Design Document

Your updated specification. Briefly highlight any additional functionality that you have implemented between phase 0 and the end of phase 2.

Every food has a name, a quantity, and a measurement. The user adds the food they have to the app. If the food is perishable, the user adds an expiry date - if the food is non-perishable, the user does not need to specify an expiry date. The app lists all the food in the food panel, while keeping track of the user's food, expiry dates, 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 app as well.

Alongside foods are recipes, which specify recipe ingredients, and the measurements and quantities of these ingredients needed. The app 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 in the settings of the app, which will then be ranked and shown on the recipe page. Additionally, the user is able to add in their own recipes.

Give us a concrete example from something like your UI or an interaction with a database to show that the Dependency Rule is consistently followed.

The user opens the app and adds one food by clicking the plus button, and inputting the name as "Potato", the amount as "2", and the unit as "lbs" when the input box appears. Android passes into the adapter method createFood the strings "Potato", "2", and "lbs". The adapter then creates a list from these strings and passes the list into the food controller method runFoodCreation, which passes the same list to the Food Handler which takes the list and makes the food object with the name of the food, the amount, and the unit. The UI is dependent on the adapter, which is dependent on the controllers, which is dependent on the use cases, which is dependent on the entities.

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

- We implemented the Android GUI/App since a food list and food tracker should be easily accessible, especially in the day to day process of cooking, buying food, and keeping track of expired foods. The app allows the user to access this service any time, while also

running in the background so that the user can be continually updated on any foods that have newly expired.

- We moved our data source from package CSVs to android app internal storage, as the android application did not support saving and deleting data from the CSV file. To counter this challenge, we created AndroidDataParser which creates, reads, relays and edits the data stored in the android APK data structure to maintain our functionality on the Android GUI such that food could be stored, added, and deleted, and recipes could be stored and added.
- For the user interface of our Android app, to make navigating through the app more intuitive, we created three main panels/activities. Food: Food tab is the panel/activity that contains all the food that the user has entered into the program which would serve as the "inventory" in the app, Recipe: Recipe tab lists a number of recipes on the page depending on the number of recipes the user has set to be returned, Settings: settings is the system preferences customizing panel/activity where the user can change colour scheme of the application from light mode to dark mode (supported by android), and the user can adjust the font size used throughout the application. Each panel has a scrollable layout if the amount of recipes or food surpasses the space of the phone. Making an intuitive design for the application helps us fill the embedded ethics of simple and intuitive use, and by allowing customization for the visuals we tried to account for the accessibility of the application, making it more accessible to people that require dimmed lighting from the application or have difficulties reading small texts.
- The two design principles we added during phase two were the design patterns adapter and dependency injection.
    - We implemented the adapter design pattern in adapting our java software to work with the android gui. As the application was originally on the java command lines, we needed an adapter to change the returned data type of the different methods, which was then able to display the results of the java program on the android GUI. To be specific, the adapter class in the adapter package imports all controllers and use cases, and entities, and changes the return type of the methods in the controller into a manageable string for the Android GUI.
    - Dependency injection: FoodController had a hard dependency error in allFoodToString in the previous version of the code. This was violating the clean architecture structure we had previously. Our code had previously been initializing individual Food objects to the controller. We used the dependency injection pattern here to move the initialization of the Food object out of the controller and into the handler.

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!)

Our program's Android GUI and Android Data Parser are part of the frameworks and drivers. The Android GUI relies on the adapter to call the controllers (food controller and recipe controller) to create and run methods from the use cases (foodHandler and recipeHandler). The use cases then run their methods on the entities (the PerishableFood, NonPerishableFoods, and recipes), either creating them, modifying them, or deleting them. The use case then relays the results of these modifications to the controllers, which relay and return the results to the adapter, which then translates this information to the Android GUI that can be formatted and relayed graphically to the user of the app. Our project thus demonstrates the dependent characteristics of Clean Architecture, where the entities relay information to the use cases, the use cases relay information to the controllers, and the controllers relay the information to the Adapter which in turn relays the information to the Android GUI (Frameworks and Drivers).

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
    - Command input was shortened from phase 1 - methods were extracted to a new class named UserInput, thus dividing our classes into input and output, with other commands extracted into their own classes as well shortening command input, eliminating the bloater code smell, alongside obeying single responsibility principle.
    - For example, the method within the command input that alerted and informed the user of the expiry status of a perishable food was moved into its own class called AlertExpiryStatus.
- Open-closed principle
    - The entities - foods and recipes, are closed for modification as doing so would require a mass change of the rest of the clean architecture layers; this follows with the use cases and the controllers, as changing them would require a change of the remaining clean architecture layers above each of them respectively. Whereas adding new features would simply require the developer to add methods to the corresponding class and added to the adapter and Android GUI.
- Liskov substitution principle
    - Perishable and Non Perishable foods follow the properties specified by the Abstract parent Food Class, in which they each represent foods with food names, amounts, and units (with Non Perishable foods also having expiry statuses), all foods can be substituted for perishable or non-perishable food
- Interface segregation principle
    - Very minimal number of interfaces are implemented throughout the project, and for small functionalities, and dependency injection, thus, we did not violate this principle.
- 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)

We considered the by-feature and the by-layer strategies, and we concluded it was best to choose the by-layer strategy. This is because it helped us keep the program to adhere to the clean architecture and since our program had multiple occasions where the classes interacted between classes of the same layer, thus, we found it was more convenient to organize the classes by their clean architecture layer. The files were thus packaged as entities, use cases, controllers, and the frameworks and drivers in their own folders, with one particular folder containing command helpers. This method of organization highlights the clean architecture of our program and its adherence to the design principle of dependency inversion.

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

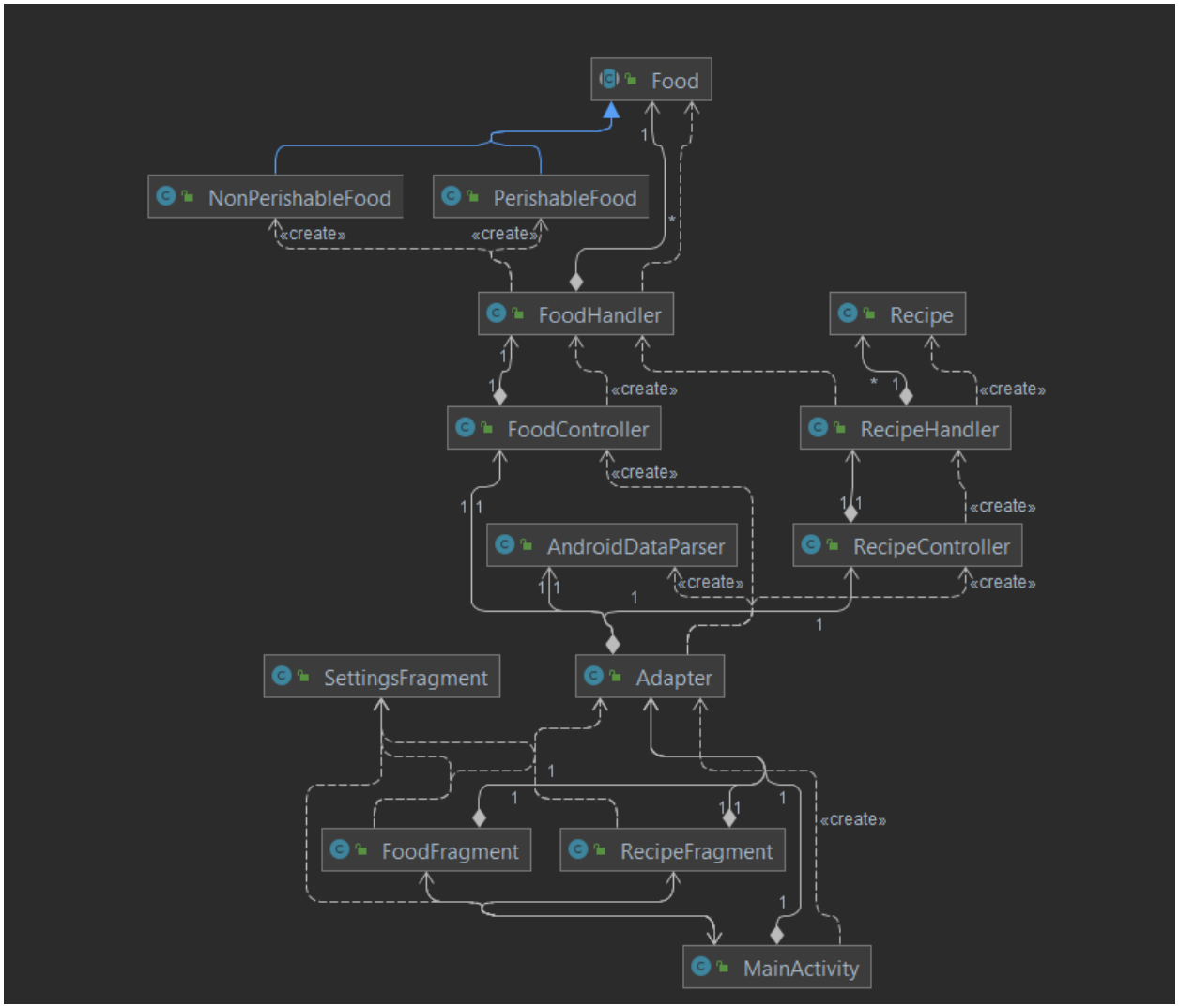
During Phase 2, a key design pattern that was added was the adapter design pattern and the dependency injection pattern.

As a brief overview, the adapter design pattern is a pattern that serves as the bridge between two interfaces/platforms, thus, without having to entirely rewrite our code, we can make it such that our original code works with different platforms, in our case, converting code that worked with default command line in java to Android GUI.

For the adapter design pattern, an adapter class was created to allow the Android GUI to interact with the java software of our program. The Android GUI calls methods from the adapter which calls methods from the controllers, and re-formats the information (Changing the data type into something more feasible for the AndroidGUI to work with) return from the controllers that can be easily formatted by the Android GUI and presented back to the user. The adapter essentially covers all the methods and functionalities covered by the original command input which was compatible with the values originally returned by the controllers but relays compatible information back to the Android GUI.

The Dependency Injection design pattern was also implemented in the FoodController to help us keep clean architecture. Before, our FoodController had a dependency on the Food Objects in the method allFoodToString. To fix this, the dependency injection design pattern was utilized in the FoodHandler; instead of the FoodController taking in FoodObjects and converting them to strings, now we made it such that the FoodHandler would now create the FoodObjects, then convert them to strings, and then return them to the FoodController. This eliminates the hard dependency on Food objects/entities in the controller, and also allows us to follow the Dependency Inversion principle/Clean Architecture Structure.

Our UML diagram for Android



Refactoring

Major refactoring was performed during phase 2. This was done to adhere to the SOLID principles, namely the single responsibility principle and the dependency inversion principle. Branches that highlight our refactoring include:

1. HVM-90: Refactored FoodHandler and FoodHandler Tests
2. HVM-85: Refactored methods and classes involving 'delete food' functionality
3. HVM-76: Refactored RecipeController
4. HVM-84: Refactored Entities
5. HVM-87: Refactored most of the recipeHandler file, extracted methods
6. HVM-88: Refactored DataParser and part of CommandInput
7. HVM-89: Shortened Command Input/Extracted alertExpiredFoods into its own class
8. HVM-83: Refactoring: Changed variable names, made variable types more abstract (ArrayList -> List), added documentation

Within these pull requests, the following changes were made:

- Variables and methods renamed to be more explicit, naming conventions followed (meaningful variable names)
- Instead of declaring interface of a List object as an ArrayList, declared it as List to be more abstract (made variable types more abstract when possible)
- Warnings were addressed, new lines were added between different methods or test cases
- Documentation was added
- Methods were extracted from classes that were too long (and where single responsibility principle was being violated)

Use of GitHub Features:

Over the duration of this project, we have used a plethora of GitHub's rich and effective features to assist us during our development. We have used GitHub branches to organize our code along with Jira to generate unique IDs for each issue. We also use pull requests to make sure that everyone's code is up to standard, as we require two approvals per pull request to merge the code to the main/android branch. Through using these we also have made sure to communicate whenever we send a pull request and this has helped our communication.

Regarding Testing in Android

Our team wrote extensive tests in the command line input environment in phase 1 and part of phase 2. Here is the test coverage for use cases, controllers, and entities:

entities	100% (4/4)	100% (23/23)	85% (55/64)
usecases	100% (2/2)	96% (26/27)	95% (134/140)
controllers	100% (2/2)	93% (14/15)	91% (43/47)

During the development of our Android environment in phase 2, we discovered that testing in our Android activities and fragments was significantly more time-consuming, resource-intensive, and simply out of the scope of our phase 2 plan. Alternatively, we decided to do testing by using our application extensively. Every new feature was thoroughly tested before merging to the Android branch. Thus, the coverage in our other classes is not an accurate depiction of the extensive testing that we have done.

Here is an example of a pull request of our testing:
https://github.com/CSC207-UofT/course-project-hivemind/pull/32/files

PROGRESS REPORT:

What has worked well with your design

Our code follows the SOLID design principles and Clean Architecture, which makes the backend easy to navigate and easy to extend and and upon without having to change the inner layers of the clean architecture. Due to this structure, refactoring and extracting the code was efficient and further reinforced the Clean Architecture. With this setup, any future extensions of this app/program must and will obey clean architecture given this pre-existing set up.

A summary of what each group member has worked on

Team member	**Tasks Completed Since Phase 1 and Significant Pull Requests**
Kaiwen	- Designed and implemented accessibility options(Dark mode, font size) - Designed and implemented SettingsFragment in Android - HVM-48-Added functionality for adding recipes and simple tests for functionalities added - Accessibility features in android environment, including slider for amount of recipe option
Mark	- Wrote Tests for handlers. - Contributed in design documentation. - Brainstormed and worked on amd documented adapters class. - Refactored Food Handler class. - Refactored RecommendRecipe function - Created Tests for Recommend Recipe and Recipe Handler class - HVM-45 Added the recommendRecipe functionality and adding the number of recipes to be recommended.
Nathan	- Refactored parts of CommandInput, and also refactored DataParser - Continued work on Android App and implemented many critical features such as viewing foods, viewing recipes, adding recipes, snackbars, loading and storage of information (HVM-99, Pull Request #69, various commits on android branch) - Wrote a new parser AndroidDataParser to work with the Android internal app storage (Made on android branch, Push ID 98f8345fbf9f5a879996b4627e6dbafd2120fb58) - Updated README.md - Worked on slides for presentation

Maggie	- Designed and implemented food popup on the android app
	  - Pull request showing the second edit made to the food popup
	- Wrote tests, documented, and refactored FoodHandler
	  - Pull request showing the refactoring and addition of tests

	for FoodHandler - Created an alert dialogue for when an expired food is added - Fixed messaging of when there is no food in the food fragment

Yvonne	- Refactored entity classes and tests
	- Refactored delete food methods
	- Implemented the delete food functionality and popup in Android
	  - Pull Request
	  - Pull Request
	- Made Presentation Slides

Carmel	- Helped write the alert system of the java program alerting the user of expired foods every 24 hours and extracted it to its own class and fixed bugs
	  - https://github.com/CSC207-UofT/course-project-hivemind/pull/46/files
	- Helped to write the same alert system but in android - wrote code for the android pop up to list out the foods that had expired upon opening the app
	- Wrote majority of the design document
	- Designed and implemented the adapter and adapter pattern
	  - https://github.com/CSC207-UofT/course-project-hivemind/pull/55/files
	- Refactored the controllers and the command input by extracting classes and getting rid of bloater code and cleaning up method and variable names

Footnote:
GitHub may not accurately reflect the amount of lines someone actually wrote. Below are some examples of how our Github insights are distorted:
Github Insights Under-representing Contributions:

- Github doesn't track the work we did on android
- Maggie did not showing up as a contributor until late in the project
- Nathan dedicated a lot of work to Android, which isn't tracked in github
- A great deal of Mark's testing code was committed on Yvonne's account
- Carmel's base code was used by other group members

Github Insights Over-representing Contributions:

- Kai and Nathan fixed errors in our program by re-copying and pasting parts of our program
- Yvonne worked on inputting data to the program