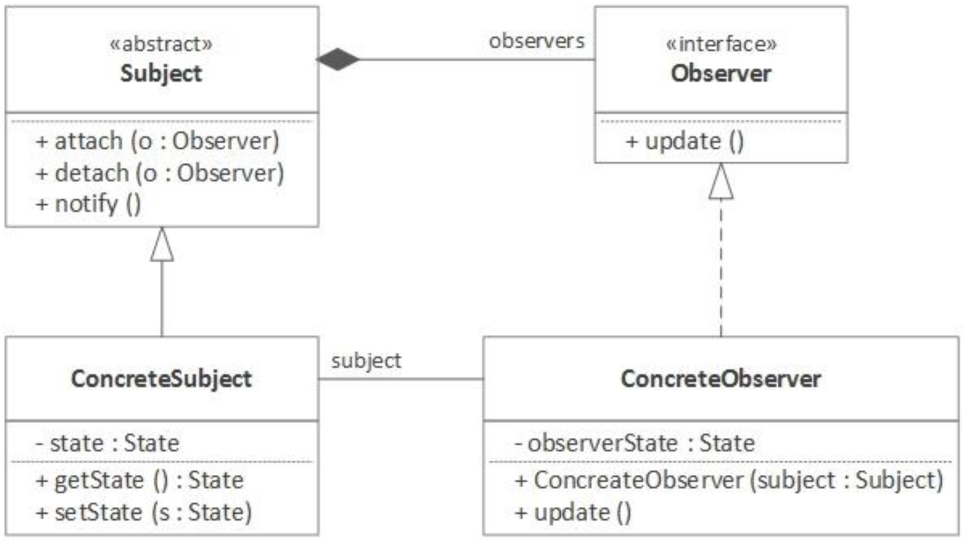
There are a few computations that have to be performed by the application. These are handled by the Server class.

One such computation is calculating the ideal burndown ratio for items. That number is equal to the item’s calorie count divided by the number of days left until expiration (and multiplied by quantity), and that is the number the item contributes to the daily caloric intake. The waste is then this value minus the user’s daily caloric goal.

Other computations are calculating differences between dates.

Data structures used in the applications are only ArrayList(s).

We also use the Observer design pattern:



public class Controller extends Observable

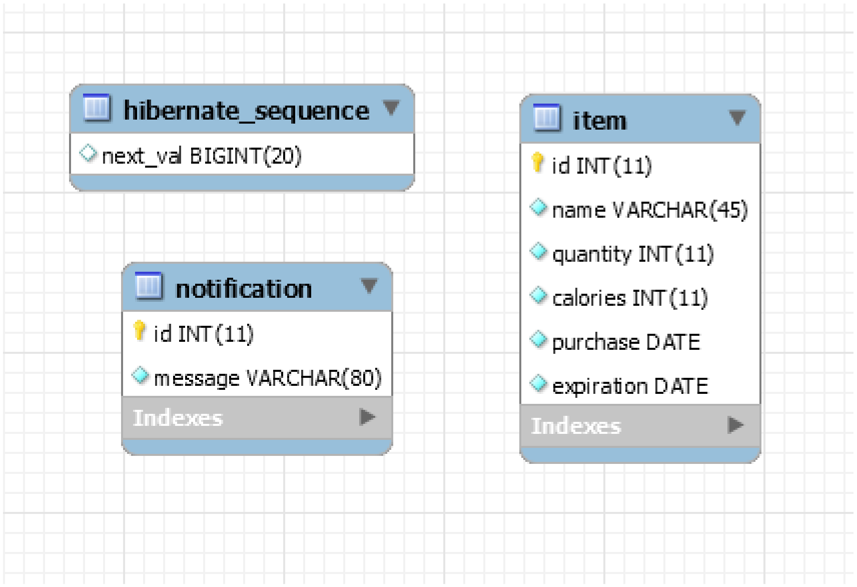
public class Client implements Observer

If a new grocery list is added, the Client is notified by the Controller and requests the generation of new notifications to the Server.

The reports are generated by the client-side ReportGenerator class.

### 3.4. Database

Diagram:



No relationships between tables are required.

# 4. Implementation

The application is split into 3 packages: Data, Business and Presentation.

## 4.1. Package com.data

This package contains the data access classes. The classes contained here are:

- Item
- ItemRepository
- Notification
- NotificationRepository

Since we used Hibernate as the ORM tool, database access is done through repositories (which are Autowired here).

```
@Repository
public interface ItemRepository extends JpaRepository<Item, Long> {

}
```

The database only stores the objects Item and Notification. The rest of the classes correspond to generating reports. They are not stored in the database.

## 4.2. Package com.application

This package contains the following classes:

- WasteLessApplication

This is the client application's driver class, having the main() method.

```
public static void main(String[] args) {

        WebLookAndFeel.install(WebDarkSkin.class);

        MainWindow mw = new MainWindow();

        Controller controller = new Controller(mw);

    }
```

## 4.3. Package com.client

This package contains the UI classes and has no logic. Windows are spawned here and the logic is handled by the com.business package. It contains the following classes:

- Client
- Controller
- DonateFoodWindow
- MainWindow
- NewGroceryItemWindow
- NewGroceryListWindow
- ReportGenerator
- ViewGroceriesWindow
- ViewNotificationsWindow

The home page is class MainWindow.

```
public MainWindow() {
		setBounds(new Rectangle(450, 300, 0, 0));
		this.setSize(450, 435);
		
		getContentPane().setLayout(null);
		
		nglw = new NewGroceryListWindow();
		vgw = new ViewGroceriesWindow();
		vnw = new ViewNotificationsWindow();
		
		JButton btnNewGroceryList = new JButton("New Grocery List");
		btnNewGroceryList.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				nglw.setVisible(true);
				nglw.addWindowListener(new WindowAdapter(){
				   public void windowClosing(WindowEvent e){
				   }
				});
			 }
		});
```

…

Class Client handles the communication with the Server:

```
public Client(String address, int port)

{

        // establish a connection

        try

        {

            socket = new Socket(address, port);

            System.out.println("Connected on port " +
socket.getPort());

            out    = new ObjectOutputStream(socket.getOutputStream());

            input  = new ObjectInputStream(socket.getInputStream());



        }

        catch(

                UnknownHostException u)
        {
            System.out.println(u);
        }
        catch(IOException i)
        {
            System.out.println(i);
        }


    }
public void addNewItem (List <Object> newData) throws IOException {
         out.writeObject(new String("newItem"));
         out.writeObject(newData);
    }
```

The Controller class adds listeners to the UI.

## 4.4. Package com.server

This package contains only one class:

- Server

It is supposed to run on the server side of the application. It handles all the logic, and only sends Strings and numbers to the Client.

```
server = new ServerSocket(port);

            System.out.println("Server started");
            System.out.println("Waiting for a client ...");

            socket = server.accept();

            System.out.println("Client accepted");

            out = new ObjectOutputStream(socket.getOutputStream());
            in = new ObjectInputStream(socket.getInputStream());

private void handleGetItems() throws IOException{
    List <List <String>> details = new ArrayList <List <String>> ();
    List <Item> items = itemRep.findAll();

    Iterator <Item> it = items.iterator();

    while(it.hasNext()) {
        List <String> newList = new ArrayList <String> ();
        Item currentItem = it.next();
        newList.add(currentItem.getName());
        newList.add(Integer.toString(currentItem.getQuantity()));
        newList.add(Integer.toString(currentItem.getCalories()));

newList.add(Long.toString(currentItem.getPurchase().getTime()));

newList.add(Long.toString(currentItem.getExpiration().getTime()));
        details.add(newList);
    }

    out.writeObject(details);
```

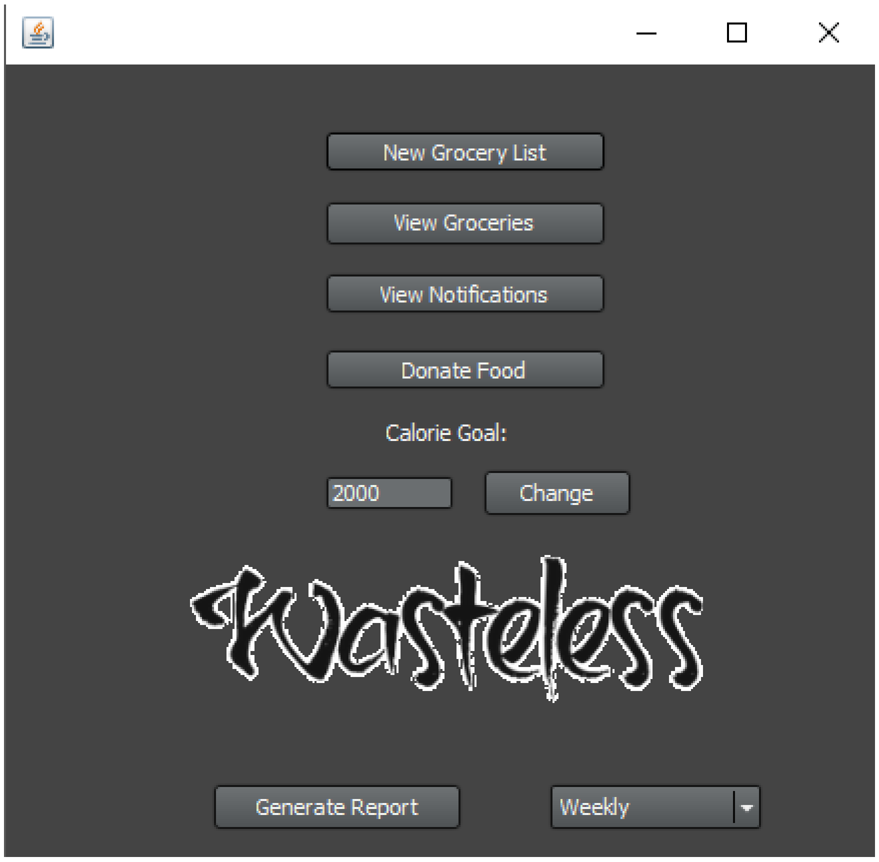
}

This way, the application logic can very easily change, without affecting the client side at all.

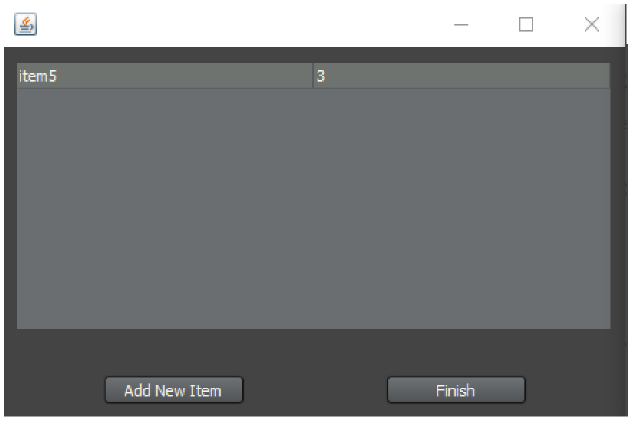
## 4.5. User Interface

The look and feel used here is WebLaF.

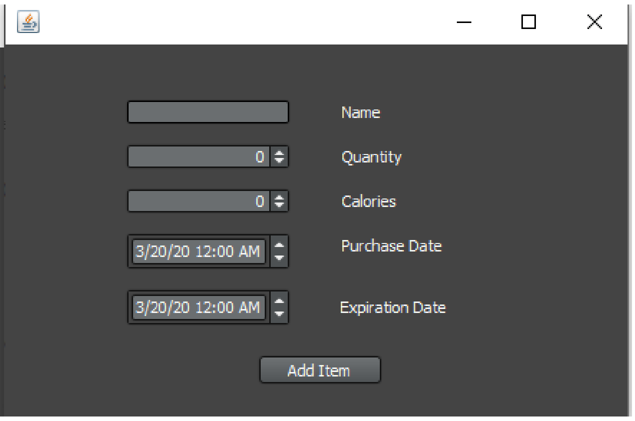
Home window:



Adding new grocery items:

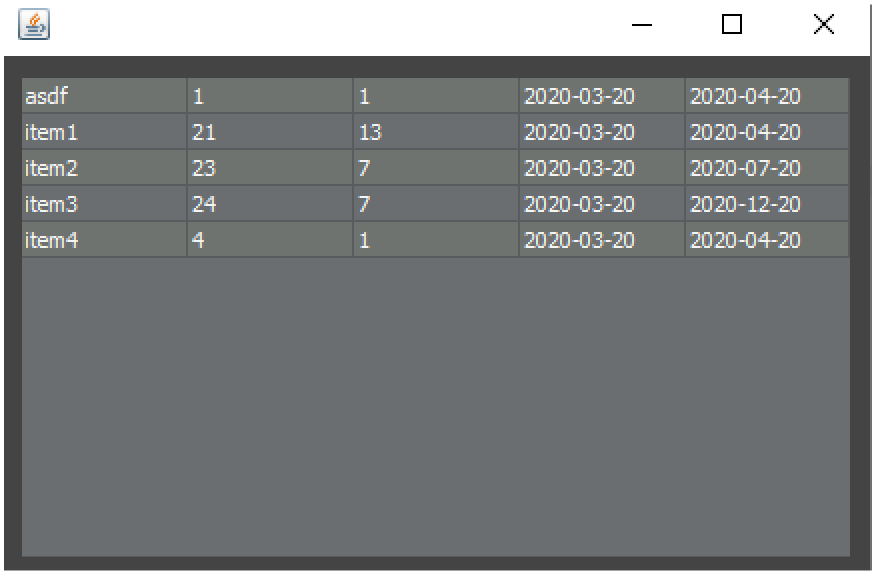


| | |
|---|---|
| item5 | 3 |



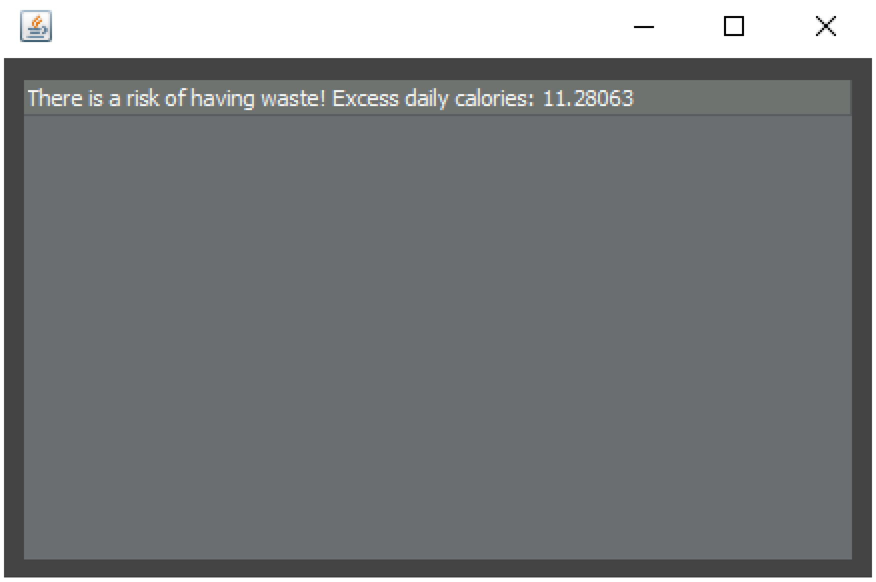
| | |
|---|---|
| | Name |
| 0 | Quantity |
| 0 | Calories |
| 3/20/20 12:00 AM | Purchase Date |
| 3/20/20 12:00 AM | Expiration Date |

## Viewing owned items:



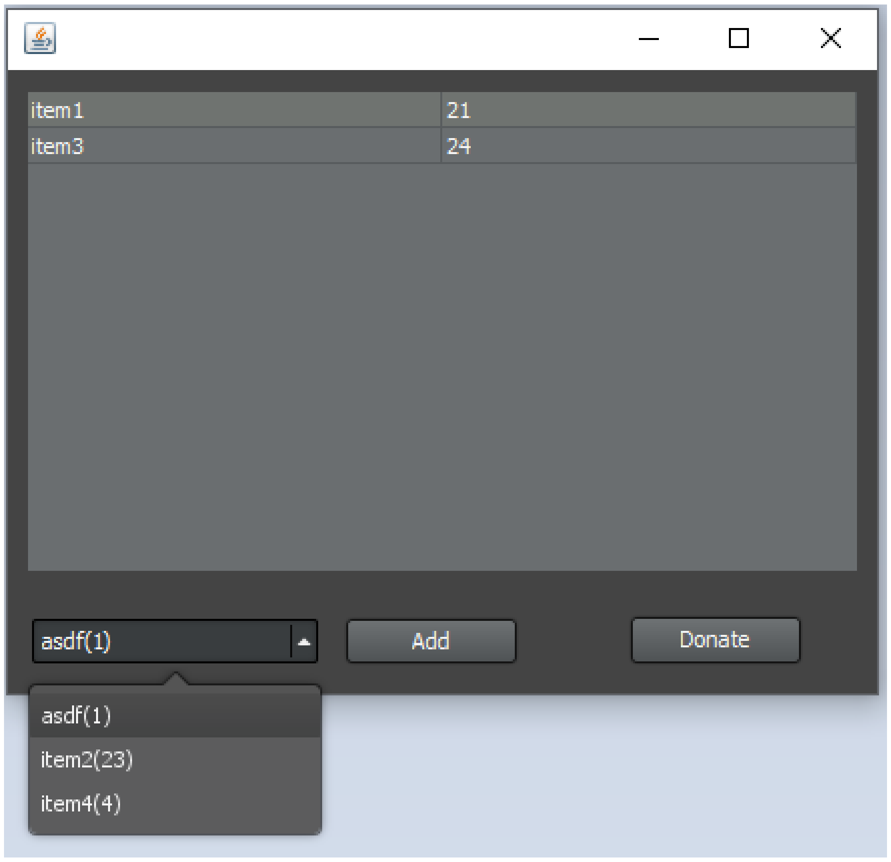
| | | | | |
|---|---|---|---|---|
| asdf | 1 | 1 | 2020-03-20 | 2020-04-20 |
| item1 | 21 | 13 | 2020-03-20 | 2020-04-20 |
| item2 | 23 | 7 | 2020-03-20 | 2020-07-20 |
| item3 | 24 | 7 | 2020-03-20 | 2020-12-20 |
| item4 | 4 | 1 | 2020-03-20 | 2020-04-20 |

## Viewing notifications:



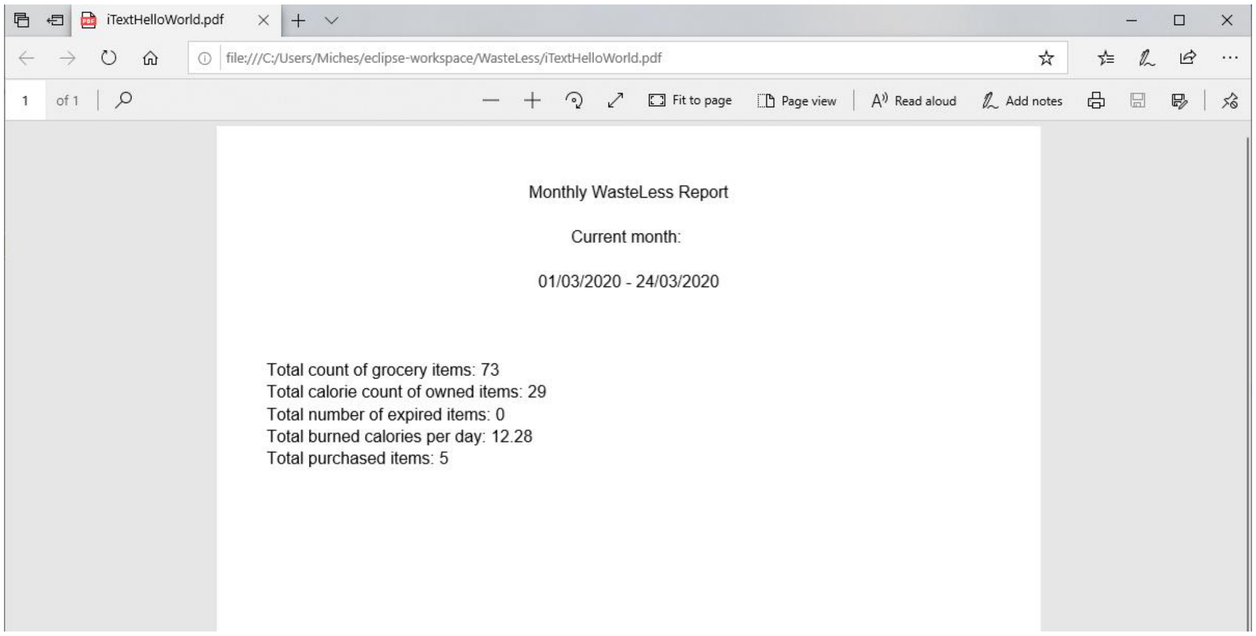
```
There is a risk of having waste! Excess daily calories: 11.28063
```

Donating items:



| | |
|---|---|
| item1 | 21 |
| item3 | 24 |

And a pdf with a generated report:


iTextHelloWorld.pdf
file:///C:/Users/Miches/eclipse-workspace/WasteLess/iTextHelloWorld.pdf
1
of 1
Fit to page
Page view
Read aloud
Add notes

Monthly WasteLess Report

Current month:

01/03/2020 - 24/03/2020

Total count of grocery items: 73
Total calorie count of owned items: 29
Total number of expired items: 0
Total burned calories per day: 12.28
Total purchased items: 5

# 5. Testing

Only manual tests have been performed and no flaws have been found.

The worst that can happen is the server connection failing. This server closes after the client connection closes! It was designed like this.

The user’s input is always checked against errors.

Main testing scenarios:

Main Screen: New Grocery List -> Add New Item -> (input valid data) -> Repeat 2-3 times -> Finish

Main Screen: View Groceries -> Data entered above appears here (this page is refreshed each time it is opened)

Main Screen: View Notifications -> If the user is to be notified about waste or items about to expire, the messages appear here. They are refreshed every time the window is opened.

Main Screen: Donate Food -> (choose item) -> Add -> Repeat 2-3 times -> Donate -> Main Screen -> View Groceries -> the items should have disappeared

Main Screen: (input new calorie goal) -> Change -> Main Screen -> Notifications -> could have triggered different waste levels

Main Screen: Generate Report -> weekly / monthly -> Project's Folder -> report.pdf

## 6. Bibliography

- Observer pattern: https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/
- Socket communication:

  https://www.geeksforgeeks.org/socket-programming-in-java/
- Spring DI: https://www.geeksforgeeks.org/spring-dependency-injection-with-example/
- Hibernate ORM: https://thoughts-on-java.org/implementing-the-repository-pattern-with-jpa-and-hibernate/
- Web LaF: https://github.com/mgarin/weblaf