<Assignment A3>
Analysis and Design Document

Student: Rusu Andrei
Group: 30431

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

The purpose of this assignment was to create an application using any OOP language, that would fullfill a few requirements, detailed below.

1.2 Functional Requirements

The system should allow a user to authenticate, add gorcery lists and grocery items to each list, and manage their waste levels. All the data is to be stored in a database (MySQL in this case) and all of the inputs are to be validated before submission to the database.
The system should be implemented using a CQRS architecture and a mediator for managing requests. It should also have the ability to display reports about the user's waste levels, specifically using a decorator for displaying different types of reports.

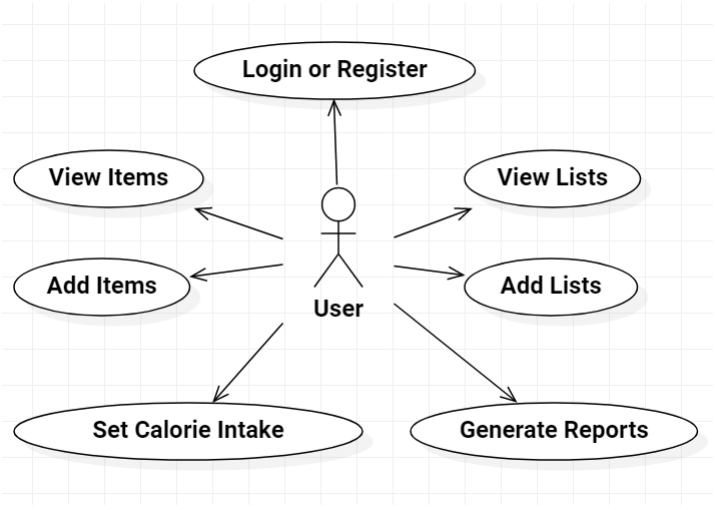
1.3 Non-functional Requirements

The application was required (if not explicitly, then by common sense) to be maintainable (hence the modularity), testable, have good usability, and also, from a later development standpoint, to be scalable and extensible further on.

Many of these requirements are fullfiled by using a good architecture, logic and coding style, which were employed during the course of this assignment.

2. Use-Case Model

The use case model is essentially the same as the previous assignment, with added report generation. The main changes are unde the hood, and they are described in the next sections.
The use case diagram is presented below:



Use case: Generate Reports
Level: Sub-Function
Primary actor: End User
Main success scenarioThe report is generated successfully and displayed on the screen

3. System Architectural Design

3.1 Architectural Pattern Description

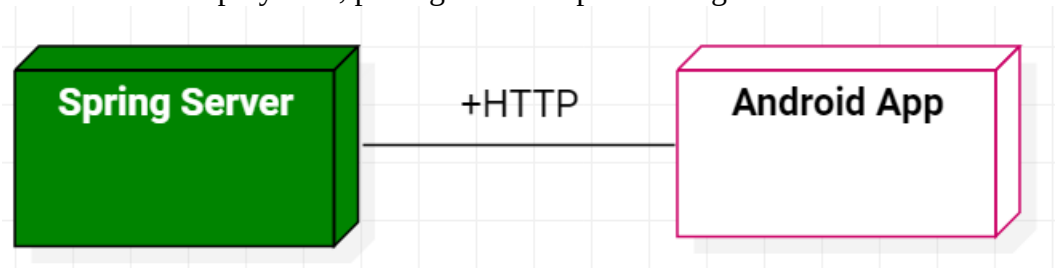
The architectural pattern used in this system is CQRS (Command-Query Responsibility Seggregation). This is a paradigm that is quite different from the regulat CRUD approach, where data access and data modification is handled uniformly.
In the CQRS architecture, there are separate models for querying and for modifying the data. They are now separated due to the fact that it is more efficient to handle reads and writes from the database separately, since writes take much more time than reads, and generally, reads happen about 10 times as often as writes, so it is a good ideea to separate them. Needless to say, it also helps with scalability, since each model can be scaled separately. CQRS in this project was developed from the client-server architecture of the A2 assignment.
Furthermore, a mediator is used to handle the requests coming to the server. Requests can either be Queries (read the data) or Commands(modify the data). These all have their separate classes and Handlers, which are essentially the "executors" of the requests: they are given a request and generate a response based on it. The mediator is what links everything together, such that the classes that know how to execute the requests are decoupled from the classes that issue the requests. It contains a type map, where the types of requests are mapped to the type of handler they require to be executed.

3.2 Diagrams

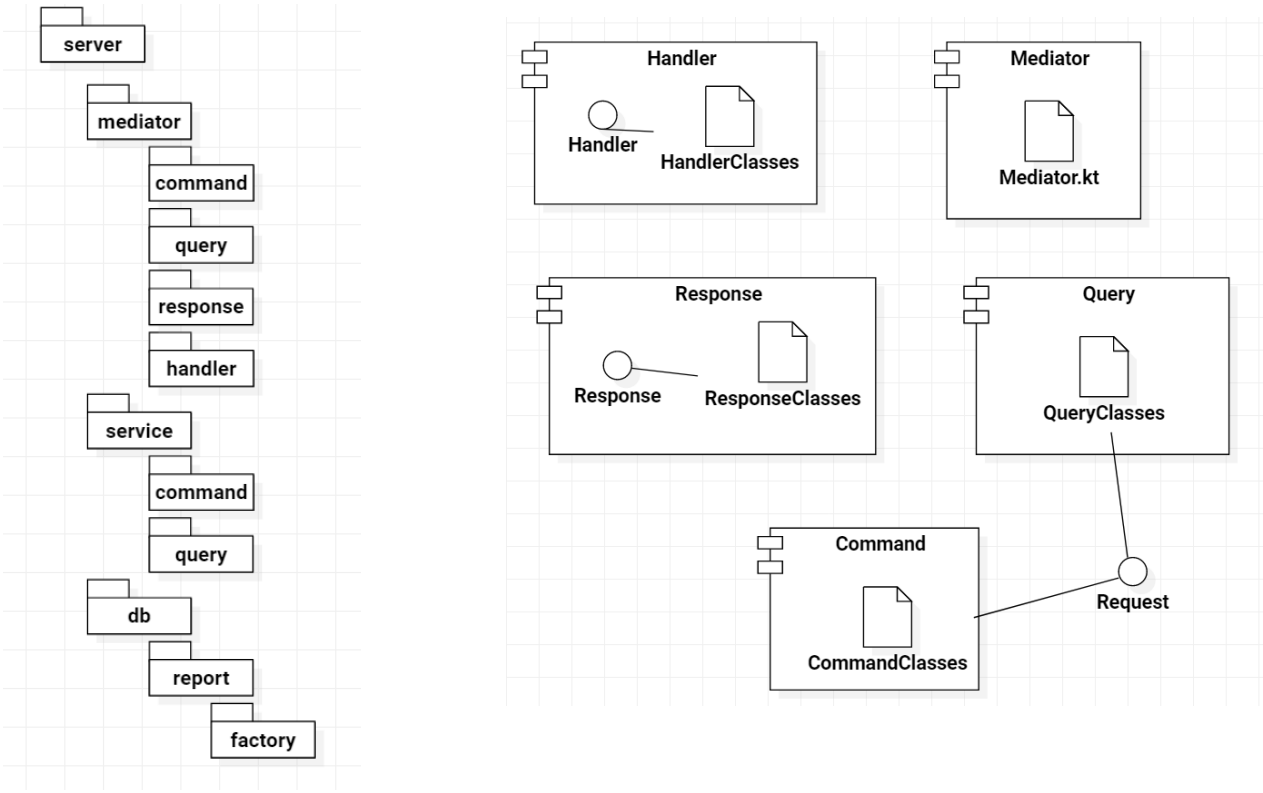
Below is the architecture diagram that pictures all of the above components together:

The classes following the Service classes were left out of the diagram, since they are the same ones from the Client-Server architecture of the A2 assignment (Model, Repository).
As it can be seen, the Controller classes do not interact directly with the service classes as before, but instead they interact with the mediator, which then supplies the handler, which in turn supplies the response from the request received in the Controller from the client. What's more, the handler only uses the type of Service class that it needs: a query handler will only use the necessary query Service class, and by analogy, the same goes for commands.

Below are the deployment, package and component diagrams:

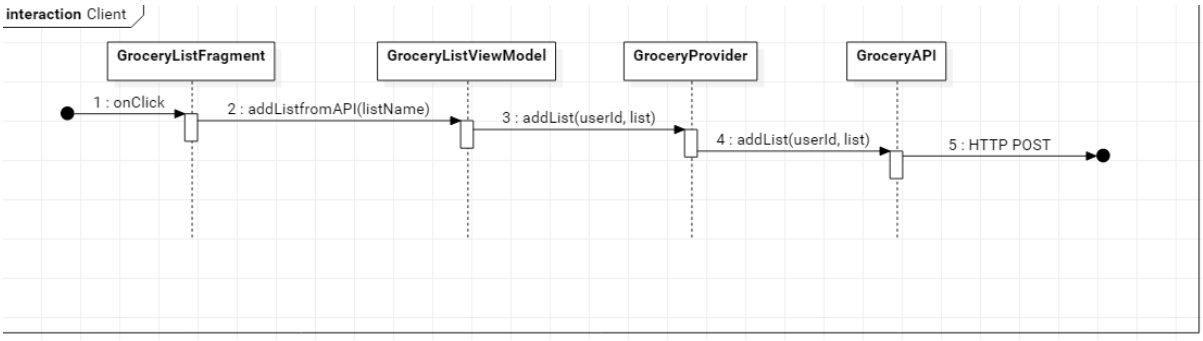


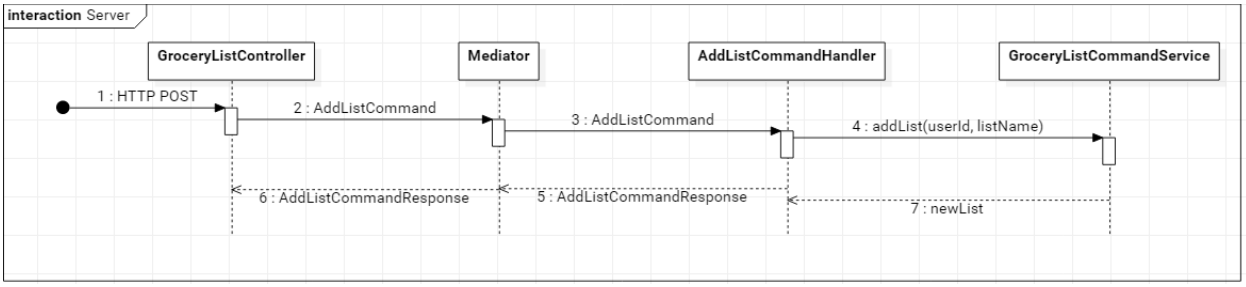
The rest of the packages and components, that were unchanged from the previous assignment, were left out to avoid clutter.



4. UML Sequence Diagrams

The sequence diagram for the scenario of adding a new list is presented below:



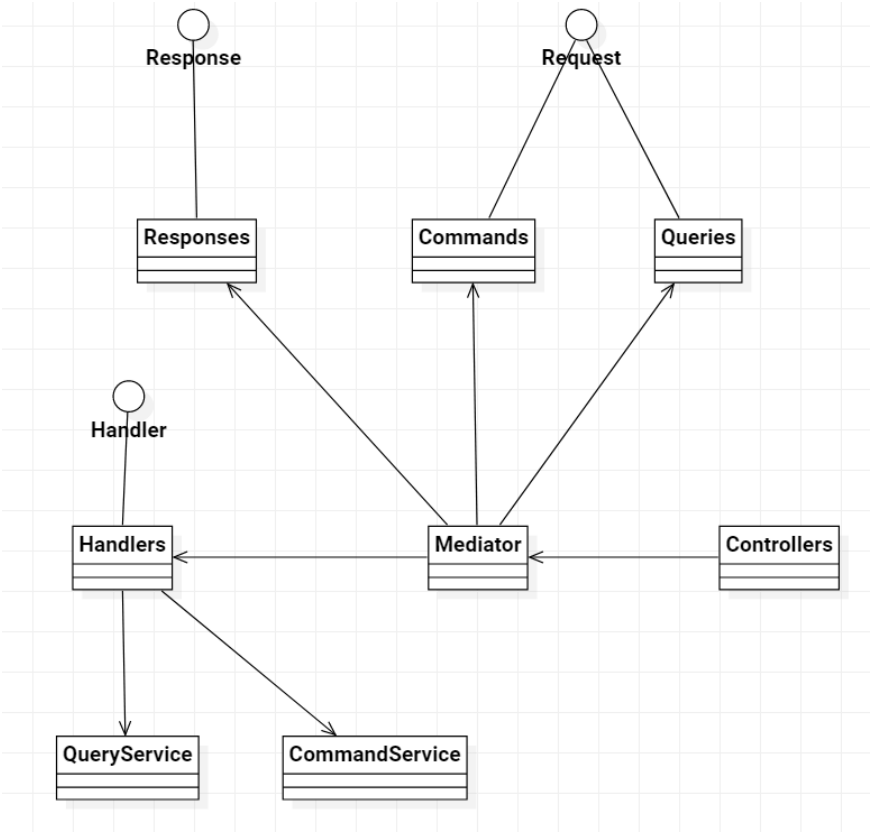


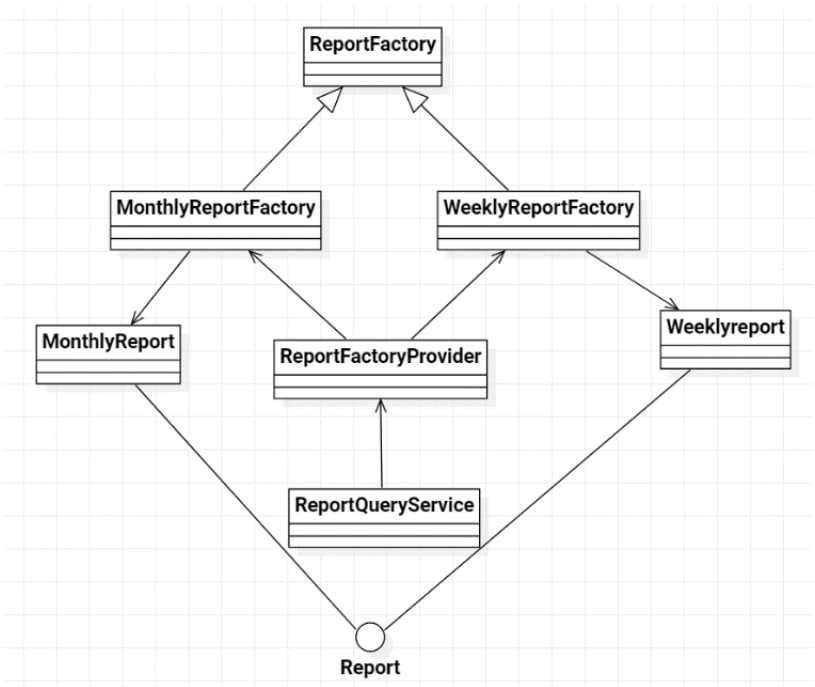
5. Class Design

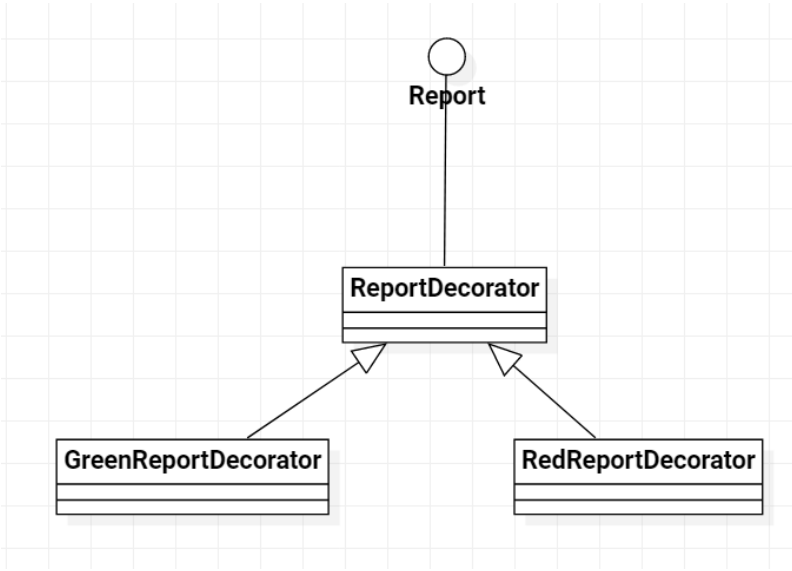
5.1 Design Patterns Description

The main design patterns used in this project are the Mediator pattern, the Abstract Factory pattern and the decorator Pattern.
The Mediator pattern (as previously explained at 3.1) has the role of decoupling request issuer from request executor and response generator classes. This is done using Handlers that process Commands or Queries and generate Responses.
The Abstract Factory pattern has been previously used in A1 and it was explained there.
The Decorator pattern is a structural pattern that allows adding new functionality to an existing object without changing its structure. It wraps the initial object (the Report) into separate decorator classes, which change the way each report looks (green for ok, red for bad). Thus, the initial report is unchanged in structure, and only the wrapper class adds functionality. Also shown in the diagrams below:

5.2 UML Class Diagrams







6. Data Model

The data model is essentially the same as the A2 assignment, with the exception of the added Reports and ReportDTO.
The Report is an interface which specifies what the classes that implement this interface can have or do. It specifies a field called waste, which indicates the waste levels of the user and a method called show, which, as the name suggests, is used to show the report on the client screen.
The interface is implemented by the MonthlyReport and the WeeklyReport.
The ReportDTO (Data Transfer Object) is the object used in http communication between the client and the server, and essentially represents a Report.
The other models (User, GroceryList, GroceryItem) are unchanged.

7. System Testing

The testing methods used were integration testing and unit testing.
Integration testing is a follow-up of the testing done in the previous assignment, since the client must work the same, because only the architecture of the server has changed.
Unit testing was performed on certain classes to test the interactions between them (especially important in the Mediator-Handler communication, since mistakes can be easily made)

8. Bibliography

https://martinfowler.com/bliki/CQRS.html
https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
https://en.wikipedia.org/wiki/Mediator_pattern
https://en.wikipedia.org/wiki/Decorator_pattern
https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm
https://www.tutorialspoint.com/design_pattern/mediator_pattern.htm