Phase 1 : Design Document

Your updated specification. Briefly highlight any additional functionality that you have implemented between phase 0 and the end of phase 1.

Specification:

Every food has a name, a quantity, and a measurement. The user specifies which foods they have, and whether they are perishable or non perishable. Perishable foods have an expiry status and date. The system keeps track of the user's foods, and keeps track of the expiry date and status of perishable foods. Upon expiring, the system alerts the user of the expired status of the food every 24 hours. The user can delete food from the inventory.

Alongside foods are recipes, which specify recipe ingredients, and the measurements and quantities of these ingredients needed. The system keeps track of the user's foods and suggests recipes based on the availability and expiry status of the ingredients. The user can choose the number of recipes that will be returned. These recipes are ranked and returned back to the user, who is free to select which one to use. Additionally, the user is able to add in their own recipes. Upon request, the user can also see all the food stored in the program.

New features: Can delete food, add recipes, choose number of recipes returned from search, User is able to see all food in inventory

A description of any major design decisions your group has made (along with brief explanations of why you made them).

- Decided to implement the Android GUI/App, as operations such as keeping track of our fridge, or finding recipes to cook is more convenient on the phone, and more accessible. Furthermore, using an app would allow users to easily access our service at all times, and the app will run in the background continually, alerting the user of expired foods even when the user is not actively using the app.
- Considered using Observer design pattern, but in the end decided against it
    - PerishableFood already extended parent class Food, therefore there would be problems with using the Observer pattern on PerishableFood
    - Additionally, there was no need to implement the observer design pattern since there was only one class that needed to be kept updated, and the observer design pattern is commonly used to address the issue of many classes that are dependent on a certain class, which need to be updated. This problem that the observer class addresses simply was not a problem that we have
- Decided not to web scrape
    - Given the scope of our program, we decided against including a feature in which users are able to utilize web scraping in order to add additional recipes to our

system. This web scraping idea became an issue when web scraping resulted in unorganized csv's which did not follow a universal pattern. As a result, we were unable to write a method to handle and clean web scraped data. To tackle this, we added an alternative feature, which allows the user to add recipes to the system, one at a time.

- Decided how users should be alerted once a food in the system expires
    - We decided to alert users on a 24 hour loop, so the user would be continuously updated on their food's expiry status. When a user starts the program, a message is printed for the user, indicating whether there are expired foods. If there are expired foods in the system, the user will be alert of them along with the given message. In addition, the user can manually call a method to see if foods are expired in the system.

A brief description of how your project adheres to Clean Architecture (if you notice a violation and aren't sure how to fix it, talk about that too!)

Command Input and Data parser are part of frameworks and drivers. These frameworks and drivers, taking in commands from the user, then call for the controllers (food controller and recipe controller) to create and run methods from the use cases (foodHandler and recipeHandler). The use cases run their methods on the entities (PerishableFood and NonPerishableFood) as directed by the controllers, either by creating the food objects or altering pre-existing ones. The use case methods then relay/return the result of these changes to the controllers, which then relay/returns the result back to command input, which alerts the user of the changes. Our project thus showcases the upward dependence of entities on use cases, use cases on controllers, and controllers on framework and drivers, and adheres to Clean Architecture.

A brief description of how your project is consistent with the SOLID design principles (if you notice a violation and aren't sure how to fix it, talk about that too!)

- Single responsibility principle
    - The Data Parser, which is a class in itself currently has 200 lines of code and has multiple functionalities. At the moment, this class potentially violates the SIP and we plan to refactor the Data parser class to change this.
    - We may have potentially violated the single dependency principle since Data Parser currently handles a large number of methods
        - This is something we plan to refactor in phase 2
- Open-closed principle
    - Followed, food classes are closed for modification as we would need to change every other function and classes
- Liskov substitution principle
    - All the other methods use "Food" class, and the use of perishable/non-perishable food is implicit

- Interface segregation principle
    - We don't implement many interfaces to begin with, so all of the ones that we do are clearly defined and do not violate this
- Dependency inversion principle
    - We followed the clean architecture, hence we have followed the principle

A brief description of which packaging strategies you considered, which you decided to use, and why. (see slide 7 from the packages slides)

The by-feature and the by-layer strategies were considered but the by-layer strategy was used. The files are packaged by entities, use cases, controllers, and frameworks and drivers, which allows us to best follow the CLEAN architecture principle and sort our files according to which layer of clean architecture they abide by. As this followed CLEAN architecture this helped us abide by the SOLID design principle of dependency inversion.

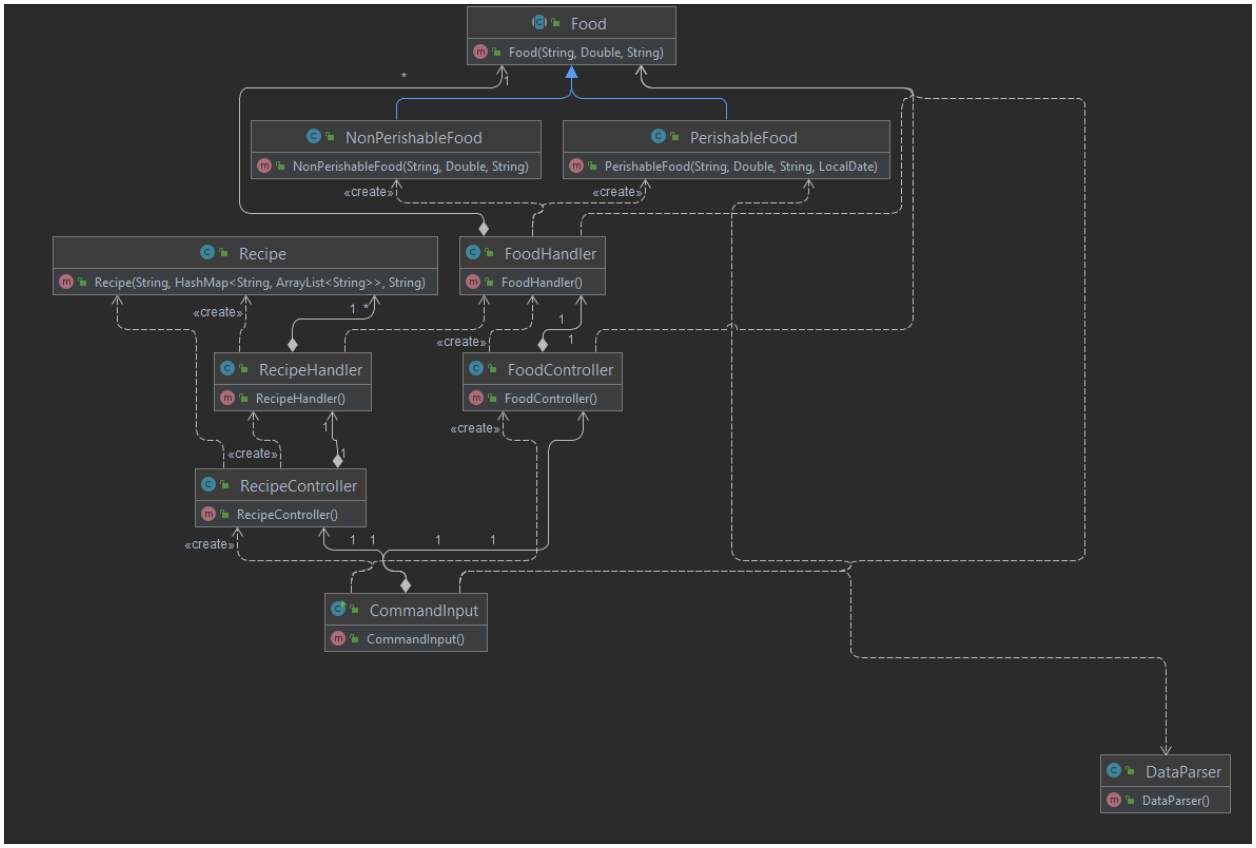
A summary of any design patterns your group has implemented (or plans to implement).

During Phase 1, we considered using several design patterns, such as iterator and observer, however, several factors resulted in the group deciding that these patterns would not benefit our project. Such reasons include existing code conflicting with these design patterns, as well as these patterns not benefiting our program's functionality due to our system's scope. Greater details regarding these decisions are outlined in the "Major Design Decision" section.

In Phase 2, our group plans to implement a Factory design pattern in conjunction with Food Handler and Food Objects, in order to support the different subclasses and use cases for Food.

Looking forward, Factory would allow FoodHandler to create different Food objects not known in advance in our program (this happens when our program first loads, and all food objects in the files are created), while efficiently calling the correct Food types to be made, such a PerishableFood, NonPerishableFood etc. This will make our program more extensible, allowing FoodHandler to continue to function as is, even if new entities are added in the future. We also plan to extend this idea to Recipe Objects and RecipeHandler.

In the case of datafiles, our program utilizes several csv files to store food and recipe information. In Phase 2, we plan on adding more files to expand our program's capabilities, such as storing Recipes in files based on regional origin. These files will be able to be called and changed according to the user input. CommandInput and DataParser will implement a Factory method, allowing the correct file to be accessed given the user input. The client would send a request to a factory, which can then call an interface to determine which file needs to be accessed, and what needs to be done (ex, extracting data, detecting data, adding data etc.) This will improve future extendibility when it comes to adding more files and will allow clients to manipulate our datafiles without specifying concrete classes or names.



The UML diagram for our project, it has a few dependency errors, but they are a priority for fixing and most issues could be dealt with in a short amount of time.

Refactoring

When it comes to refactoring, our group continuously refactored our code as we handled our tasks, making changes wherever it was necessary. This was in order to expand the functionality, and simplify our program, and includes acts such as extracting methods, and renaming.

One example of a branch that was focused on refactoring was HVM-77, which focused on renaming ambiguous method names as well as fixing long methods in the FoodHandler test cases. Essentially, this was done by extracting sample food data from most methods into a separate method, effectively cutting down the length of many of the test methods.

Another example of this is with the function recommend recipe, which originally had really long methods and bloater code smells. To tackle this, we extracted methods out of recommend recipe, and turned them into helper functions instead.

Another example of refactoring involves commandInput and dependency errors. Upon reviewing our program's UML diagram, it was apparent that commandInput violated dependency inversion by calling on Recipe and Food, which are entity classes. As a result, we fixed this issue, allowing

commandInput to solely rely on DataParser, FoodController, and RecipeController, thus maintaining clean architecture.

Overall, our group made small optimizations and refactored whenever it was relevant while working throughout phase 1.

For the next phase we plan to refactor the program at a much deeper level, in order to further optimize our code.

PROGRESS REPORT:

Open questions your group is struggling with:

1. To what extent should the single responsibility principle be followed? Classes have multiple methods, but in what scenario should a method be extracted from that class into its own?
2. Data parser works with the CSV files it has been provided with, so is it necessary to generate more tests for it knowing that it functions as it should given the actual CSVs that the program requires? If so, how could we create tests for the Data Parser?
3. Should we implement interfaces even if our current program works without?
4. Some of our current classes (such as Data parser) currently handle a large number of methods. Should we implement more classes for these methods? Should we also consider making classes for individual methods?

What has worked well so far with your design

First of all, our design follows closely with the SOLID design principles. This makes it easy to refactor code or to consider adding a new class to any of the pre-existing levels of clean architecture, given that our program functions on the upwards dependency of entities on use cases, use cases on controllers, and controllers on the frameworks and drivers. Any new code that is added to our program thus must obey this set up to correctly function.

A summary of what each group member has been working on and plans to work on next

Team member	Tasks Completed	Future Tasks
Kai	- Created the see all food functionality - added the ability to add in recipes, - added code documentation and	- Create more tests for code - Work on the android gui,

	tests for past code - Added list of all commands on boot - Added user friendly display behavior on recipe search - Search function error catching - Delete function user friendly QOL change - Search recipe functionality for controller and input - Added tests for RecipeController - Team support and bug fixes for previous functionalities - Fixed cases inequalities in user input -	- Refactor Command Input to eliminate code smell. - Stricter user input check for command input
Mark	- Added the feature to choose the number of recipes you want recommended. - Helped to implement the algorithm to rank recipes and the delete recipe functionality - Created Tests for Recommend Recipe and Recipe Handler class	- Create more tests for code - Complete the Delete recipe functionality
Nathan	Worked on developing the Android GUI for our service and helped update command input to make the command interface more user friendly (fixed the command input structure/syntax). Additionally worked on the design document and presentation.	Create more tests for code Work on the Android GUI
Maggie	- created the design document - helped to add the expiry update feature to our program - added documentation and tests for past code - added tests for FoodHandler - Refactored FoodHandler and FoodHandlerTest	Create more tests for code Help with designing the Android GUI Improve the 24 hour update food functionality
Yvonne	- Created entity tests and tests for the delete food functionality	Create more tests for code Help with designing the Android GUI

	- Added the functionality to delete food from the program and database - Added a way to add and fetch recipes from the web - Assisted with writing tests for reccomendRecipe, foodController, and foodHandler.	
Carmel	- helped to add the expiry update feature to our program, - Created 24 hour time loop on command input - added documentation - created FoodController tests - helped to write the design document - helped create the presentation	Create more tests for code Improve the 24 hour update food functionality