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

- We added a way for users to add ingredients to their ‘fridge’ so that they can keep track of all the items they have available for making food. Additionally, the user can now search for recipes based on the ingredients in their fridge. The program returns all the recipes that contain an ingredient present in the user’s fridge. This eliminates the need for the user to find recipes by inputting one ingredient at a time. And for the recipe class and its use case class, we have nothing done about it. However in this phase, our code can run the things about recipes. Users can find a recipe according to the ingredient the user enters, and users can know what recipe they can cook by the ingredients they have in their fridge.

- **Class Diagram**

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

In deciding the best way to store and retrieve data on recipes, our group considered both serialization and simply reading/writing to a csv or plaintext file. Since one goal of the project is to eventually have the user be able to write up and save their own custom recipes, serialization was a fine choice. However, we realized that using serialization alone wouldn’t be enough, since we needed to have a set of seed data that the program can use prior to the user adding anything of their own. We felt a csv file would better suit that purpose since data on recipes found on the internet could be used as seed data, and so that is what we decided to go with. The csv file we’ve ended up with contains around 7,000 unique recipes, each with almost all of the data we need for a recipe in our program.

- **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!)**
  In our project, we strictly adhere to clean architecture as we divided our classes into 4 distinct levels (Ingredient, recipe, user, and UserList belong to the highest level which is the entity. RecipeBook, RecipeFacade, RecipeFinder, RecipePrinter, UserFridgeManager, UserManagerLogingAndNewUser, UserManagerFacade belong to the use

case. MealSystem belongs to the controller. Lastly, SystemInOut and MealsMain are our user interfaces for now). Furthermore, our classes follow dependence on adjacent layers and we made sure higher-level classes do not depend on lower-level ones.

- **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!)**
  Our project strictly follows the single responsibility principle as each of our classes is responsible for the needs of just one business function. We have implemented the facade design patterns to protect the single responsibility principle where we think a common interface for the client is necessary when they interact with actors.

  The UserFridgeManager class implemented the observer interface (following an observer design pattern), and more features and functionalities can be extended as we defined classes that implement the observer interface following the open-close principle.

  As the observer class (UserFridgeManager) implements all methods and functionality of the CurrentUserObserver interface, hence it follows Liskov substitution principle as UserFridgeManager may be substituted for any objects of type CurrentUserObserver without altering any of the desired properties of the program.

  As none of our higher-level classes depend on lower-level ones, the dependency inversion principle is not violated.

- **A brief description of which packaging strategies you considered, which you decided to use, and why.**
  The package strategies we considered are package by layer and package by component. We decided to use package by layer as it strictly protects the clean architecture structure. Also, this way of packaging makes our communication easy as we are all aware of what level the class belongs to when we discuss and implement it.

- **A summary of any design patterns your group has implemented (or plans to implement).**
  As mentioned above, we implemented a facade design pattern to protect the single responsibility principle. Furthermore, the classes UserManagerLoginAndNewUser and UserFridegeManager implemented an

observer design pattern. When a new user logs in, UserFridgeManager needs to get updated since it needs to know which user's fridge it's dealing with. Hence, implementing an observer design pattern would allow UserFridgeManager to know as soon as a new user logged into his/her account. Also as mentioned above, we are planning to implement more observers that follow the observer design pattern in the next phase when we extend additional functionalities, as many features need to know which user they are currently working with.

- **A progress report**

  Open questions your group is struggling with:

  What is the best way to interface our csv file of recipe data with our program? We need to both read and write to file, so is there a good Java library we can use? We have GetUserIngredientsName and Find which are in two different use case classes, and we use two of them in our controller to implement a method which is about finding recipes according to users' fridge, is this approach violates SOLID or Clean Architecture?

  what has worked well so far with your design

  Packaging by layer has worked well so far as it strictly enforces the clean architecture structure. Also, in my opinion, the implementation of the facade design pattern has worked well to adhere to the solid design principle. Furthermore, implementing serialization and saving UserList directly into the ser file as an object has worked well as the program doesn't need to convert everything into String back and forth when running (opposing to saving them in for example a txt file).

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

  Rongguan: I have fixed the SRP problem in the original UserManager class. Now, a UserManagerFacade class has been introduced which acts as an interface for UserFridgeManager and UserManageLoginAndNewUser class so the client is capable of accessing any functionalities for these two classes from the facade. Also, I have implemented the observer design pattern in which UserFridgeManager will get notified when a new user logged in as mentioned above. Furthermore, I have created a new UserList entity class (which stores User), and I made both UserList class and User class serializable hence enabling the implementation of serialization in which the UserList can be stored in the userlist.ser file. At last, I added the test cases

for UserManageLoginAndNewUser and UserFridgeManager. I plan to implement functionalities revolving around users (implement more observers), and I will work on a better interface next.

Adam: My task for this phase of the project was getting all of the data we need and organizing it into a database for our program to pull from and add to. The way we format our recipes meant I had to find and adjust data from recipes I found on the internet, and also clean it up a bit so that there wasn't anything extraneous or unnecessary to our program. Moving forward I plan to complete the full connection between our program and our database so that we may read/write to it efficiently, and I also want to work on some graphical user interface components to take our program beyond just text in the console.

Daniel: I have implemented the use case class for Recipe as we want in phase 0. I created 4 more classes, which are RecipeFacade, RecipePrinter, RecipeFinder and RecipeBook. RecipePrinter can give out the string format of a list of recipes, RecipeFinder is used to find a recipe among a list of Recipe by giving an ingredient. RecipeBook is used to manage a recipelist. And RecipeFacade is used since the single responsibility principle. Also I have adjusted our Recipe class a little, to make sure it satisfies our use case and database, and implement some basic methods in Recipe class, like getRecipeName, toString….. Also I tried to implement the method which is about finding the recipe according to the users' fridge. For that, I created a method in UserFridgeManage which is called getUsersIngredientsName which turns a list of Ingredients into string format. Finally, I created tests for RecipeFacade, which can also test RecipePrinter, RecipeFinder and RecipeBook, which I think it's pretty convenient. In phase 2, I hope I can extend the functionality of Recipes, for example, users can create recipes, and the project can read it and turn it into our database. Also I am planning to extend our function about the RecipeFinder, and make sure it can have different parameters.

Arya: I was in charge of implementing the controller class MealsSystem and its tests. I worked alongside Antony to consider every scenario and make the user experience feel as fluid and friendly as possible. Apart from this, I helped the group by finding bugs and inconsistencies and lending ideas on implementing various functionalities. In the future, I plan to help implement more functionalities to broaden the scope of the project as well as work on the UI for a better user experience.

Antony: I was tasked with the implementation of the controller class MealsSystem alongside Arya. I personally worked on the user experience after they have logged in, bringing the functionality in the facades to the

console, which includes the user’s access to the fridge, adding ingredients, finding recipes for a certain ingredient and finding recipes that included at least one item in their fridge. For the next phase, I plan to heavily contribute in the UI process, from the brainstorming stage to its implementation.