Group-069 Meals

Specification

In our Meals ingredient management and recipe recommendation program, the first page it will show the user is the login page. The login page prompts the user for a username and a passcode. If it is the first time a user is using this program, there is also a create account option in which the user could create a new account by following the prompts in creating an account page. After a user has successfully logged into the account, the main menu page will be shown to them.

On the main menu page, a user can choose to add ingredients to their fridge, find recipes, view the fridge, view their shopping list and to logout. If the user clicks “Add Ingredient”, the program will prompt the user to enter the name of the ingredient and the type of the ingredient on the add ingredients page. A user can add multiple ingredients at once on this page. If the user adds an ingredient that is present in their shopping list, the ingredient will automatically be removed from the shopping list to avoid more manual effort. Once finished, clicking on the Done button will bring the user back to the main menu page.

If the user clicks the “Find Recipe” button, lists of available recipes the user can make given the ingredients they have in the fridge will be displayed on the page. A user can go back to the main menu page by clicking the close recipe book button or by entering a recipe name in the text box and clicking the get direction button to see the specific details of a recipe desired. After seeing the specific details of a recipe, a user can close the page and go back to the lists of available recipes page by clicking the go back button or they can click the cook button. The program will assume that the dish will be cooked by the user and it will automatically remove all the ingredients associated with the dish from the user’s fridge (afterward returning to the main menu page).

If the user chooses to click the “View Fridge” button from the main menu page, lists of available ingredients in the user’s fridge will be displayed, and the user can click close fridge to go back to the main menu.

Clicking the “Shopping List” button on the main menu opens a window that displays all the items in the user’s shopping list on the left side of the screen. On the right, the user can choose between three buttons, “Add Ingredient”, “Remove Ingredient”, or “Close Menu”. If the user selects either of the first two options, a box will open, prompting them to input an ingredient they’d like to add/remove from their shopping list. In the case of adding, the user also

has the choice to input multiple ingredients at once, by separating each ingredient with a comma (for e.g. beef, cheese, fish). Clicking “OK” on the box will confirm that the ingredients have been added/removed. The user back will then be returned to the Main Menu, after which they can see the updated shopping list when they choose “Shopping List” again. Choosing “Close Menu” simply closes the window and brings the user back to the main menu.

Lastly, the logout button will close the program and all operations of a user from the current session will be saved into a .ser file through serialization.

New updates in phase 2

In this phase, we implemented a GUI by using JavaFX. So instead of having the user interact with command lines, they are now able to interact with the program by clicking the clickable buttons and by inputting information in text boxes in an aesthetic and pleasant environment. Furthermore,

1. The recipe information is read directly from a csv data file which could contain as many as 7000 recipes if desired.
2. Our find recipe functionality no longer returns all the recipes that contain an ingredient present in the user’s fridge but returns lists of available recipes the user can make given the ingredients he/she has in the fridge.
3. Get recipe details functionalities which return the making directions and other useful recipe details such as estimated making time duration, cuisine type, and etc.
4. A “Cooked It” functionality was added, in which the program will automatically remove all the ingredients associated with the dish from the user’s fridge.
5. Shopping Lists were added which allow the user to store items that they’d like to buy the next time they go grocery shopping. Items can be added/removed from this list, and adding an ingredient to the user’s fridge automatically removes the ingredient from their shopping list. This saves the user’s time and improves user experience.

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

1. 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 of the main goals of the project is to provide recipe recommendations for the user, 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.

2. In deciding the type of GUI to implement, our group has considered both Spring Boot and JavaFX. Out of the consideration that we decided not to implement the community feature of our program, and the program can be run entirely on a local device, there wasn't the necessity of implementing Spring boot over JavaFX. Furthermore, styling in JavaFX can be done entirely with Java code, hence it is easier to communicate it within the team.
3. In deciding the user's operation to the recipes, our group has made some changes to make the user have a better experience to choose their expected recipes. In Phase 1, our project returns all the recipes containing the ingredient that the users will use, which will cause problems for the user choosing recipes because there are too many recipes to look at. However, in Phase 2, our project can return a list of recipes that is available for users, which means it will only contain the recipes that users can cook according to their contents in the user's fridge, this saves users' time to check if they can cook or not.
4. In deciding the user's operation that interacts with both their fridge and recipes, we implement a method called cooked that will delete the contents in the fridge that the user has cooked(this can be done after the user chooses a recipe), this functionality makes fridge real and user can update their fridge automatically after they cook, so it saves user's time that users don't need to delete ingredients one by one from the fridge by themselves. Also, we have implemented a pop-up alert box for the user before they cook and remove the related ingredients from their fridge out of accessibility considerations.
5. In phase1, there was a use case class called UserManagerFacade which implemented the facade design pattern out of the consideration of protecting the single responsibility principle and at the same time providing a common interface for the only interface adapter layer class (which was named Controller) to interact with. But as we divided the controller class into controller and presenter, it was no longer necessary to have such a common interface, and this facade design pattern has gotten rid of.
6. Regarding the MVC decision, we have chosen to implement the model view controller pattern, in which the user interacts with the user interface that will immediately ask the controller to take over, the

controller manipulates the model, and the model updates the view which will be seen by the user interface. The reason we have chosen this pattern is out of consideration of the styling purpose of our GUI module, as this pattern does not require the user interface modules to implement the update methods and other functionalities.

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, and we package them into 5 parts(entity, use_case, controller, gateways, and gui). Entity contains classes that represent a recipe, an ingredient, a user, and a list of users in this program. The use case module runs the main logic of the program which contains classes that manipulate the user, the user's fridge, the user's shopping list, and recipes. The controller package contains the controller class and the presenter class which are responsible for receiving a user's interaction with the program and presenting the user with the program's output. The gateway package contains gateway classes that provide access to the databases (ser file for serialization and csv file for saving recipes). Lastly, the gui package contains classes that help produce the graphical user interface.

Furthermore, our classes follow dependence on adjacent layers and we made sure higher-level classes do not depend on lower-level ones, or in other words the direction of the dependency always points inwards. In instances of importing data through gateways or updating the presenter, interfaces were implemented to achieve dependency inversion in regard to boundary-crossing.

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. In this phase, we have separated out the controller class into a controller and a presenter class which further protects this principle as the controller is strictly responsible for getting the data from the user and the presenter is strictly responsible for outputting the data from the model. Also, 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 implements the observer interface (following an observer design pattern), and it will get notified when a new user logs in. As there are a lot more features and functionalities that are user-related, and most likely those features would require knowing which user they are working with. Thus, we can define new classes that implement the observer interface which will extend the user-related functionalities without modifying the existing ones and hence follow the open-close principle. Such extension was demonstrated in this phase by the implementation of UserShoppingListManager. Although we had to add a new instance variable in the user entity class since we needed a representation of the shopping list for a specific user, we were able to extend this functionality easily without making further modifications by having the UserShoppingListManager implement the observer interface. Also, the gateway interfaces in our gateway module would allow us to extend to different types of databases without affecting the use cases and the entities, hence protecting the open-close principle.

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

All interfaces defined in our program are small, specific interfaces that focus on certain functionality. The interface that the service provider needs is absolutely necessary for the client to implement, thus no functionalities in the client code are unused. Hence, protecting the interface segregation principle.

As none of our higher-level classes depend on lower-level ones, the dependency inversion principle is not violated. And as mentioned earlier, in instances of importing data from through gateways or updating the presenter, interfaces were implemented to achieve dependency inversion.

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 given a number of different actors that interact with the recipe. Furthermore, the classes UserManagerLoginAndNewUser, UserFridgeManager, and UserShoppingListManager implemented an observer design pattern. One reason for implementing such a design pattern is, as mentioned, as there are a lot more features and functionalities that are user-related, implementing this design pattern allows us to easily extend the user-related functionalities without modifying the existing ones. Second, when a new user logs in, UserFridgeManager and UserShoppingListManager needs to get updated since it needs to know which user's fridge/shoppingList it's dealing with. Hence, implementing an observer design pattern would allow UserFridgeManager and UserShoppingListManager to know as soon as a new user logged into his/her account.

Major Code smells/Refactoring

Long method:

1. The find method in Recipe is way too long, it has 13 lines. It was resolved by splitting into two parts (or in other words by defining a helper method).
2. The start method of the class GUI was extremely long as it contains many styling codes. All the layout has been refactored into separate classes to resolve this issue.

Large class:

1. The UserManager class contains too many methods and it violates SRP as it manages both a user's fridge and a user's account. This class has been refactored to UserFridgeManager and UserLoginManager
2. The controller class violates the SRP and used to contain too many lines of code. This class is refactored to a controller class and a presenter class that works with the GUI.

Long Parameters:

1. Our recipe class has too many instances, which makes its constructor have too many parameters. I think we can just safe delete some of them to solve this issue (as shown under speculative generality below)

Comments:

1. All the TODOS and comments are removed
2. All commented out codes are removed

Duplicate code:

1. Our RecipeFinder some similar codes between two methods and we deleted the duplicate codes inside RecipeFinder
2. Deleted GetIngredient method in UserFridgeManager as it does a similar function as getUserIngredientName.

Dead code:

1. Removed the email instance variable in the user entity class
2. Loginpage is a class we don't use in GUI and is safe deleted

Speculative Generality:

1. Safe deleted the instance variable storingDuration and its relative methods from the ingredient entity class, as this serves the recipe recommendation functionality which our program plans to implement in the future.
2. Safe deleted the instance variables cusineType, difficultyScale, and preparationDuration and their relative methods as those variables serve for future functionalities we didn't implement.

Other Refactoring

1. Some of the classes are renamed (UserManagerLoginAndNewUser is refactored to just UserLoginManager).
2. Renamed some of the variables and methods to make sure camel case consistency
3. Renamed a couple of methods in UserShoppingListManager from addIngredient and removeIngredient to addItem and removeItem to distinguish from the addIngredient and removeIngredient methods in UserFridgeManager and hence, avoid confusion.
4. Renamed some variables to better reflect their purpose. For example, containStatus was renamed to removeStatus since it is associated with and changes values only when the user removes an ingredient from their shopping list.
5. In the User entity class, the instance variables username, fridge, and shopping list were public. But out of encapsulation considerations from other parts of our program and from other programmers, those variables are made private and corresponding getter and setter methods are defined.

Accessibility

- For each Principle of Universal Design, write 2-5 sentences or point form notes explaining which features your program adheres to that principle. If you do not have any such features you can either:

  (a) Describe features that you could implement in the future that would adhere to principle or

  (b) Explain why the principle does not apply to a program like yours.
  - Principle 1: Equitable Use

    The program is suitable for everyone who has an interest in cooking or who wants a more convenient way to organize and keep track of their cooking ingredients.

    This program is capable of loading up to 7000 recipes, so it is equally suitable for users who have different cuisine preferences and tastes.

    The login and create user functionality which involves setting a username and password provides privacy, security, and safety should be equally available to all users.

    The user interface has high contrasting themes and a decent font size, thus it is suitable for the majority of users including users with minor vision impairment. But it is unsuitable for users with major vision impairment as there’s no audible implementation to this program.

- Principle 2: Flexibility in Use

  The functionality of this program can act independently. In specific, a user can use the add/remove an ingredient from the fridge, or add/remove ingredient to the shopping list, or search up recipes details independently from other functionality. Hence, a user has the flexibility to choose their preferred method of using this program depending on their need.

- Principle 3: Simple and Intuitive use

  A user can simply interact with the program by typing in simple words in the prompted textboxes or clicking on the prompted clickable buttons, hence the design is easy to understand regardless of the user's experience, knowledge, language skills, or current concentration level. In addition, the background image of the user interfaces (for example there is a display of a refrigerator when a user is viewing his/her fridge) is aesthetically pleasant, relaxing, and it further enhances the intuition and confirms the user’s expectations of the program.

- Principle 4: Perceptible Information

  The program communicates necessary information effectively as there are minimal amounts of words in each functionality scene (Except for one which is the recipe details scene since words are necessary to provide adequate information for a recipe). Each functionality scene display conveys its proper usage through a main functionality title label, and those functionality title labels are confirmed by the labels on clickable buttons and the background image of the scene hence achieving redundant presentation of essential information. The high contrasting theme between the words, clickable, and the background image provided an easily perceptible environment for presenting essential information, and the adequate text size secures the legibility of information as well. Furthermore, after a user performs an action (such as add ingredient to the fridge or add ingredient to the wish list), there will be a popup box confirming the action thus the user is able to perceive the action is performed successfully. In the future, it is under our consideration to implement this program for different platforms (such as android) so it provides compatibility with a variety of techniques or devices for the users.

- Principle 5: Tolerance for error

  The cooked functionality is where the most unwanted error can take place in this program as this functionality automatically removes all the

ingredients associated with the dish cooked from the user's fridge. So in preventing such hazards, there will be a popup alert box to double confirm whether the user wants to perform this action, hence minimizing accidental unintended actions. If cooked unintended actions still happen nevertheless, the program provides a fail-safe feature as a user can simply add those ingredients back to their fridge. Similar case to adding the wrong ingredient to the shopping list or fridge as a user can simply remove the falsely added ingredient.

- Principle 6: Low physical effort

  The program requires low physical effort as the interaction with the program can be done by typing in simple words in the prompted textboxes or clicking on the prompted clickable buttons. Furthermore, the cooking functionality assumes the user cooked the dish hence automatically removing the related ingredients in a user's fridge thus minimizing repetitive actions. Similar repetitive action elimination functionality exists for the shopping list. As a user adds the ingredient to the fridge, that ingredient gets removed from the shopping list as the program assumes the user has obtained the item. Lastly, the find recipe menu displays the list of recipes available given a user's fridge ingredient, hence dramatically reducing the effort of checking each recipe to find a dish he/she can make. In the future, if possible, photo scanning ingredients could be a nice way to reduce physical effort as inputting every single ingredient a user got from a grocery isn't an effortless thing to do.

- Principle 7: Size and Space for Approach and Use

  Not applicable.

- Write a paragraph about who you would market your program towards, if you were to sell or license your program to customers. This could be a specific category such as "students" or more vague, such as "people who like games". Try to give a bit more detail along with the category.
  - Very generally, anyone who cooks food for themselves would be good to market our program towards. Whether they are beginner cooks trying to organize their time in the kitchen, or more experienced cooks who are already good with recipes, our program offers value in the way that it allows them to record exactly what ingredients they have. More specifically, our program would also be good for people who want to begin cooking but don't really know where to start. The way our program gives users a list of recipes they can cook based on their ingredients makes it so that a new cook can simply look through that list of recipes and select one that sounds good, without needing to

have any prior information about the recipe.

- Write a paragraph about whether or not your program is less likely to be used by certain demographics. For example, a program that converts txt files to files that can be printed by a braille printer are less likely to be used by people who do not read braille.
    - People who don’t always cook are less likely to use our project, since our project is facing all people who want to manage their fridge and cook food. Furthermore, our project hasn't done any functionalities for the disablity, so people who are blind or unable to type may have a problem running our project.
    -

Progress Report

Open questions your group is struggling with:

Our GUI can’t hold strings that are too large. We are considering how to create a scroll so our GUI can store more information so that we can have recipes that display longer directions to users.

Our git action workflow has not been operating as there was an error in a test with gradle which we couldn’t resolve by visiting office hours or changing to a different gradle.yml file.

Our program can only match the exact name of the recipe and the exact name of the ingredient. So if a user input a recipe that he/she is looking for that doesn't match the exact recipe name, the desired result won’t come out. We struggle to find a way to output similar results to a user’s input even if the input isn’t exactly the one the program is looking for.

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 our 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 (opposed to saving them in for example a txt file). We have benefited from the observer design pattern because of the simplicity of extending new functionality relating to the user.

A summary of what each group member has been working on since Phase 1

Rongguan: In this phase, I have set up the JavaFX project and I have designed the login, creating new user, add ingredient, find recipe, display recipe directions, view fridge and logout graphical user interface and their relating styling. Furthermore, I have implemented the model view controller pattern, the controller and presenter class and the their methods that corresponds to the gui functionalities I implemented. In addition, I have added a new gateway package and the GetSaveUserList gateway class to access the database. I have modified the login and createNewUser method in UserLoginManager and I created a data access interface called ReadWriter to protect the clean architecture structure when UserLoginManager is interacting with database userList.ser. Also I implemented the logout method in which the operations of a user from the current session would be saved to the SER file through serialization when he/she logs out. Moreover, I have implemented the cooked method in UserFridgeManager in which the program will automatically remove all the ingredients associated with the dish from the user's fridge. Lastly, I contributed to Recipe finder and other classes to make sure different parts of the program run smoothly together and I wrote corresponding tests to all the codes I have implemented.
https://github.com/CSC207-UofT/course-project-group-069/pull/16

https://github.com/CSC207-UofT/course-project-group-069/pull/9

Adam: In this phase of the project I finalized all of the data we need and organized it into a database for our program to pull from. I cleaned the data and made sure it suited our program's capabilities. I implemented the code related to handling the reading of our database. I made the data access interface ReaderGateway which preserves clean architecture while RecipeCSV reads from the database containing all the recipes. In the RecipeCSV class, I implemented the method from the interface ReaderGateway called getRecipes that used a BufferedReader and some String and ArrayList functionalities to correctly parse the recipes of our database and instantiate a new Recipe object for each. I also wrote code for the Recipe.java and RecipeFacade.java classes as well. The Recipe class didn't really match what we needed when I was implementing the code to handle the database, so some changes were made to adjust for that. RecipeFacade originally had hard-coded recipes that were used as our list of recipes to pull from. I changed it so that it instead instantiates a new RecipeCSV object that lets our program get all the recipes it needs from our database via the getRecipes method. I was also involved in some refactoring.

https://github.com/CSC207-UofT/course-project-group-069/pull/20

https://github.com/CSC207-UofT/course-project-group-069/pull/25/

https://github.com/CSC207-UofT/course-project-group-069/pull/37

Daniel: I have changed some functionalities of RecipeFacade, Recipe and UserFridgeManager. In detail, I've changed the Find method, Find method now returns a list of recipes in String format, which all user's fridge are available for the list of recipes. New methods called showSimple and showDetail are added in our Recipe class to satisfy our GUI, the first one returns a string that includes the recipe name and its ingredient, the second one returns a string that includes all the details about the recipe which is used frequently in our use cases. Print methods are adjusted into two methods called printSimple and printDetail which are similar to showSimple and showDetail. Cook method added in our UserFridgeManager which can delete the items that user cooked from users' fridges. In phase2, most of the work I did is about entities and use cases. I am trying to adjust our basic codes for the GUI. Then, I refactored what I have written since phase 0, and the codes are simpler and cleaner, basically make some changes to methods that have the long methods, long parameters, and delete the duplicate codes. Same for tests, I wrote tests for some entity classes and use case classes and refactored them by deleting some duplicate codes. Lastly, I added doc to all codes I've completed. In the future, if I am still going to improve this project, I will try to implement methods to let users create recipes into our database. This change can make us rely less on online resources, and be more flexible with our database, so I think that's very important functionality.

https://github.com/CSC207-UofT/course-project-group-069/pull/35

https://github.com/CSC207-UofT/course-project-group-069/pull/34

https://github.com/CSC207-UofT/course-project-group-069/pull/32

https://github.com/CSC207-UofT/course-project-group-069/pull/29

Arya: For Phase 2, I was mostly responsible for implementing the Shopping List functionality as I came up with the idea to implement this. Specifically, I modified the User entity to hold this shopping list and created a new use case class called UserShoppingListManager. For this class, I implemented an observer design pattern to stay consistent with the other two User use case classes. Next, I modified the controller and presenter classes, and added to the GUI in order to display this functionality. Specifically, I created a new window in the GUI and gave options to the User to add and remove ingredients from the list while viewing it. I also added to the Fridge functionality so that when the user adds an ingredient to their fridge that is present in the user's shopping list, it automatically deletes that ingredient from the list. Finally, I added comprehensive tests, not only for my classes, but also

helped teammates by writing other tests. In the end, I performed some refactoring for the User entity due to encapsulation considerations. I also refactored various methods and variables in order to avoid confusion and improve understanding. Lastly, I changed the program to solve all the IntelliJ warnings.

Unfortunately, I ended up pushing my biggest commit directly to the main branch instead of creating a pull request and merging. But this pull request was one of my other big contributions since I added more functionalities to my classes, performed refactoring and helped my team by writing tests, and getting rid of small bugs and warnings: https://github.com/CSC207-UofT/course-project-group-069/pull/38

Antony: