**Documentation**

**Project Folder Structure**

The folder that hosts all the code for the project is called NewProjectJustDropped. Initially, the root of NewProjectJustDropped (NPJD) contained the recipeProject.py (this hosts the code for RecipeManager class) script and tests folder. It was later found that this folder structure was unpleasant for executing tests and imports so the folder structure was overhauled. Now, NPJD consists of a folder called “Core”, a folder called, “tests”, a folder called “IntegrationTests” and a folder called “ShellIntegrationTests”. Core hosts recipeProject.py, rm\_shell.py and app.py. These files will be discussed in detail below. The tests and IntegrationTests hold the unit tests and integration tests for methods of RecipeManager class, respectively. ShellIntegrationTests hold the unit tests for the command shell implementation.

**RecipeManager Class (recipeProject.py)**

RecipeManager class is the backbone of the recipe management system. It consists of the methods that carry out main CRUD operations, as well as export and import operations.

The data structure that acts as the memory and holds all of the system’s information is in the form list of recipe objects. Each recipe object has the following structure:

**Recipe Class:**

1. ID
   1. Datatype - int
2. Recipe name
   1. Datatype - string
3. Recipe Author
   1. Datatype - string
4. Prep Time
   1. Datatype - int
5. Cook Time
   1. Datatype - int
6. Serving Size
   1. Datatype - int
7. Ingredients
   1. Data structure – List of dictionaries
      1. {ingredientName: \*\*\*, measuerement: \*\*\*, quantity: \*\*\*}
8. Instructions
   1. Data type – dictionary
      1. {1:” loremipsum”,2:” loremipsum”}

**Methods:**

1. **View Recipes**
   1. **View all -** This method is responsible for displaying all the recipes stored in the memory.
   2. **View by ID – This method displays a specific recipe based on the id provided.**
2. **Add Recipe – The add method allows the user to add a recipe with all the details to the memory.**
3. **Edit Recipe – This method** handles the editing of a specific recipe using the id of the recipe.
4. **Delete Recipe – This method prompts the user to confirm deletion of a specific recipe using the ID of the recipe.**
5. **Import Recipes**
   1. **Imports a JSON file based on the filename provided. The folder that it imports the recipes from is called JSONFiles.**
   2. **If the imported JSON file is imported then it is handled accordingly and user is informed.**
6. **Export Recipes – Exports all of the recipes in the memory to a JSON file. Name of the file depends on what the user has provided. If no name is provided, a default name is used instead. Exported JSON files are also saved to JSONFiles folder.**

**Unit Tests and Integration Tests**

Each method of the RecipeManager Class has a dedicated unit test. Each method is also tested with other methods to form the integration tests. The unit tests and integration tests have a dedicated folder in the root of the project.

The “tests” folder contains all of the unit tests for the project. All tests have the class TestRecipeManagement(unittest.TestCase) that define the unit tests for the RecipeManager class in the recipeProject module.

**File name - testViewRecipe.py:**

The following method is present in the file:

* test\_view\_recipe(self): A test method that verifies the behavior of viewing all the recipes in the RecipeManager class.

**File name - testViewSpecificRecipe.py:**

The following are the methods that are present in the file:

* setUp(self): A method that is executed before each test method. It creates an instance of the RecipeManager class for testing.
* test\_viewSpecificRecipe\_existingID(self): A test method that verifies the behavior of viewing a specific recipe based on id from the RecipeManager class.
* test\_viewSpecificRecipe\_nonexistingID(self): A test method that verifies the behavior of viewing a non-existent recipe with id from the RecipeManager class.

**File name - testAddRecipe.py**

The following are the methods that are present in the file:

* test\_add\_recipe(self): A test method that verifies the behavior of adding a new recipe with all details in the RecipeManager class.
* test\_add\_recipe\_wrongID(self): A test method that verifies the behavior of adding a new recipe with a used id in the RecipeManager class.
* test\_add\_recipe\_wrongName(self): A test method that verifies the behavior of adding a new recipe with a used name in the RecipeManager class.

**File name - testEditRecipe.py:**

The following are the methods that are present in the file:

* setUp(self): A method that is executed before each test method. It creates an instance of the RecipeManager class for testing.
* test\_editRecipe\_existingRecipe(self): A test method that verifies the behavior of editing an existing recipe in the RecipeManager class.
* test\_editRecipe\_nonExistingRecipe(self): A test method that verifies the behavior of editing a non-existing recipe in the RecipeManager class.
* test\_editRecipe\_noChanges(self): A test method that verifies the behavior of editing a recipe without making any changes in the RecipeManager class.
* test\_editRecipe\_emptyData(self): A test method that verifies the behavior of editing a recipe when the data is empty in the RecipeManager class.

**File name - testDeleteRecipe.py:**

The following are the methods that are present in the file:

* test\_delete\_recipe(self): This test verifies the basic functionality of the deleteRecipe method by deleting a recipe with an ID and checking if the recipe is deleted from the collection.
* test\_delete\_recipe\_NON\_EXISTENT\_ID(self): This test ensures that the deleteRecipe method handles the deletion of a non-existent recipe ID correctly.
* test\_delete\_recipe\_STRING\_ID(self): This test validates the behavior of the deleteRecipe method when a recipe with a string ID is attempted to be deleted.

**File name - testImportRecipe.py:**

The following are the methods that are present in the file:

* test\_import\_NoFile(self): A test method that verifies the behavior of importing an invalid file in the RecipeManager class.
* test\_import\_recipes(self): A test method that verifies the behavior of importing recipes from a file in the RecipeManager class.

**File name testExportRecipe.py:**

The following are the methods that are present in the file:

* test\_exported\_JSON\_file(self): A test method that verifies the behavior of exporting recipes to a file.
* test\_noNameExport(self): A test method that verifies the behavior of exporting recipes to a file with no name.

The “IntegrationTests” folder contains all of the integration tests for the project. All tests have the class TestRecipeManagerIntegration(unittest.TestCase) that defines the integration tests for the RecipeManager class in the recipeProject module.

**File name - testAddDelete.py:**

The following are the methods that are present in the file:

* test\_add\_and\_delete\_recipe(self): A test method that verifies the integration between the addRecipe and deleteRecipe methods in the RecipeManager class.
* test\_add\_and\_delete\_recipe\_RECENTLY\_DELETED\_ID(self): A test method that verifies whether the addRecipe method correctly adds a recipe with a previously deleted ID to the data.

**File name - testAddEdit.py:**

The following are the methods that are present in the file:

* setUp(self): A method that is executed before each test method. It creates an instance of the RecipeManager class for testing.
* test\_addRecipe\_and\_editRecipe(self): A test method that verifies the integration between the addRecipe and editRecipe methods in the RecipeManager class.

**File name - testAddViewSpecificRecipe.py:**

The following method is present in the file:

* test\_add\_and\_view\_specific\_recipe(self): A test method that verifies the integration between the addRecipe and viewSpecificRecipe methods in the RecipeManager class.

**File name - testCRUD.py:**

The following method is present in the file:

* test\_CRUD(self): A test method that verifies the integration between the addRecipe, viewRecipeList, editRecipe, and deleteRecipe methods in the RecipeManager class.

**File name - testImportExport.py:**

The following method is present in the file:

* test\_(self): A test method that verifies the integration between the importRecipes, editRecipe, deleteRecipe, and exportRecipes methods in the RecipeManager class.

CircleCI was implemented in order to verify to verify that the code pushed by developers was working appropriately before being pushed to main branch. Along with CircleCI, a dedicated code reviewer was also assigned as another strong measure for branch protection. This allowed the detection of errors and fixes in a more efficient manner.

**Command Shell**

A custom command shell was created for the RecipeManager class. The command shell is activated by running the rm\_shell.py script found inside Core. The command shell is implemented using the “cmd” built-in python framework. The framework allows for a loop to initiate and continuously receive input from a user. The user knows that the shell is active because when activated, the command line displays “**RmMode>“** instead of the usual text cursor in a command line terminal.

Instead of manually implementing a feature that mimics a shell by initiating a for loop, the cmd module includes features that we do not need to manually implement. It can handle invalid commands and generate custom help texts.

The command shell implementation consists of 9 major commands. Each command also has its own help text.

1. View
   1. View Recipe List – Command: view all
   2. View specific Recipe – command: view <id>
   3. View Recipe List sorted by recipe name or authors – command: view sort <recipe\_name>or <recipe\_author>
2. Add
   1. Add a recipe – Command: add
3. Edit
   1. Edit an existing recipe – Command: edit <id>
4. Delete
   1. Delete an existing recipe – Command: delete <id>
5. Export
   1. Export all recipes to JSONFiles folder – Command: export <filename>
6. Import
   1. Import recipes through a JSON file from JSONFiles folder. – Command: import <filename>
7. GUI
   1. Activates or deactivates the GUI based on the argument provided.
      1. Command – gui activate
      2. Command – gui deactivate
8. Clear Memory
   1. Clear the memory of the recipe manager - Command: clear
9. Exit
   1. Exits RmMode: Command - exit
10. Help (not a system command)
    1. Prints instruction manual for the specific command to the screen. If no command provided then it prints the commands available.
    2. Command – help <command\_name>
    3. Command - help

**Test plan for Command Shell and GUI**

Unlike the recipe manager class, the command shell does not have extensive unit testing and integration testing. The shell unit test folder houses only 2 tests and these are unit tests for the add command and exit command. They strictly test if the command shell is working as expected.

Testing for the command shell was done using exploratory methods. Reason for this decision is due to time constraints and the convolution that comes with writing tests for the command shell as it constantly requires input from the user, making it vastly different from usual unit tests.

Hence, testing for command shell follows a mostly exploratory fashion.

The GUI implementation also shares a similar reason for why it has no unit tests or integration tests. GUI also followed an exploratory testing scheme.

Since the GUI and command shell lack dedicated unit tests and integration tests, there was a stronger emphasis on exploratory testing for these and these implementations still went through vigorous testing. Moreover, the Recipe Manager class consists of multiple unit tests and integration tests that should cover all aspects of the core functionality of the recipe manager.

**GUI Implementation**

The GUI implementation for our Recipe Management system is done using flask. From the command shell, the gui command launches the GUI in a chrome tab (assuming chrome is installed on the target computer). The GUI shares the same memory as the recipe manager that was launched in the command shell. Because of this, when the GUI is active, certain commands are blocked from the command shell until the GUI is deactivated. This is to avoid potential race conditions. The command shell actions that get blocked are Add, Edit, Delete, Import and Export.

The gui command launches the chrome tab with the use of Selenium package. Once launched, the GUI can be shutdown from within the GUI or from the command shell itself. The GUI shares almost all of the same features as the command shell implementation.

Some methods needed to be re-written to be implemented in the GUI. The methods were not re-written but new methods were made with the name GUI attached to it to distinguish and mark the fact that this method is used for the GUI.

The homepage loads all the recipe using a newly made view recipes method which returns the list of recipes from memory.

A difference that can be seen in the GUI implementation is that users have more sorting options. Users can sort the recipes by ID, Name, Author, Total Time and Serving size. Sorting can also be done in ascending and descending order. Users can sort the recipes by clicking on the title that they want to sort by.

There is a new attribute called “Total Time” which is not seen in the main data structure and the command shell as well. This total time is calculated from the Prep Time and Cook Time attributes of the recipe objects. Sorting can be done by Total time as well.

Another difference that can be seen is in the export and import features. Unlike in the command shell version, the GUI version allows the user to save their export, or load their import file from wherever on their computer. The export/import buttons allow this by opening a save as dialog window or a open as dialog window using the Tkinter library for python.

The Exit button in the GUI works by executing the do\_gui command of the active command shell. It is programmed to send the “gui deactivate” command to the command shell.

**The following are the methods present in the app.py file:**

* **def home(): Displays all the recipes in the index page.**
* **def sortID(): Sorts recipes by id in the table.**
* **def view\_id(id): Displays information regarding a specific recipe.**
* **def add(): Adds a new recipe to the memory.**
* **def edit\_id(id): Edits an existing recipe.**
* **def delete\_id(id): Selects the recipe for deletion.**
* **def deleteConfirm\_id(id): Deletes the recipe selected after confirmation is received.**
* **def sortRecipeName(): Sorts recipes by name in the table.**
* **def sortRecipeAuthor(): Sorts recipes by author in the table.**
* **def sortRecipeTotalTime(): Sorts recipes by total time in the table.**
* **def sortRecipeServingSize(): Sorts recipes by serving size in the table.**
* **def import\_file(): Imports recipes from a file.**
* **def export(): Exports recipe list to a file.**
* **def exitGUI(): Quits the program.**

**All of the methods are linked to different html pages that are present in the templates folder with the main Core folder. The templates folder contains:**

* **index.html – The main page that displays a table containing all the recipes and the option to either view (more details), add a new recipe, edit an existing recipe, delete a recipe, sort recipes in the table, or exit the application**
* **add.html – Allows the user to add a new recipe after all fields are entered.**
* **edit.html – Edit page allows the user to edit a recipe.**
* **viewID.html – This page shows more information regarding a specific recipe.**

**Conclusion**

This documentation provides a comprehensive overview of the project's folder structure, the RecipeManager class, unit tests, integration tests, command shell implementation, GUI implementation, and the testing approach used for the command shell and GUI.

The project's folder structure was organized to improve the execution of tests and imports. It consists of folders such as Core, tests, IntegrationTests, and ShellIntegrationTests. The Core folder contains the main script files, including recipeProject.py, rm\_shell.py, and app.py.

The RecipeManager class serves as the backbone of the recipe management system. It handles CRUD operations, as well as export and import functionalities. The class uses a list of recipe objects as the data structure, and each recipe object contains various attributes such as ID, recipe name, recipe author, prep time, cook time, serving size, ingredients, and instructions.

The unit tests and integration tests were developed to ensure the proper functioning of the RecipeManager class. Each method in the class has its dedicated unit test, while integration tests verify the interaction between multiple methods. The tests cover various scenarios, including viewing recipes, adding, editing, and deleting recipes, importing and exporting recipes, among others.

The command shell implementation provides a user-friendly interface for interacting with the RecipeManager class. It utilizes the "cmd" built-in Python framework, allowing users to execute commands such as viewing recipes, adding, editing, and deleting recipes, importing and exporting recipes, and more. The shell implementation also includes a help command for displaying instructions and a GUI command to activate or deactivate the graphical user interface.

The GUI implementation, built using Flask, enhances the user experience by providing a web-based interface for recipe management. It shares the same memory as the command shell, ensuring data consistency. Users can view, add, edit, and delete recipes, as well as sort them based on different criteria. The GUI also supports importing and exporting recipes, and users can save or load files from their local systems.

The testing approach for the command shell and GUI primarily relied on exploratory testing due to the nature of user interaction. While unit tests and integration tests were limited for these components, rigorous exploratory testing was performed to ensure their functionality and usability.