WasteLess3.0

Analysis and Design Document

Student: Andrei Tiberiu

**Group: 30431**

Table of Contents

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 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

The application 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 reminders are send automatically by email, once a user has logged in, only if at least one day has passed since he has logged in last(this is used in order to avoid spamming)

The system also provides an option to donate excess food to various local food charities and soup kitchens and notifies the user of them prior to item expiration.

# Functional Requirements

# Non-functional 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 CQRS architecture, use a mediator pattern to handle requests

● Use a decorator pattern for changing the color of the report (green for above the ideal

rate and red for under)

● 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.

2. Use-Case Model

O imagine care conține captură de ecran

Descriere generată automat

*[Create the use-case diagrams and provide one use-case description (according to the format below).*

*Use-Case description format:*

*Use case: Log In*

*Level: sub-function*

*Primary actor: user*

*Main success scenario:*

*Enters correct credentials*

*Is redirected to the welcome page*

*Extensions:*

*Enters invalid credentials*

*Is redirected to the log in page again*

*Use case: Add Item*

*Level: user-goal*

*Primary actor: user*

*Main success scenario:*

*Enters valid data in the form*

*The item is added in the database and displayed on the main page*

*Extensions:*

*Enters invalid data*

*The item is not added and an error message is displayed, letting the user know the cause*

*of failure*

*Use case: Delete Item*

*Level: sub-function*

*Primary actor: user*

*Main success scenario:*

*The user deletes the item he wants from the databse*

*Extensions:*

*Impossible in this case because we assured the user can delete only existing items*

*Use case: Edit Consumption Date*

*Level: sub-function*

*Primary actor: user*

*Main success scenario:*

*The user enters the new date in a valid format*

*The user is then redirected to the main page*

*Extensions:*

*The user enters an invalid date*

*An error message is displayed*

*Use case: View Reports*

*Level: user-goal*

*Primary actor: user*

*Main success scenario:*

*The user is redirected to the reports page when he sees the resume of the data*

*Extensions:*

*Not possible in this case*

*Use case: Donate*

*Level: user-goal*

*Primary actor: user*

*Main success scenario:*

*The user is redirected to the page where the current available charity organisations are listed*

*Extensions: Not possible in this case*

*]*

3. System Architectural Design

**3.1 Architectural Pattern Description**

*As a requirement we had to choose a CQRS architecture. I have chosen to extend the three-layer architecture from the previous assignment, so that is respects the CQRS model. The Command and Query Responsibility Segregation (CQRS) pattern separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security. The flexibility created by migrating to CQRS allows a system to better evolve over time and prevents update commands from causing merge conflicts at the domain level. Some benefits of the pattern are:*

*-A clear Read model with a list of queries and domain objects you can use*

*-An isolation of each command inside a Write model*

*-A simple definition of each query and command*

*-Logging queries and commands*

*-Being ready to optimize the Read model or the Command model at any time (considering the underlying database)*

*-And also writing new queries/commands without breaking previous changes*

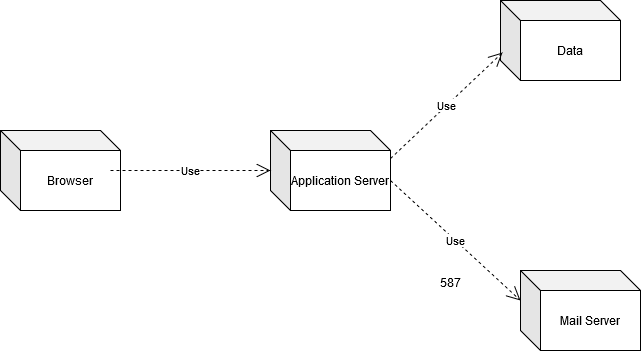
**O imagine care conține computer, fotografie, așezat, bărbat

Descriere generată automat**

**3.2 Diagrams**

*O imagine care conține captură de ecran

Descriere generată automat*

**

4. UML Sequence Diagrams

*O imagine care conține captură de ecran

Descriere generată automat*

5. Class Design

**5.1 Design Patterns Description**

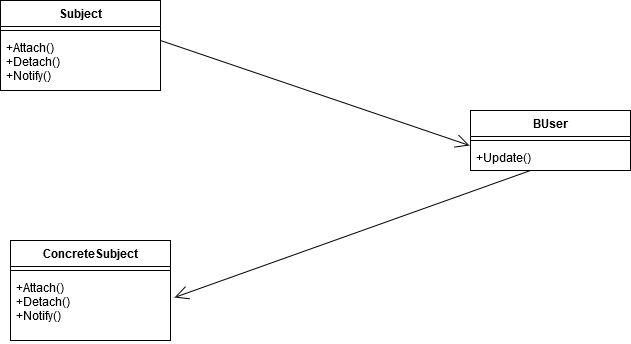
*Observer design pattern has been used to send notification to the user when items have just expired. We have an interface ISubject which is implemented by a class concrete subject that holds a reference to a list of users. This class implements the methods attach, detach and notify. Whenever notify is called, all the observers call their update method.*

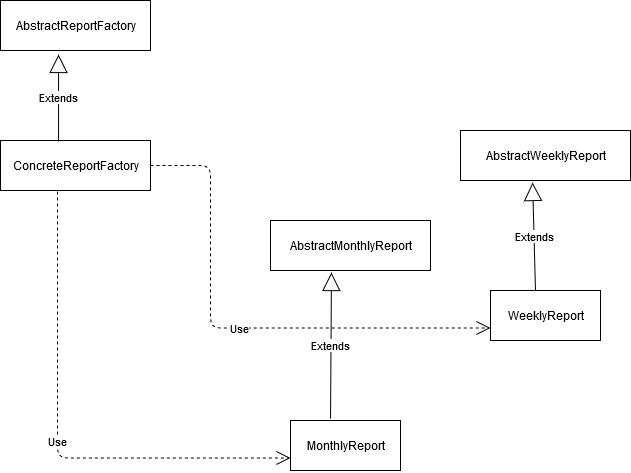
*Abstract factory design pattern has been used. We started by creating an AbstractReportFactory abstract class, containing the abstract methods createWeeklyReport and creatMonthlyReport. This class was extended by ConcreteReportFactory, which implemnts the abstract methods. The factorty products, in our case, will be the classes WeeklyReport and MonthlyReport, which extends the abstract classes AbstractWeeklyReport and AbstractMonthlyReport respectivly*

**5.2 UML Class Diagram**

**O imagine care conține hartă, text

Descriere generată automat**



**

*O imagine care conține captură de ecran

Descriere generată automat*

*The class diagram above is the same for UserManger, MailManager, NotificationManager and CharOrgManager*

*O imagine care conține captură de ecran

Descriere generată automat*

6. Data Model

*The data models used in the implementation are:*

*User: store the log in credentials*

*FoodItem: Stores all the information about the food, which is needed for the creation of both the reports and the notifications*

*MailBot: Stores the credentials required to automatically send email notifications*

*CharOrg: Stores the address and the phone number of the current available charity organisations*

7. System Testing

*During development, each individual component has been tested individually, and then combined with the other modules already developed. Data flow testing has also been used used to assure the final product executes without bugs.*

8. Bibliography