Software Engineering Project 2024

Background:

Software Engineering has been an important field in the world for the past many decades. It is a powerful tool to manage big project development and management.

This project is about a mobile application that :

* Keep up the record of your Fridge items.
* Send you reminders when your items are about to expire.
* Helps you figure out what you can cook with the items you have.
* Tells you the nutrients of the food that you can cook.

Objective:

The objective of the project were the following:

1. Learning how collaborative development happens.
2. Learning new tech stacks.
3. Building integrated systems and managing them.
4. Managing Databases.
5. Disciplined work flow.

Team Members:

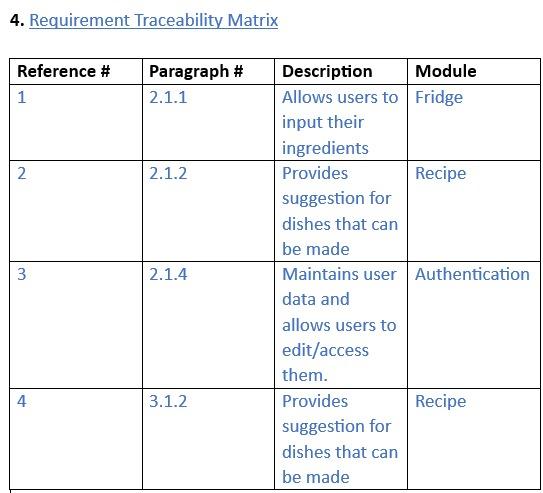
1. Dr. [Romi Banerjee](mailto:romibanerjee@iitj.ac.in) - Mentor
2. Bhavna - Mentor
3. [SUMEET S PATIL](mailto:sumeetpatil20004@gmail.com)
4. [Tanmay Parashar (B22CS053)](mailto:b22cs053@iitj.ac.in)
5. Mukund Gupta

**The Project**

Software Requirements Specification:

The requirements were taken after considering various factors, but the main factor was that the system should assist users in deciding which dish to cook, not limited to this but to also maintain their inventory. Apart from this, the non functional requirements included ensuring that the suggested dishes are correct , UI is responsive, user data is secured,etc.Constraints were taken into consideration before moving ahead with the project.

Traceability Matrix:



Software Lifecycle:

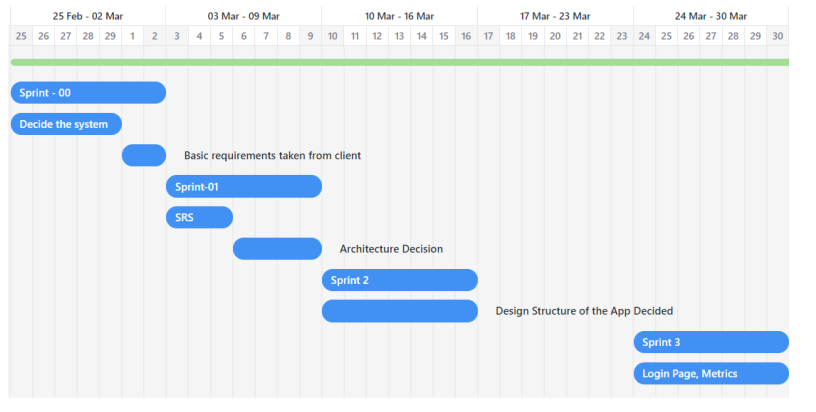
We have adopted the evergreen Agile Development Life cycle to complete the project.

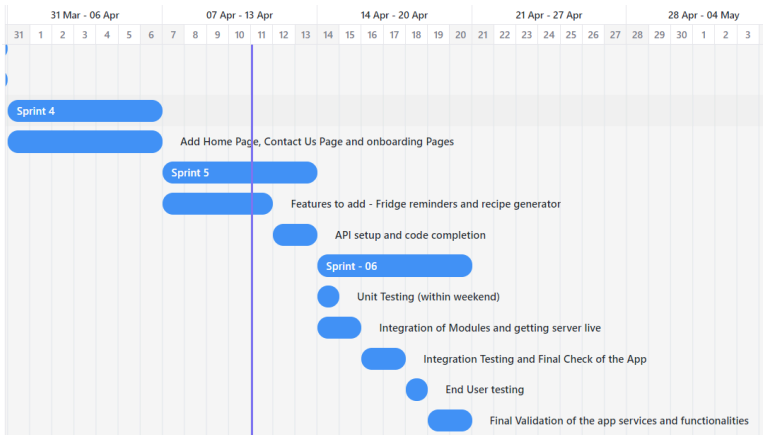
Reason: It helps us reach small goals in short periods of time, which is suitable to the fractal based structure that we have.

We were able to complete 06 Sprints of the life cycle.

The meeting minutes are attached below:

Project Timeline:





Software Architecture

We as a team decided that the main architecture of the software will be Model View Controller architecture.

The mobile application is the model.

The screen with all the buttons is the view that gets updated.

The controller is different for different pages.

Also, since we are calling the server very frequently, we are following a Client-Server architecture. The wsgi acts as a middleware between the client and the server, resulting in a three tier architecture.

The client is identified by a unique id that is generated when a user registers with the application. The encryption method being confidential can not be documented here.

**Version Control system**:

We have used a Distributed Version Control system.

The system:

* There is a central repository, where all the final changes are updated and is the master copy of the application.
* Every team member will create a fork of the central repository.
* They will commit in their central repository.
* Then, finally a pull request will be sent to the central repository, to merge in the code.

Version 0.1.0 - When Login page and onboarding was released

Version 0.2.0 - When Contact us page was released

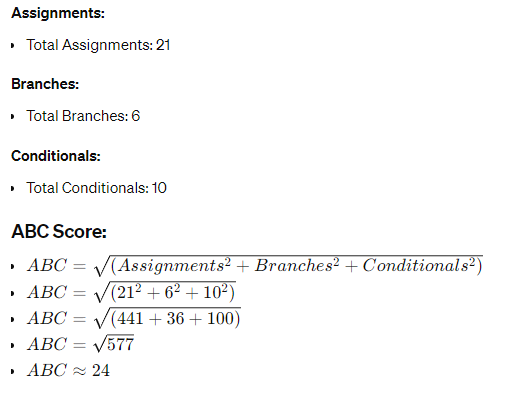
Version 0.2.1 - When Hyper links to github ids were added

Version 1.0.0 When Fridge feature was added

Version 1.1.0 When Recipe feature was added

Version 1.2.0 When Server was deployed

The current version of the application is : 1.2.1 - Local Storage added

Metrics:  
We Have done metrics analysis for login.dart file in authentication module.  
  
We have implemented following metrics:  
a) ABC metrics:  


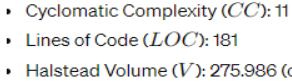
b)Defect Density:  
Weighted Defect Count = (4 \* 1) + (6 \* 2) + (4 \* 3) = 4 + 12 + 12 = 28

Total Size = 2 \* Medium + 3 \* Small + 2 \* Large = 2 \* 2 + 3 \* 1 + 2 \* 3 = 4 + 3 + 6 = 13

Defect Density = Weighted Defect Count / Total Size

Defect Density = 28 / 13 ≈ 2.154

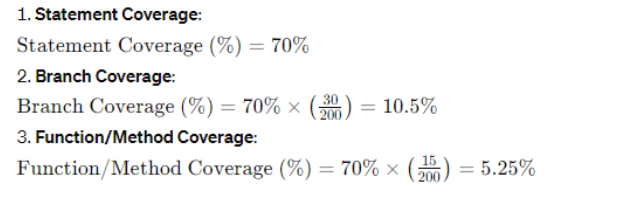
So, the final defect density for this code is approximately 2.154.

c)Maintainability index:  
  


  
D)Cyclomatic complexity (McCabe's complexity):

Total Cyclomatic complexity: 11

E) Code coverage:



Risk Analysis:

Risk 1: Server crash

Risk 2:user data breach

Risk 3:Handling data conflicts

Risk 4:Customer dissatisfaction

Risk 5:App dependencies

**Legends of the Matrix:**

Likelihoods Consequences

1- Unlikely A - Insignificant

2- Seldom B - Marginal

3- Occasional C - Moderate

4- Likely D - Critical

5- Definite E - Catastrophic

|  |  |  | 1 | 5 |
| --- | --- | --- | --- | --- |
|  |  |  |  | 2 |
|  |  | 4 |  |  |
|  | 3 |  |  |  |
|  |  |  |  |  |

1

2

3

4

5

A B C D E

**Color Legends**

| Low risk - No Further Action | Medium risk - Can take action (optional) | High Risk - Further Action Necessary | Extreme Risk- Act Now |
| --- | --- | --- | --- |

Testing:

We have done unit testing and Integration testing.

Unit Testing:

So we have performed various testing on our app modules and its subparts.

**a)login dart testing(authentication module):**

1. Test case checks if the "Create" button is disabled when fields are empty in the login form.

2. Test case verifies that the "Create" button becomes enabled when valid username and email are entered.

3. Test case validates the OTP verification process with a valid OTP, using a mocked SMTP server and OTP verification function.

b)**category widget testing(fridge module):**

1. CategoryCard Widget Test:

- Verifies the UI of the `CategoryCard` widget, ensuring that the category name and the add button are displayed correctly.

- Tests the navigation functionality when the add button is tapped, ensuring it navigates to the correct destination (`TakePicture` widget).

2. Category\_Card Widget Test:

- Tests the UI of the `Category\_Card` widget, checking if the category name and the "Add item" button are displayed correctly.

- Validates the navigation functionality when the "Add item" button is tapped, confirming it navigates to the expected screen (`TakePicture` widget).

**c)fridge.dart file testing:**The test case verifies the rendering of the `Fridge` widget by:

1. Ensuring that the text "My Fridge" is displayed.

2. Confirming the presence of category cards, assuming there are 6 categories displayed.

3. Verifying the text of each category card, including "Frozens", "Dairy", "Fruits", "Vegetables", "Cooked Food", and "Other Groceries".

**d)add category.dart:(fridge module):**

1. addCategory success test:
   * Sets up a mock HTTP client and a CategoryProvider instance.
   * Stubbs the HTTP POST request to return a success response when adding a category.
   * Calls the addCategory method with a test category.
   * Verifies that the HTTP POST request was made with the correct parameters.
   * Asserts that the result of adding the category is true (success).
2. addCategory failure test:
   * Similar setup as the success test but stubbs the HTTP POST request to return a failure response.
   * Calls the addCategory method with a test category.
   * Verifies that the HTTP POST request was made with the correct parameters.
   * Asserts that the result of adding the category is false (failure)

**e)delete item.dart(fridge module):**The test cases in this file focus on testing the UI and functionality of the `DeleteItem` widget:

1. DeleteItem UI Test:

- Renders the `DeleteItem` widget with a test item.

- Verifies that the item details (name, quantity, date) are displayed correctly.

- Ensures the presence of the "Delete item" button.

2. DeleteItem Functionality Test:

- Sets up a mock `ItemProvider` to simulate successful item deletion.

- Builds the `DeleteItem` widget with a test item.

- Taps the "Delete item" button.

- Verifies that the item deletion process is triggered by checking if the `deleteItem` method of the `ItemProvider` mock is called once with the correct item.

Additionally, there's a mock class `ItemProviderMock` defined to mock the `ItemProvider` class for testing purposes.

**f)user add.dart(authentication module):**These tests focus on the `User` class:

1. User toJson() test:

- Creates a `User` instance with a username and email.

- Calls the `toJson()` method to convert the user object to JSON format.

- Asserts that the resulting JSON is a map and contains the correct username and email.

2. User object equality test:

- Creates three `User` instances with different username and email combinations.

- Asserts that `user1` and `user2` are equal because they have the same properties.

- Asserts that `user1` is not equal to `user3` because they have different properties.

These tests verify the serialization of a `User` object to JSON and ensure correct behavior regarding object equality.

**g)know more.dart(contact us module):**The test cases in this file validate the UI of the `know\_more` and `member` widgets:

1. know\_more Widget Test:

- Verifies the presence of specific text widgets such as "About us", "Hello Users!", "User Policies", and "Click on this to see our policies".

- Ensures the existence of the `CarouselSlider` and `DotsIndicator` widgets.

2. member Widget Test:

- Tests the UI elements of the `member` widget, including the display of the member's name, designation, position, and icons for email, GitHub, and Instagram.

These tests ensure that the UI components of both widgets are correctly rendered and displayed as expected.

**h)camera.dart(camera module):**The test cases in this file evaluate the UI and functionality of the `TakePicture` widget:

1. TakePicture UI Test:

- Checks if the camera preview and camera button are displayed correctly.

2. TakePicture Functionality Test:

- Verifies that the `initialize` method of the `CameraController` is called during widget initialization.

- Taps on the camera button and checks if the `takePicture` method of the `CameraController` is called once.

- Simulates a successful image capture and verifies if the response from the API is handled correctly.

Additionally, mock classes `MockCameraDescription` and `MockXFile` are defined to mock the `CameraDescription` and `XFile` classes for testing purposes.

**i)add\_item(fridge module):**The test cases in this file assess the UI and functionality of the `AddItem` widget:

1. AddItem UI Test:

- Checks if the app bar title displays the correct category.

- Verifies if the text fields for item name, quantity, and date are displayed correctly.

- Ensures that the "Add item" button is present.

2. AddItem Add Functionality Test:

- Enters text into the item name, quantity, and date text fields.

- Taps on the "Add item" button.

- Additional verification steps should be added based on your implementation to check if the `AddItem` function was called correctly and if the item was added successfully.

**j)edit item (fridge module):**

The test cases in this file evaluate the UI and functionality of the `EditItem` widget:

1. EditItem UI Test:

- Renders the `EditItem` widget with a test item.

- Verifies that the item details (name, quantity, date) are correctly displayed in the text fields.

- Ensures the presence of the "Save item" button.

2. EditItem Functionality Test:

- Sets up a mock `ItemProvider` to simulate successful item editing.

- Builds the `EditItem` widget with a test item.

- Simulates entering new values in the text fields.

- Taps the "Save item" button.

- Verifies that the item editing process is triggered by checking if the `editItem` method of the `ItemProvider` mock is called once with the correct parameters.

Additionally, there's a mock class `ItemProviderMock` defined to mock the `ItemProvider` class for testing purposes.

**k)settings.dart(fridge):**The test case in this file assesses the rendering of the `Settings` widget:

1. \*\*Settings Widget Rendering Test:\*\*

- Builds the `Settings` widget with a specified category.

- Verifies that the category field displays the correct value.

- Ensures that the reminder field displays the correct value.

- Checks for the presence of the "Save Changes" button.

This test ensures that the UI components of the `Settings` widget are correctly rendered and displayed as expected.

**l)items.dart(fridge module)**:

The test case in this file evaluates the rendering and functionality of the `items` widget:

\*\*items Widget Rendering Test:\*\*

- Creates a `TestItem` widget with sample data.

- Builds the `items` widget with the `TestItem` embedded within a scaffold.

- Verifies that the item name, quantity, and date are displayed correctly.

- Ensures the presence of "Edit item" and "Delete item" buttons.

- Checks the initial state of the "Add to Chef Desk" button.

- Taps the "Add to Chef Desk" button and verifies the text change to "Added to Chef Desk" and the presence of an icon.

The test comprehensively evaluates the UI elements and initial state of the `items` widget, as well as the behavior of the "Add to Chef Desk" button.

**Integration Testing:**

Integration testing was done by running multiple debug sessions.

Debug sessions are provided by Flutter to run the application and integrate many modules together.

Let us now talk about the development aspect of the project:

Tech Stacks used:

1. Flutter - The SDK to develop the application’s User Interface.
2. Microsoft Visual Studio 2022 - The IDE to develop the application.
3. Django and Django rest framework - Helps to maintain and interact with the backend server.
4. Render: It is used to deploy the server live for 90 days.
5. Git - the DVCS
6. Java Script - To create UML Diagrams
7. Kotlin - Native app specifications
8. Google Docs - For Documentation

Now, let us dive into the modules of the project.

We have developed 08 modules each handling either a particular view or service.

1. Authentication

This module deals with the service of registration, encryption and authentication.

1. Contact us

This module handles a single independent view dealing with the app’s introduction and team members.

1. Fridge

This module handles all the services related to Fridge.

1. Home

A stand alone module handling the home page of the application.

1. Onboarding

Has all the onboarding pages

1. Recipe

Deals with all services related to the services

1. Constants

Stores all the constants used in the application

1. Backend

Handles the database deployed at the backend

A few RunTimes:

1. Creation of account - 10 ms (Depends on internet)
2. Encryption - 5 ms
3. Add item - 3 ms (Depends on internet)
4. Delete item - 6 ms (Depends on internet)
5. Edit item - 3 ms (Depends on internet)

Problems Faced:

1. Topic to be chosen
2. SDK deciding
3. Backend handling
4. Server deployment
5. Posting around 6800 recipes onto a single server
6. Integration