

**Student: Dragoteanu Bogdan**  
**Group: 30431**

# Table of Contents

.....	1
.....	1
Student:.....	1
Table of Contents.....	2
1. Requirements Analysis .....	3
1.1 Assignment Specification.....	3
1.2 Functional Requirements.....	3
1.3 Non-functional Requirements.....	3
2. Use-Case Model.....	3
3. System Architectural Design.....	3
4. UML Sequence Diagrams.....	3
5. Class Design.....	3
6. Data Model .....	3
7. System Testing.....	4
8. Bibliography.....	4

# 1. Requirements Analysis

## 1. Assignment Specification

The assignment is an application that helps users manage food waste.

Authenticated users can input grocery lists and see reports of how much food is wasted weekly and monthly. An item in the grocery list has the following data:

Name, quantity, calorie value, purchase date, expiration date and consumption date

The system allows the users to track the goals and minimize waste by reminding them if the waste levels are too high based on ideal burn down rates.

The system gives the users options to donate excess food to various local food charities and soup kitchens.

If an item consumption date is close to the current date the user is notified.

## 2. Functional Requirements

- Search → The user is able to find data about the grocery item by searching explicitly for it
- Reports → The system will generate a weekly and monthly report for the user
- Modify → The user is able to change a grocery item's information
- AddItem → Insert a new item

### 2.1 Non-functional Requirements

- Security → The system shall ensure that data is protected from unauthorized access  
→ System data may be accessed only by users authenticated by means of a username and password
- Portability → The application works on both Windows and Linux machines
- Extendibility → Features can be further enriched

## 2. Use-Case Model

Use case: Log In

Level: user-goal

Primary actor: Registered User

Main success scenario: User log into the system

Extensions: On fail it notifies the user that something is wrong

Use case: Sign Up

Level: user-goal

Primary actor: User

Main success scenario:

Extensions:

Use case: Search

Level: user-goal

Primary actor: Registered User

Main success scenario: The system finds the item in the list and shows its data to the user

Extensions: On fail it notifies the user that the item doesn't exist or that they may have misspelled its name

Use case: Refresh

Level: user-goal

Primary actor: Registered User

Main success scenario: Refreshes the Grocery List

Extensions: -

Use case: Modify

Level: user-goal

Primary actor: Registered User

Main success scenario: Changes the data of a selected item in the List

Extensions: If some parameters are bad then the user is notified

Use case: Add Item to Grocery List

Level: user-goal

Primary actor: Registered User

Main success scenario: Adds the data of a new item to the List

Extensions: If some parameters are bad then the user is notified

Use case: Create Report

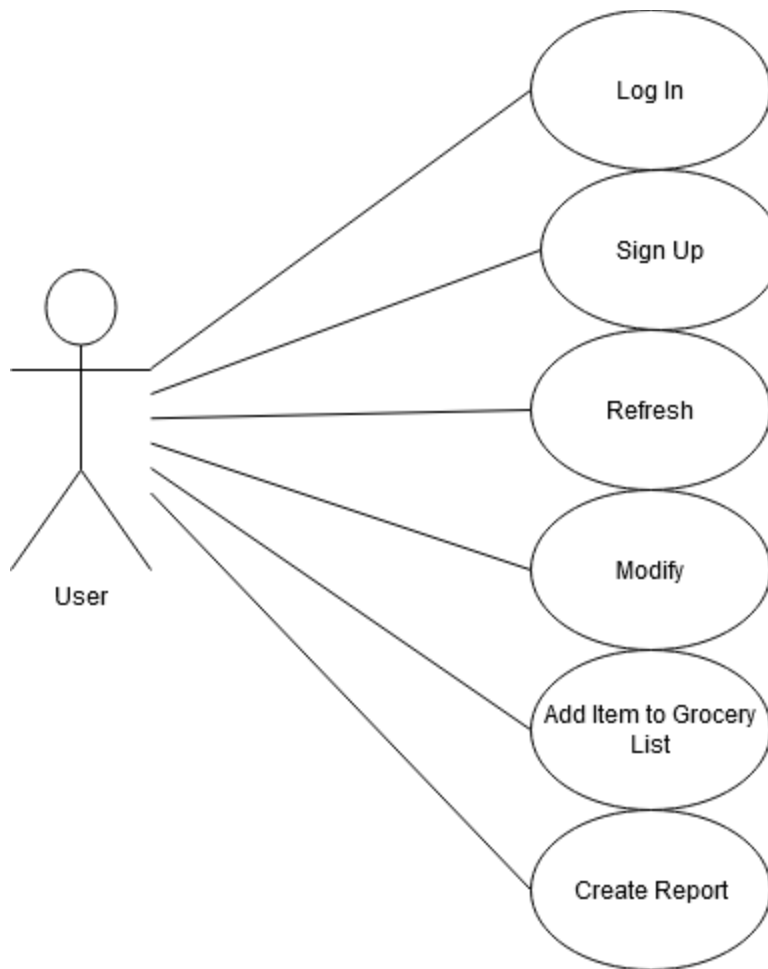
Level: user-goal

Primary actor: Registered User

Main success scenario: Creates report with the data from the grocery list and shows it to the user.

Type depends on the chosen one (Weekly/Monthly).

Extensions: If the list is empty the user is notified that nothing can be shown.



### 3. System Architectural Design

The system will be made using a client server architecture.

The client will be made using a layered architecture

#### 3.1 Architectural Pattern Description

Components in a layered architecture are organized into horizontal layers, each performing a specific role within the application. In our case we have four layers:

- Presentation → handles the user interface
- Business → handles requests
- ServerAccess

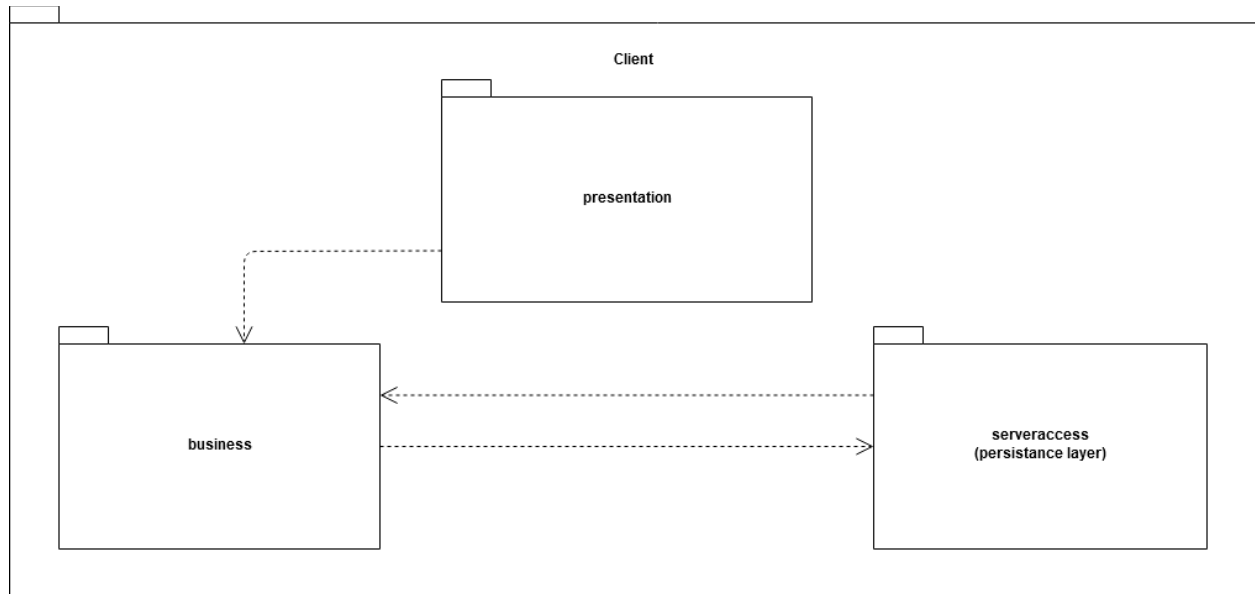
The layers are closed → requests move from layer to layer:

presentation → business → serveraccess

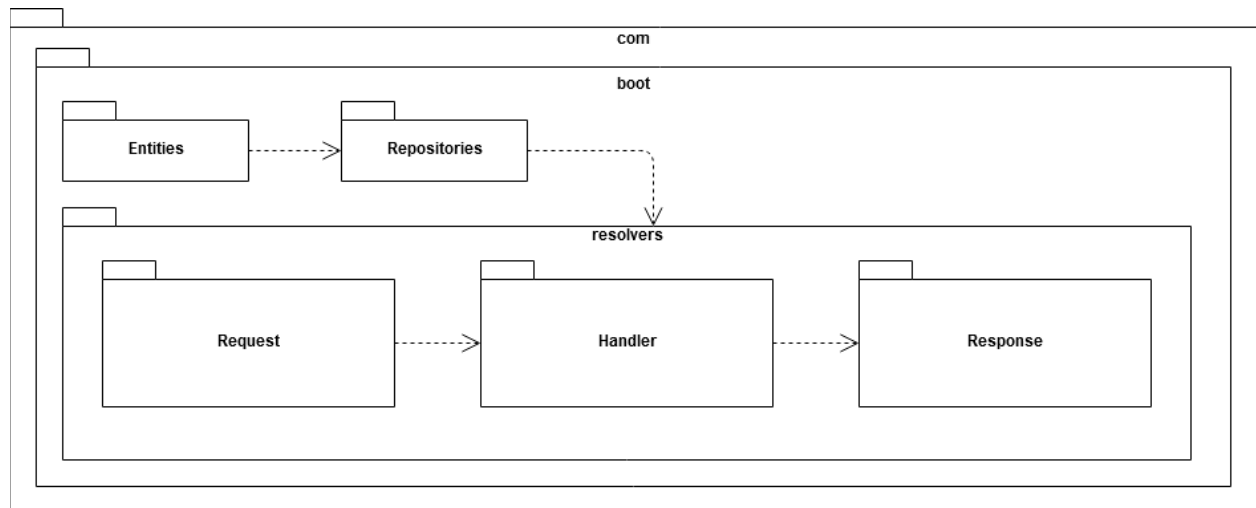
## 3.2 Diagrams

### Package Diagram

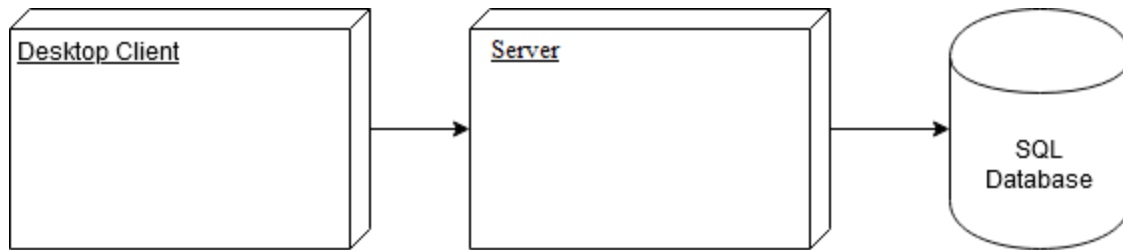
#### CLIENT



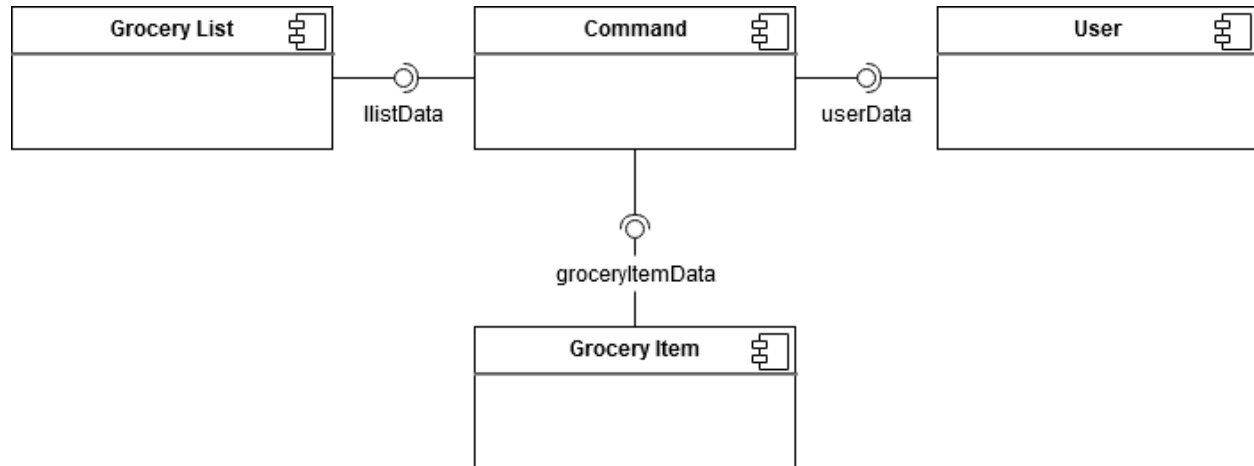
#### SERVER



## Deployment Diagram



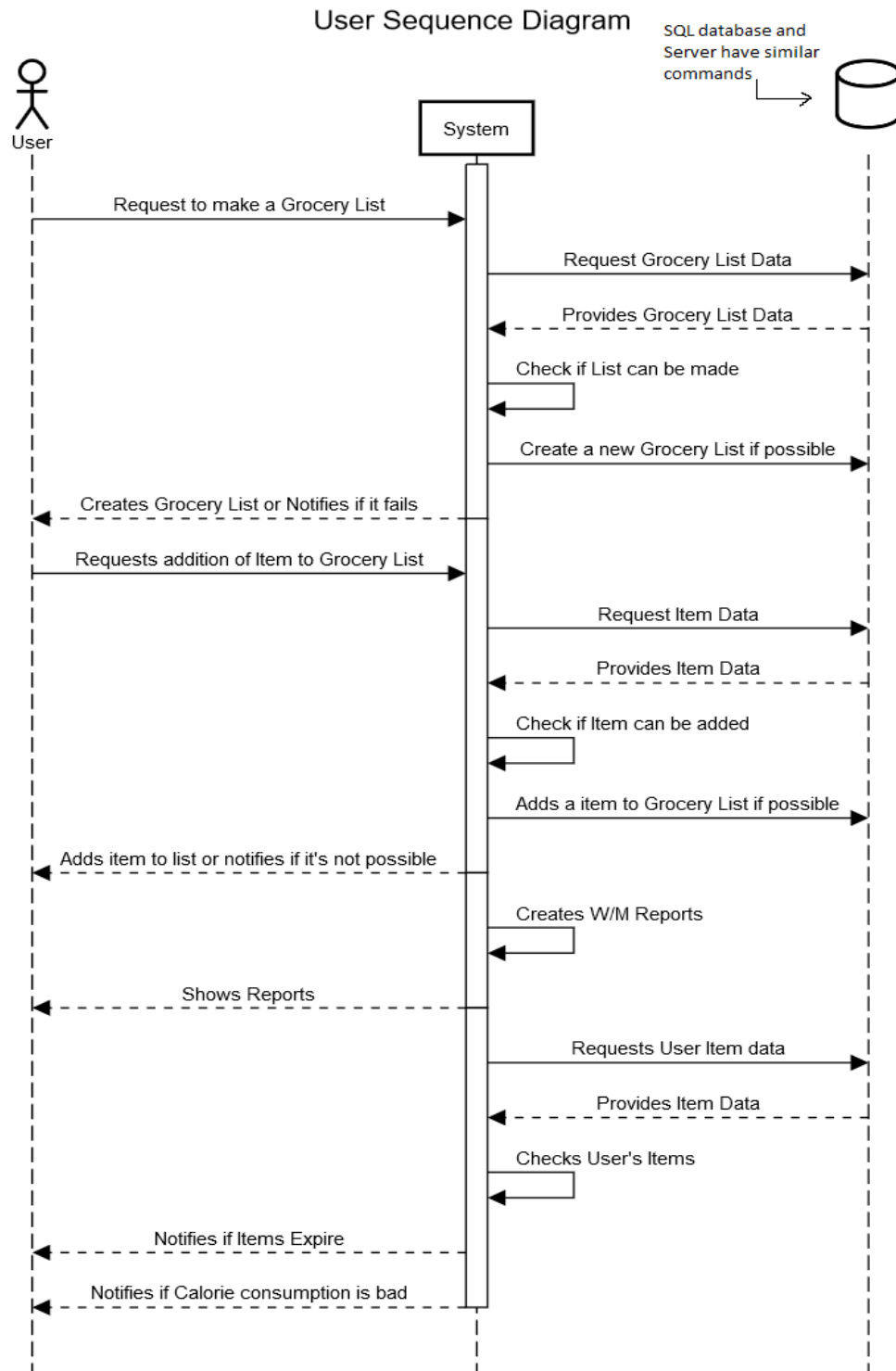
## Component Diagram



## Patterns used:

- layered → used to structure the program into groups of subtasks  
→ each layer provides services to the next higher layer
- Server will be using an ORM and dependency injection that come from the libraries used:
  - Spring
  - Graphql
- Observer → Used to notify the user if today is the final day set for certain item:
  - Notified with a list that contains the name and list of the item
- Decorator Pattern → separate the result writing of a bad or good calorie intake
- Mediator Pattern → separate requests from responses (reduce coupling)  
→ makes implementing and modifying functions easier

## 4. UML Sequence Diagrams





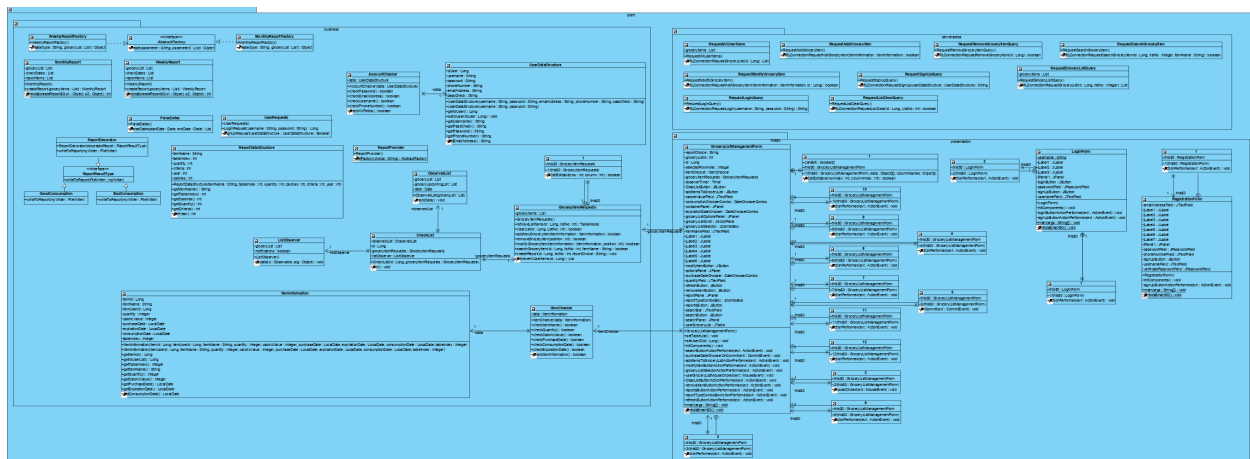
## 5. Class Design

### 5.1 Design Patterns Description

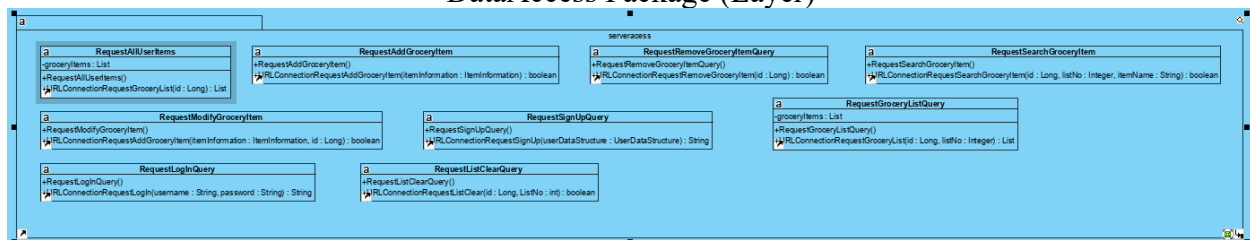
- Abstract Factory → Used in the creation of Weekly/Monthly Reports
  - Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Layered → used to structure the program into groups of subtasks
  - each layer provides services to the next higher layer
- Observer Pattern → Notify the user if some item is close to consumption date
- Mediator Pattern → separate requests from responses (reduce coupling)
  - makes implementing and modifying functions easier
- Decorator Pattern → separate the result writing of a bad or good calorie intake

### 5.2 UML Class Diagram

#### CLIENT



#### DataAccess Package (Layer)



```

classDiagram
    package business {
        class WeeklyReportFactory {
            +WeeklyReportFactory()
            +createType(String, groceryList: List) Object
        }
        class MonthlyReportFactory {
            +MonthlyReportFactory()
            +createType(String, groceryList: List) Object
        }
        class MonthlyReport {
            +groceryList: List
            +checkDates: List
            +reportsItems: List
            +MonthlyReport()
            +localizeReport(groceryItems: List) MonthlyReport
            +embedsCreateReport(Spl_Object, Spl_Object) int
        }
        class WeeklyReport {
            +groceryList: List
            +checkDates: List
            +reportsItems: List
            +WeeklyReport()
            +embedsCreateReport(Spl_Object, Spl_Object) int
        }
        class ParseDates {
            +getDates()
            +parseDates(startDate: Date, endDate: Date) List
        }
        class UserRequests {
            +UserRequests()
            +loginRequest(username: String, password: String) Long
            +loginUpRequest(userDataStructure: UserDataStructure) Boolean
        }
        class ReportDecorator {
            +ReportDecorator(splObjectReport: Report, ReportResultType)
            +writeToReport(myWriter: FileWriter)
        }
        class ReportResultType {
            +ReportResultType()
            +writeToReport(myWriter: myWriter)
        }
        class GoodConsumption {
            +writeToReport(myWriter: FileWriter)
        }
        class BadConsumption {
            +writeToReport(myWriter: FileWriter)
        }
        class ReportDataStructure {
            +itemName: String
            +tableIndex: int
            +quantity: int
            +criteria: int
            +year: int
            +calories: int
            +ReportDataStructure(itemName: String, tableIndex: int, quantity: int, calories: int, criteria: int, year: int)
            +getItemName() String
            +getTableIndex() int
            +getCalories() int
            +getQuantity() int
            +getCriteria() int
            +getYear() int
        }
        class ListObserver {
            +groceryList: List
            +notifications: Boolean
            +ListObserver()
            +update() Observable, arg Object) void
        }
        class CheckList {
            +observedList: ObservedList
            +getNewGroceryItem(itemInformation: ItemInformation) int
            +addNewGroceryItem(itemInformation: ItemInformation) int
            +removeGroceryItem(position: int) Boolean
            +modifyGroceryItem(itemInformation: ItemInformation, position: int) Boolean
            +searchGroceryItem() Long, latInfo: int, itemName: String) Boolean
            +createRecord() Long, latInfo: int, reportChoice: String) void
            +getNewLatItem() Long, Lat
        }
        class RemInfo {
            +remId: Long
            +remName: String
            +remItemid: Long
            +quantity: Integer
            +caloricValue: Integer
            +purchaseDate: LocalDate
            +expirationDate: LocalDate
            +consumptionDate: LocalDate
            +tableIndex: Integer
            +ItemInformation(remId: Long, remName: String, remItemid: Long, quantity: Integer, caloricValue: Integer, purchaseDate: LocalDate, expirationDate: LocalDate, consumptionDate: LocalDate, tableIndex: Integer)
            +getRemId() Long
            +getRemName() String
            +getRemItemid() Long
            +getCaloricValue() Integer
            +getPurchaseDate() LocalDate
            +getExpirationDate() LocalDate
            +getConsumptionDate() LocalDate
        }
        class RemChecker {
            +data: ItemInformation
            +ItemChecker(data: ItemInformation)
            +checkItemName() Boolean
            +checkQuantity() Boolean
            +checkCaloricValue() Boolean
            +checkPurchaseDate() Boolean
            +checkConsumptionDate() Boolean
            +checkExpirationDate() Boolean
            +checkItemInformation() Boolean
        }
    }

    WeeklyReportFactory --> MonthlyReportFactory
    WeeklyReportFactory --> MonthlyReport
    MonthlyReportFactory --> MonthlyReport
    MonthlyReport --> WeeklyReport
    WeeklyReport --> ParseDates
    ParseDates --> UserRequests
    UserRequests --> ReportDecorator
    ReportDecorator --> ReportResultType
    ReportResultType --> GoodConsumption
    ReportResultType --> BadConsumption
    ReportDataStructure --> ListObserver
    ListObserver --> CheckList
    CheckList --> RemInfo
    RemInfo --> RemChecker
    RemChecker --> RemInfo
    
```

The diagram illustrates the business logic and data structures for a grocery store application. It is divided into two main packages: **business** and **reminfo**.

**business Package:**

- WeeklyReportFactory** and **MonthlyReportFactory** are abstract factories that implement the **AbstractFactory** interface. They create **WeeklyReport** and **MonthlyReport** objects respectively.
- MonthlyReport** and **WeeklyReport** are concrete report classes that inherit from **ReportDataStructure**. They contain attributes like `groceryList`, `checkDates`, and `reportsItems`.
- ParseDates** is a utility class that provides methods to parse dates and generate reports.
- UserRequests** is a class that handles user login and registration requests.
- ReportDecorator** is a decorator class that implements the **ReportResultType** interface. It delegates the reporting logic to **GoodConsumption** or **BadConsumption** objects.
- ReportDataStructure** is a base class for reports, defining attributes like `itemName`, `tableIndex`, `quantity`, `criteria`, `year`, and `calories`.
- ListObserver** is a class that implements the **Observable** interface. It is used to notify subscribers of changes in the **groceryList**.
- CheckList** is a class that implements the **Observable** interface. It is used to manage the **observedList** and provide methods for adding, removing, and searching for items.

**reminfo Package:**

- RemInfo** is a class that represents a reminder item. It contains attributes like `remId`, `remName`, `remItemid`, `quantity`, `caloricValue`, `purchaseDate`, `expirationDate`, and `consumptionDate`.
- RemChecker** is a class that implements the **ItemChecker** interface. It is used to validate the reminder item against the **RemInfo** object.

**Associations and Dependencies:**

- WeeklyReportFactory** and **MonthlyReportFactory** are associated with **MonthlyReport** and **WeeklyReport** respectively.
- MonthlyReport** and **WeeklyReport** are associated with **ParseDates**.
- ParseDates** is associated with **UserRequests**.
- UserRequests** is associated with **ReportDecorator**.
- ReportDecorator** is associated with **ReportResultType**.
- ReportResultType** is associated with **GoodConsumption** and **BadConsumption**.
- ReportDataStructure** is associated with **ListObserver**.
- ListObserver** is associated with **CheckList**.
- CheckList** is associated with **RemInfo**.
- RemInfo** is associated with **RemChecker**.
- RemChecker** is associated with **RemInfo**.

```

classDiagram
    class Actor1["my km Request"]
    class Actor2["t"]
    class UC1["1 GroceryListManagementForm"]
    class UC2["2 GroceryListManagementForm"]
    class UC3["3 GroceryListManagementForm"]
    class UC4["4 GroceryListManagementForm"]
    class UC5["5 GroceryListManagementForm"]
    class UC6["6 GroceryListManagementForm"]
    class UC7["7 GroceryListManagementForm"]
    class UC8["8 GroceryListManagementForm"]
    class UC9["9 GroceryListManagementForm"]
    class UC10["10 GroceryListManagementForm"]
    class UC11["11 GroceryListManagementForm"]
    class UC12["12 GroceryListManagementForm"]
    class UC13["13 RegistrationForm"]
    class UC14["14 LoginForm"]
    class UC15["15 RegistrationForm"]

    Actor1 -- UC1
    Actor2 -- UC1
    Actor2 -- UC13
    Actor2 -- UC14
    Actor2 -- UC15

    UC1 -- UC2
    UC1 -- UC3
    UC1 -- UC4
    UC1 -- UC5
    UC1 -- UC6
    UC1 -- UC7
    UC1 -- UC8
    UC1 -- UC9
    UC1 -- UC10
    UC1 -- UC11
    UC1 -- UC12

    UC2 -- UC13
    UC2 -- UC14
    UC2 -- UC15

    UC3 -- UC13
    UC3 -- UC14
    UC3 -- UC15

    UC4 -- UC13
    UC4 -- UC14
    UC4 -- UC15

    UC5 -- UC13
    UC5 -- UC14
    UC5 -- UC15

    UC6 -- UC13
    UC6 -- UC14
    UC6 -- UC15

    UC7 -- UC13
    UC7 -- UC14
    UC7 -- UC15

    UC8 -- UC13
    UC8 -- UC14
    UC8 -- UC15

    UC9 -- UC13
    UC9 -- UC14
    UC9 -- UC15

    UC10 -- UC13
    UC10 -- UC14
    UC10 -- UC15

    UC11 -- UC13
    UC11 -- UC14
    UC11 -- UC15

    UC12 -- UC13
    UC12 -- UC14
    UC12 -- UC15

    UC13 -- UC14
    UC14 -- UC15
  
```

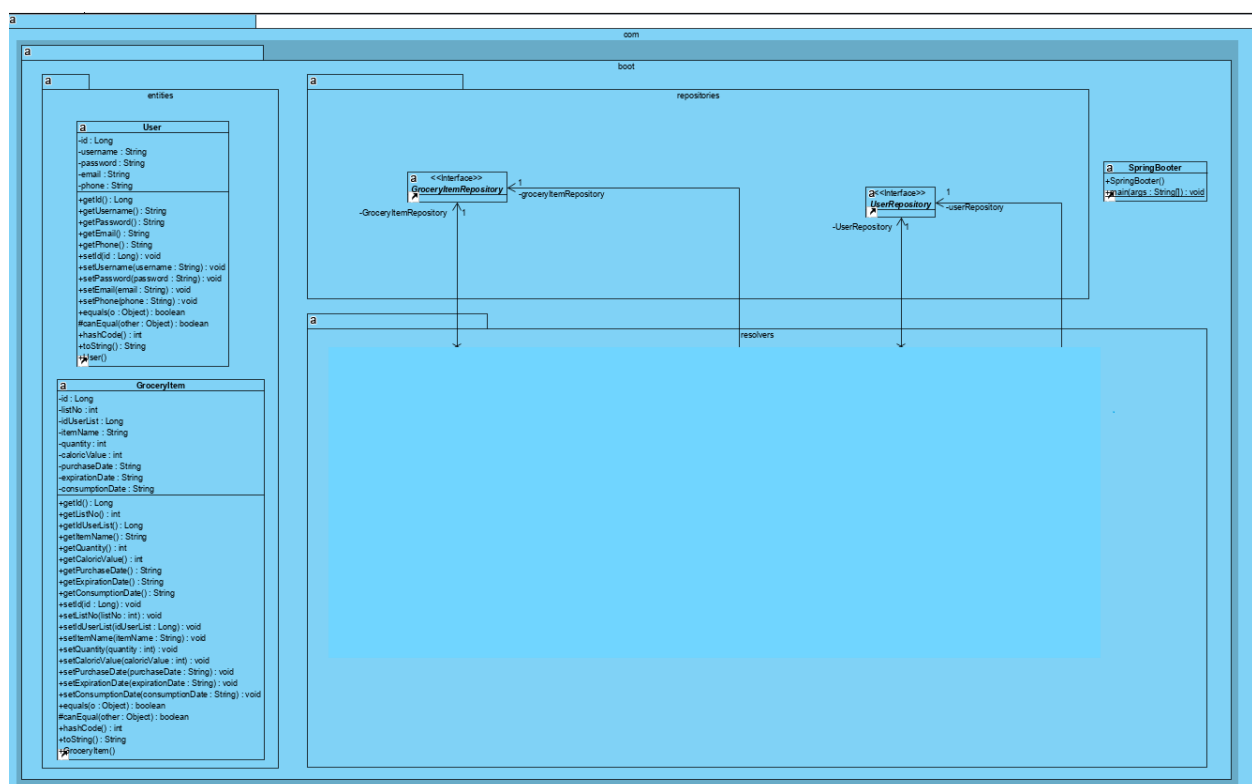
As it can be seen from the 4 images the structure of the program is divided into the three parts mentioned at the architectural pattern. This separates the data flow into 3 sections which makes it easier to troubleshoot and modify.

Abstract Factory is used to generate Factories to create reports based on the user's preference. This is a grouping of individual factories that have a common theme but no concrete classes. It separates the details of implementation the sets of objects WeeklyReport and Monthly Reports from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface .

Observer Pattern is used to notify the user if there are any items that are closed to the consumption date in his lists.

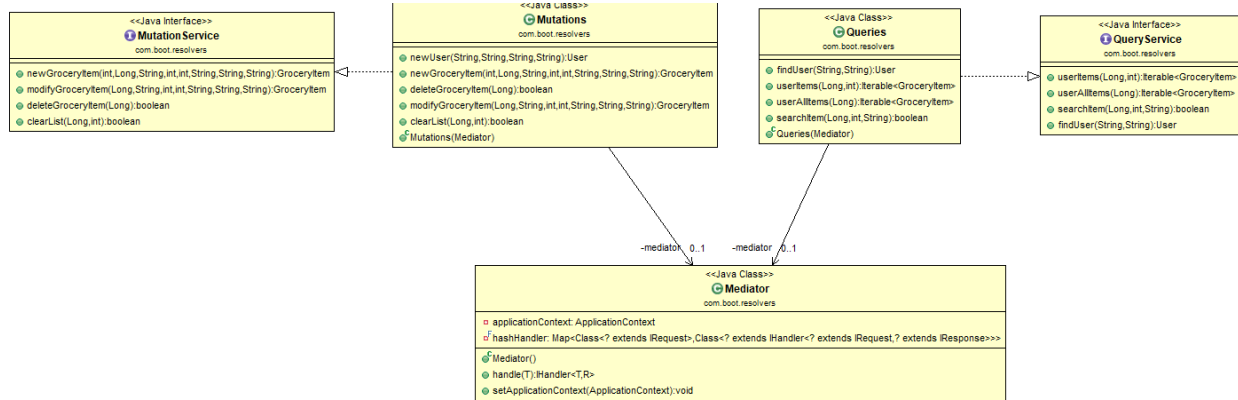
The Decorator pattern is used to differentiate how the program writes to the report how is the user's calorie intake.

## SERVER

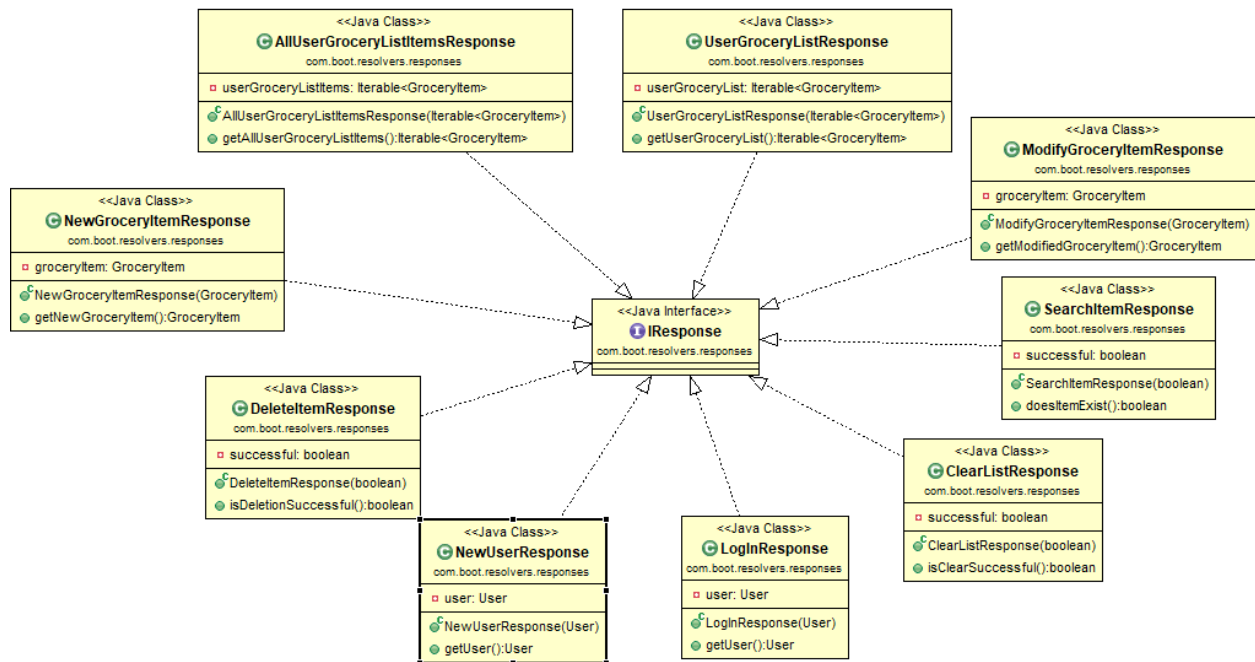


(Empty space in resolver package stems from some issues regarding Visual Paradigm.  
The following diagrams are what should've been there.)

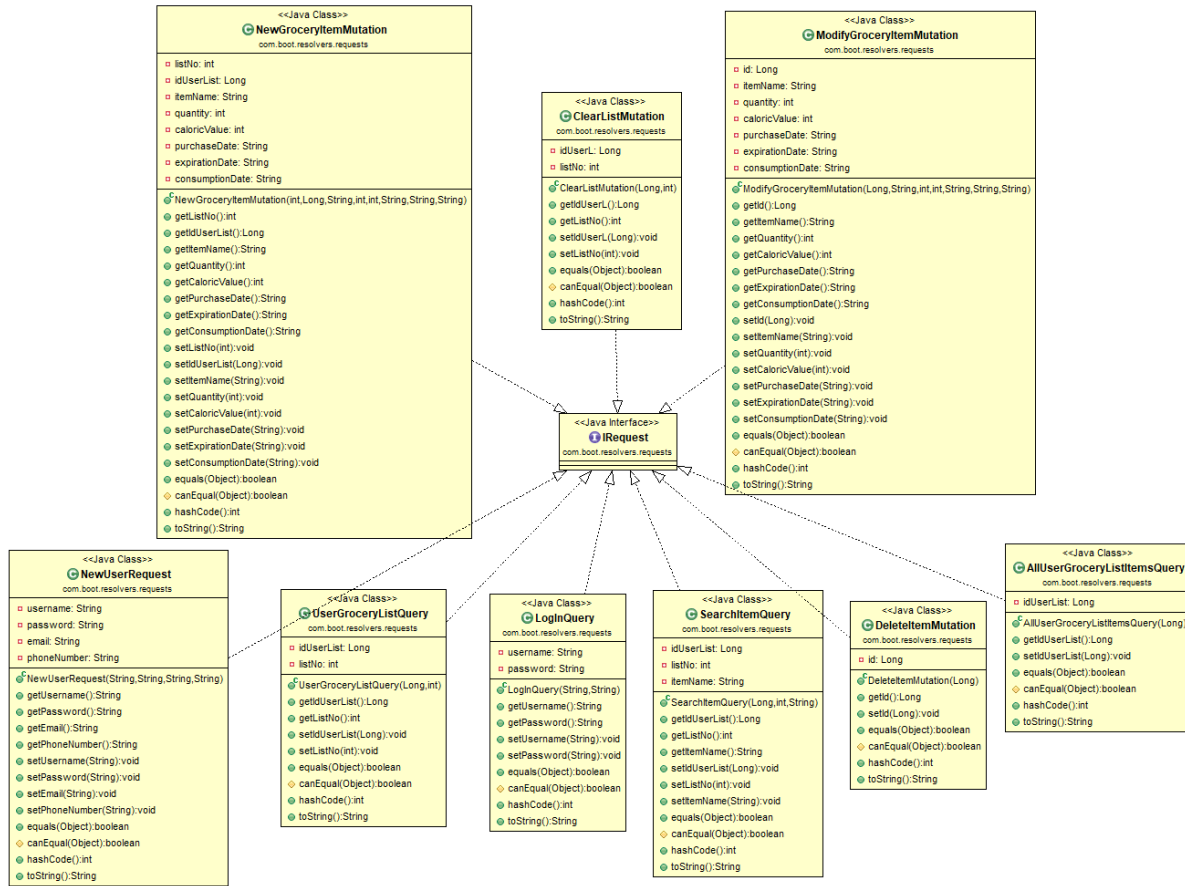
## Server resolvers (Package)



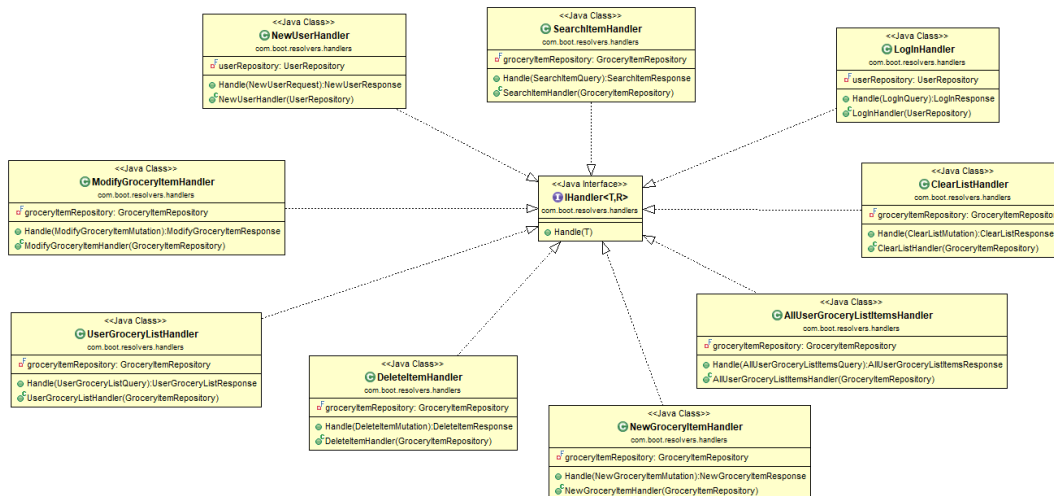
## Server resolvers.responses (Package)



## Server resolvers.requests (Package)



## Server resolvers.handlers (Package)



The server is divided into 3 main parts:

- entities – Contains the structures of the items (Contains the columns of the sql table)
- resolvers - Contains the functions that work with the database
  - This was modified to work through a mediator pattern
    - Used to separate requests and responses
    - The system receives the request (creates one for the mediator)
    - The mediator finds it and activates the necessary handler
    - The handler works with the request and creates a response
    - The response is then sent to the user
- repositories – Contains the list of items based on the entity data and id type

The server knows how to connect to the database based on the parameters in the application.properties file. Connection to the database is established by the spring framework + graphql API through hibernate.

## 6. Data Model

Relational Model for Database information:

- This model is based on first-order predicate logic and defines a table as an n-ary relation.
- Data is stored in tables called relations.
- Each row in a relation contains a unique value.
- Each column in a relation contains values from a same domain.

Entity-Relationship Model in DatabaseMS:

- Entity – is a real-world entity having attributes
  - attributes are defined by a domain
- Relationship – Mapped entities (in our case one to one (ex: PrimaryUserKey to PrimaryGroceryListKey))

## 7. System Testing

The testing method used was Unit Testing:

- Program was tested step by step through snippets of code
- Input data is verified by means of two classes AccountChecker and ItemChecker which look for bad inputs by seeing if they are in certain intervals (boundary analysis)
- Data Flow testing has been used in several areas to check if there's a proper flow (report making, retrieving Grocery List data). Here the data was checked to be:
  - properly defined in place
  - used properly (completely) for their usage
- The functions that were implemented in the server were tested using the graphql library

- using the link: <http://localhost:8080/graphql> you are sent to an interface
- in the interface graphql queries can be tested

## 8. Bibliography

[1] [www.stackoverflow.com](http://www.stackoverflow.com) → Used for Explanation of:

- Email checking
- Abstract factory (?)
- Parsing dates

[2] <https://www.baeldung.com> → Used for Explanation of:

- Spring
- HTTPConnection Explanation

[3] <https://github.com> → Used for:

- GraphQL Documentation
- HTTPConnection Example Explanation

[4] Design Patterns - Decorator Pattern

[https://www.tutorialspoint.com/design\\_pattern/decorator\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm)