GROUP NAME: Group 3 Diet Manager V1 – Rubric

SWEN.383 SW Design Principles and Patterns

Final Java Solution for Diet Manager V1

|  |  |  |  |
| --- | --- | --- | --- |
| **Level** | **Tasks** | **Points** | **Score** |
| ☐ 1 | Load a foods.csv file with only basic foods and an empty log.csv file. User can see an empty log for today with default calories and weight. User can view the basic foods loaded.  TODO: Explain what you did here and where to find this feature in your code.  At application startup, the system loads all available basic foods from the foods.csv file and displays them in the interface. This is done through the Controller constructor, which calls foods.loadFromFile("assets/data/foods.csv"). The Foods model then delegates the file reading to the FileHandler.readFoods() method. During this process, all basic food items are parsed and loaded into the model.  Simultaneously, the controller attempts to load any existing log data using logs.readLogs("assets/data/log.csv", foods.getAllFoods()), and if the log file is empty, it safely defaults to showing an empty log for the current day. The view is updated with View.updateFoodList() to display the food list, and View.updateLogList("") to clear the log area. Nutritional stats default to 0 and are shown using View.updateStats(0, 0, 0, 0). This initialization sequence can be found in the Controller constructor. | (60) |  |
| ☐ 2 | Level 1 PLUS Select a basic food(s) for the daily intake. The selection is stored in the database (log.csv) the log view is updated with the dietary information about the nutrients consumed.  TODO: Explain what you did here and where to find this feature in your code.  The application allows users to select one or more basic foods and add them to the current day’s log. When the user clicks “Add Log Entry”, the AddLogButtonListener is triggered. This listener retrieves the selected food names using View.getSelectedFoodNames() and uses Foods.findFoodByName() to convert those names into Food objects.  The system then prompts the user for serving sizes via View.promptForServingsForMultipleFoods() and creates a Log object for each item using the selected date (fetched from Controller.getCurrentDate()). These logs are added to the model using Logs.addLog(). The view is then updated using View.updateLogList() to reflect the day’s log, and nutritional stats are recalculated using the Logs model's daily methods (like getTotalCaloriesForDate()) and displayed via View.updateStats(). | +10 (70) |  |
| ☐ 3 | Level 2 PLUS Add new basic food(s) to the food database. This implies the ability to then add such basic food(s) to today's log as in level 2  TODO: Explain what you did here and where to find this feature in your code.  Users can add new basic food items to the food database during runtime. This functionality is initiated when the user clicks the “Add New Food” button. The event is handled by AddFoodButtonListener, which first prompts the user to choose the type of food via View.promptForType(). If "Basic Food" is selected, the system displays a form through View.promptForBasicFood(), where the user can input name and nutrient values.  The resulting BasicFood object is passed to Foods.addFood(), which appends the food to the internal list and saves the updated data to foods.csv via saveToFile(). Once added, the food appears in the GUI, thanks to a call to View.updateFoodList(). This allows the user to immediately select and log the newly created item in the same session, extending the dynamic flexibility of the app. | +5 (75) |  |
| ☐ 4 | Level 3 PLUS Loading and viewing a foods.csv file with recipes.  TODO: Explain what you did here and where to find this feature in your code.  The application supports loading recipes from foods.csv, alongside basic foods. This is handled using a two-pass strategy implemented in the FileHandler.readFoods() method. In the first pass, all basic foods are loaded and added to the model. In the second pass, lines starting with "r," are interpreted as recipes. Each recipe contains a list of ingredient names and serving amounts.  Recipes are reconstructed by matching ingredient names with already-loaded Food objects from the first pass. A Recipe object is created and populated with ingredients using recipe.add(food, amount), and the complete recipe is added to the model. This approach ensures that dependencies are resolved in order and that composite foods (recipes) are fully functional and visible when the program launches. | +5 (80) |  |
| ☐ 5 | Level 4 PLUS Select recipe(s) as well as basic food(s) for the daily intake. The selection is stored in the database (log.csv) and the log view is updated with the dietary information about the nutrients consumed.  TODO: Explain what you did here and where to find this feature in your code.  The system allows users to log both basic foods and recipes in the same way. This is possible because both BasicFood and Recipe extend from the abstract Food class, following the Composite design pattern. This enables uniform treatment of both types during logging and nutrient calculation.  When the user selects multiple foods (basic or recipe), the AddLogButtonListener does not need to differentiate between them. For each food, a Log object is created with the selected servings and date, and is added using Logs.addLog(). During stats calculation, each recipe recursively calculates its total calories, fat, carbs, and protein, ensuring that all values displayed to the user are accurate and aggregated properly. The same update flow works seamlessly due to polymorphism. | +5 (85) |  |
| ☐ 6 | Level 5 PLUS add new recipe(s) to the food database. This implies the ability to  then add such recipe(s) to today's log as in level 5.  TODO: Explain what you did here and where to find this feature in your code.  Users can create custom recipes using ingredients from the currently loaded food list. This functionality is accessible through the "Add New Food" button, and is handled in AddFoodButtonListener. When the user selects "Recipe", the system prompts them with a custom form via View.promptForRecipe(), allowing them to choose ingredients and specify serving amounts.  A new Recipe object is constructed with the given name and ingredients, then added to the model using Foods.addFood(), which saves it to foods.csv. Since recipes are just Food objects, they immediately appear in the list and can be logged alongside basic foods. This enables powerful composition and reusability of existing food items, with recursive nutrition calculations handled in the Recipe class | +5 (90) |  |
| ☐ 7 | Level 6 PLUS the ability to read a non-empty log.csv file, to navigate to  different days in the log, and to select foods for the intake for the days other than today.  TODO: Explain what you did here and where to find this feature in your code.  The system loads previously logged entries from log.csv during application startup and displays the logs for the current date. This is handled in Controller.java by calling Logs.readLogs() and adding those entries to the in-memory model using addAllLogs(). The user can also change the viewed date using the "Change Date" button, which is handled by ChangeDateButtonListener.  When a new date is selected, the controller updates its internal state via Controller.setCurrentDate() and refreshes the logs and nutrition stats for that date. This allows users to view and add logs for any date, past or future. All log calculations are done per-date using filtering methods in Logs.java such as getLogForDate() and getTotalCaloriesForDate(). | +5 (95) |  |
| ☐ 8 | Level 7 PLUS the ability to save the log and food database back to the log.csv and foods.csv files.  TODO: Explain what you did here and where to find this feature in your code.  All updates to the food list and log entries are saved persistently to foods.csv and log.csv. The Foods model automatically calls saveToFile() whenever a new food is added. Similarly, the AddLogButtonListener explicitly calls Logs.saveLogsToFile() after logging, ensuring that all entries are written to disk.  File operations are handled in FileHandler.java, which includes proper formatting and serialization for both food and log data. This ensures data integrity across sessions, allowing users to close and reopen the app without losing any information. This final layer of persistence completes the full workflow, supporting both read and write operations for all data types. | +5 (100) |  |
| SUBTOTAL: | | **100** |  |

To receive any credit for level N, the preceding levels must be sufficiently functional to test level N. In general, this means previous levels must work without failure when the user enters normal (non-error) data.

Remember, the solution should apply design patterns as required. The final grade will be adjusted based on how good you have applied the pattern. You may lose up to 25% of the final project grade in this regard.