-

**Faculty of Computer Science**

**Dalhousie University**

## CSCI 5308 - Quality Assurance

**PROJECT REPORT**

# Recipe App

**Members and Contributors**


**Lakshmi Suresh Kumar Gangumalla**
**B00789128**
**lk737217@dal.ca**

**Naveen Kusakula**
**B00781205**
**nv633449@dal.ca**

**Navneet Prakash Singh**
**B00810744**
**navneet.singh@dal.ca**

**Srisaichand Singamaneni**
**B00792835**
**chandu.singamaneni@dal.ca**

# Table of contents

# Implementation of Continuous Integration

## Tools used

**Server Side Technologies -**

- Java (Spring Boot framework)

**Client Side Technologies -**

- Hyper Text Markup Language (HTML)
- Cascading Style Sheet (CSS)
- Java Server Pages (JSP)

**Database**

- MySQL Server

We used stored procedures, to persist, retrieve, update or delete the data from the database tables.

**Continuous Integration/Continuous Deployment Tool**

- Jenkins

**Project Build tool**

- Maven

**Project Version Control Tool**

- GitHub

**Project Management & Ticket Controlling Tool**

- Trello

We started off by using Jenkins, Github and Azure as a medium for providing continuous integration. We created two branches on Github namely develop, test from production/master(default) branch and mapped Azure environment with these branches namely dev environment, test environment and production environment. Further we mapped the environment with Jenkins. We had three setups on Jenkins such that whenever there is a change in develop branch of the project, Jenkins would pick up the change and run it by checking the unit test cases against the code. If everything was successful, Jenkins deploys the application to Azure.

From the develop branch, feature branches are created by each teammate to work on their feature. When a considerable amount of work is done, the code is pushed to the feature branch and a new pull request is done to merge the feature branch code into develop. The merge is done by any other team member who is not the owner of pull request.

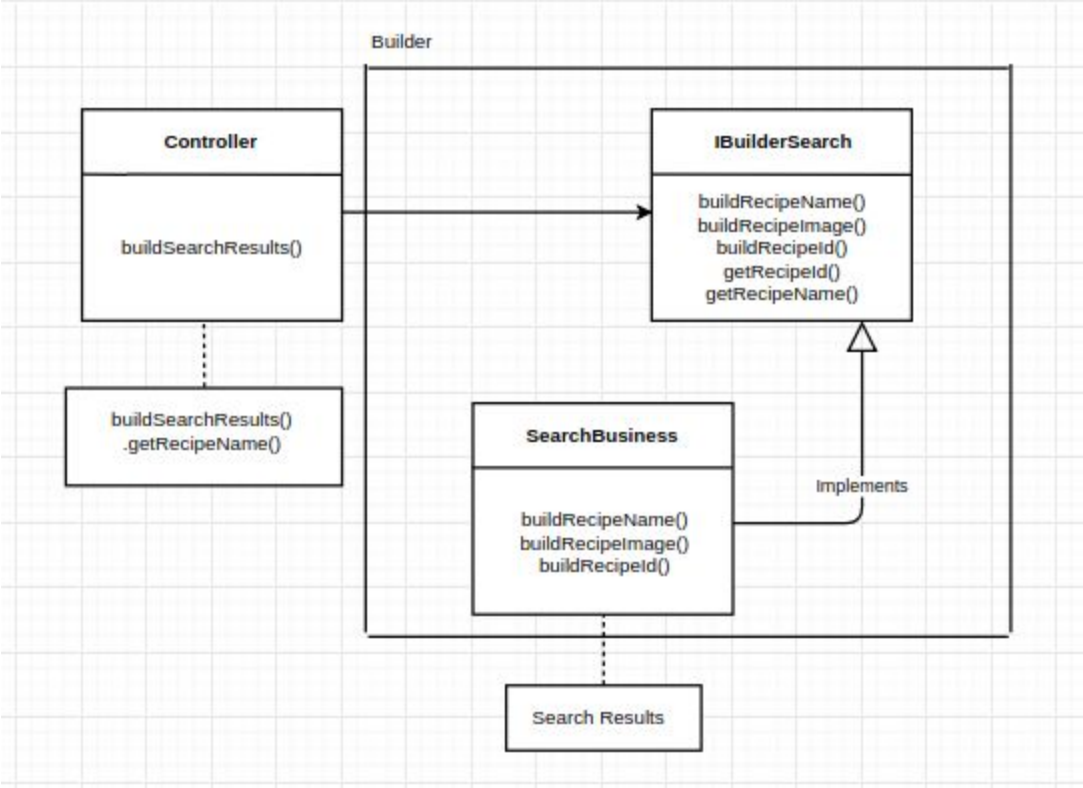
Since Azure has a limitation, we could not connect the database with Azure and hence we shifted onto Heroku to deploy the application. Heroku provided its in house support of continuous integration so whenever a change is made in a branch, heroku observes the change, builds the application and deploys it. In case of any build error, owner of the application is notified through email. Using these tools, we were able to provide continuous integration to our project.

We did not have any hard code values for database, rather it was stored in a config file and spring boot would pick up the config file to access the database.

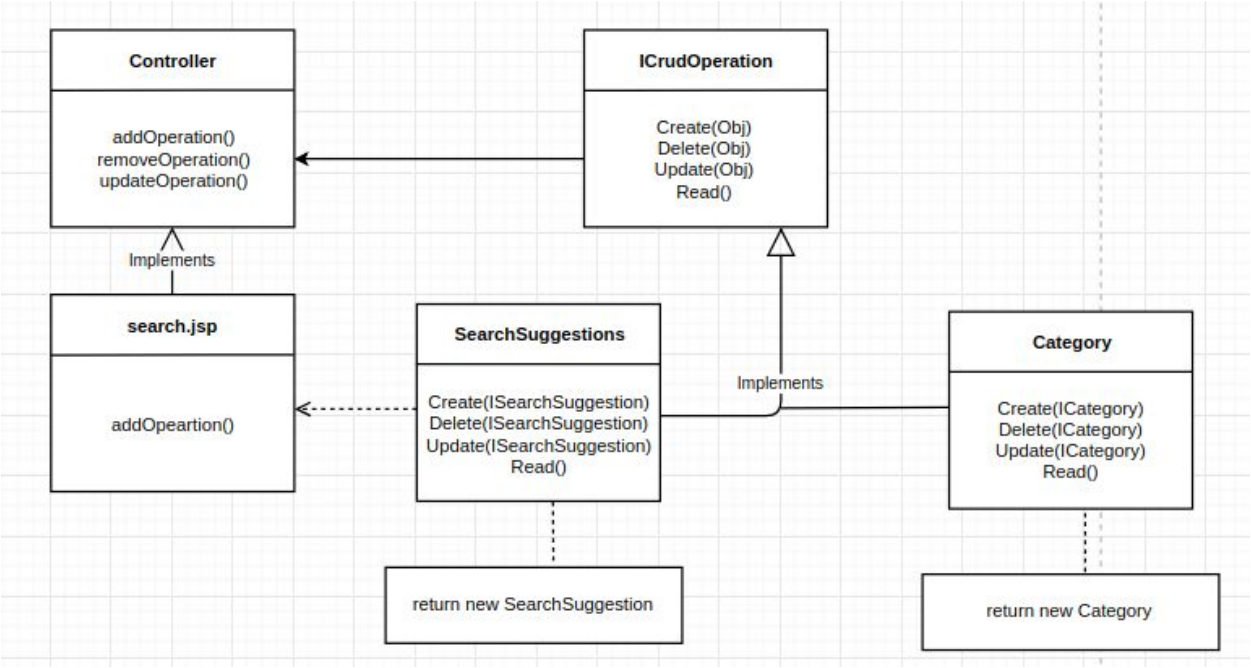
# Design Patterns Used

**Builder pattern**

For the search functionality, Builder pattern has been implemented. The main intent of using builder was to separate the construction of a complex object from its representation so that the same construction process can build different parts of a representation. Let us consider the following UML diagram for search using builder pattern.



Controller would build various parts of the results like recipe name, recipe image and recipe id of the returning result. Now suppose we want to add ratings to the search results, we should be able to do that without modifying the Controller. We can simply add a method declaration in IBuildSearch, making it easy to add a new part of search result without modifying the controller. The second design pattern we implemented was template pattern to implement the CRUD operations by defining the skeleton of an algorithm which are implemented by the subclasses. It allowed us to redefine certain steps of CRUD operation without changing the structure of the main interface. Following is our implementation of CRUD using template pattern.




A template method defines an algorithm in terms of abstract operations that subclasses override to provide concrete behaviour. Within the above diagram, we can see that we have an interface ICrudOpeartion which contains the various operations and the implementation for each operation was done in the subclass to provide the concrete behaviour.

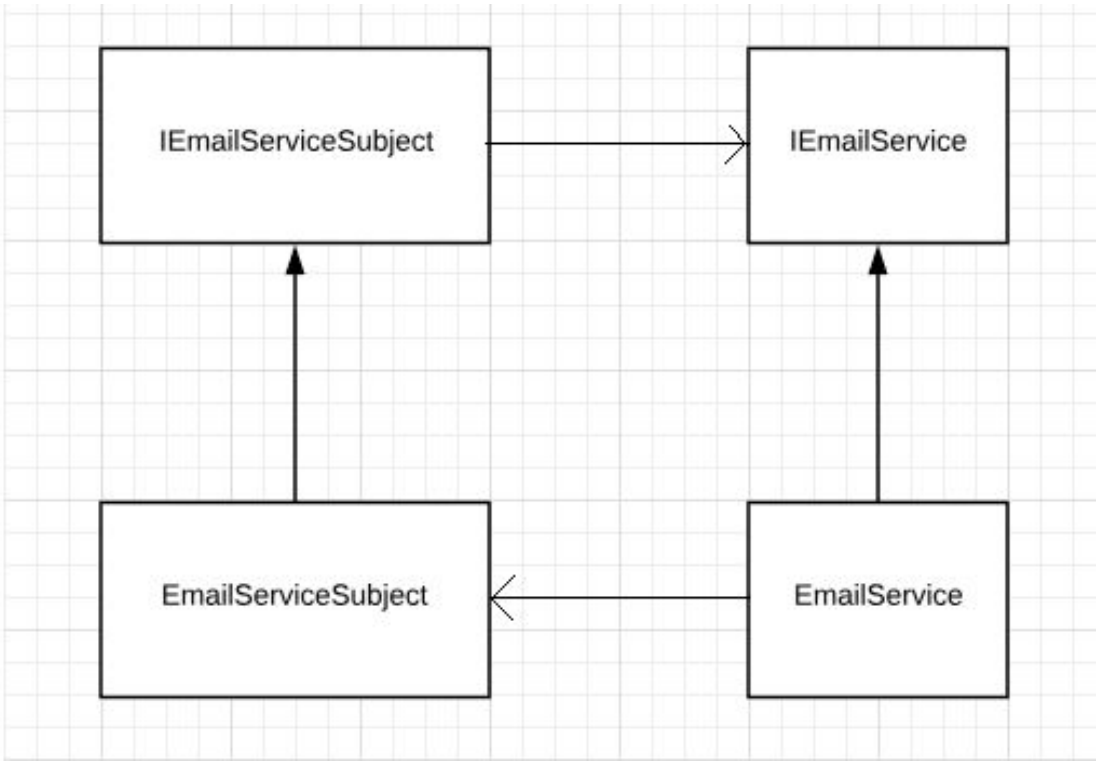
**Singleton design pattern**

Singleton design pattern has been implemented for DBConnector, SessionHandler, and Logger. Since only one of these objects should be created and used it wherever necessary, we felt singleton design pattern solves our requirement.

Singleton design pattern has been implemented for ImagesToBeSaved class which is used in Add/Edit page to store the list of images that are needed to be saved in the recipe. The Add/Edit page contains functionality upload multiple images each at once and remove the images before saving the images. In order to manage images like to add, remove and finally clear all the them after saving the recipe, this singleton is used. Add/edit page have multiple submit buttons and also validations to be performed like empty mandatory fields, same recipe name e.t.c., which will result in removal of images from the page. In order to retrieve and maintain the list of images to be saved, this singleton is used across the add/edit recipe process to maintain only one object which can used to maintain the list of images to be saved.

**Observer design pattern**

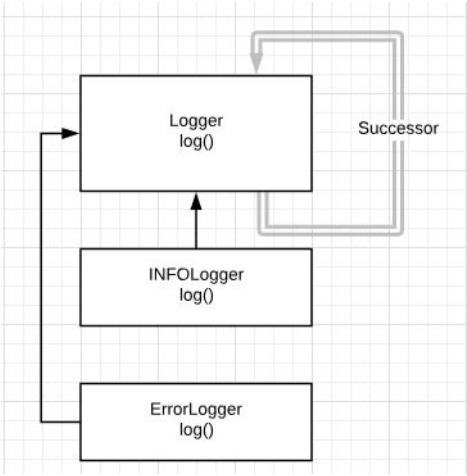
Observer design pattern has been implemented for email feature which emails all followers of a followee. EmailServiceSubject is a subject here which does the whole thing. When admin approves a recipe, all the followers who are following this recipe owner should be informed. The admin approve logic loads the subject with all the followers when admin is prompted to approve the recipe. When admin approves, it immediately notify subject to send email to all the followers and removes the followers once it is done with emailing. For this purpose, we felt implementing observer design pattern better suits the scenario. It notifies everyone at a time.




*UML of bserver design pattern*

**Chain of Responsibility design pattern**

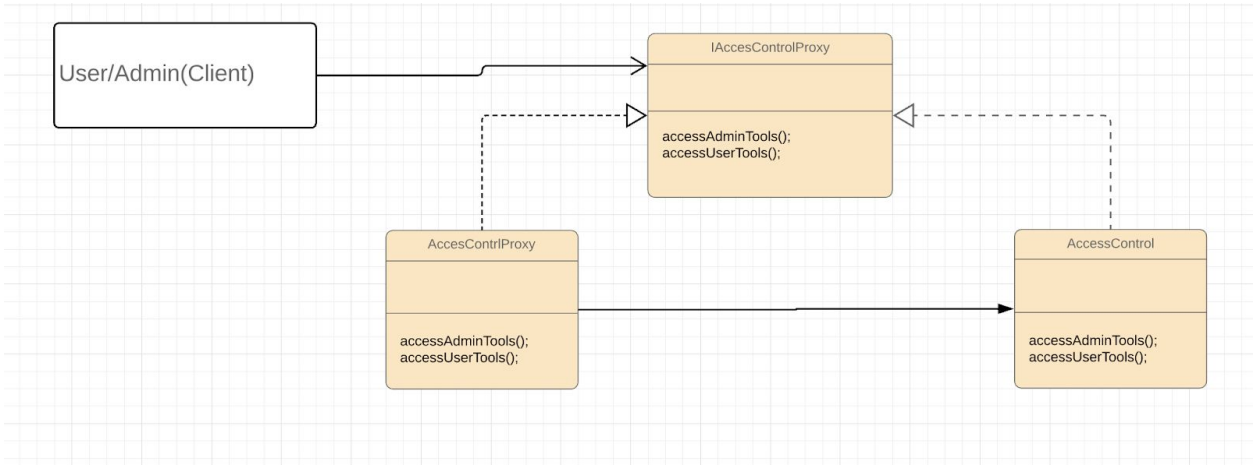
For the Logger, we have implemented chain of responsibility design pattern which is contained in LoggerInstance singleton. We implemented this to eliminate usage of enum/switch based business logic. Since the logger needs to be implemented in such a way that a request has to be sent along the chain until an object handles it. Also LoggerInstance does not know which object would handle the request until one of objects handles it.




*UML of Chain of responsibility design pattern*

**Protected Proxy design Pattern:**

For getting the tools for the respective user i.e, admin, logged in user, non logged in user, we have used protected proxy pattern. AccessControl is the class which contains all the tools for both the users and admin, AccessControlProxy is the proxy class used to protect the AccessControl class which have all the tools and provide the appropriate tools required based on the logged in user. For Logged user 'MyRecipes' tool is provided which will be used to view and add recipes, For Non Logged in user no tools are provided, If the admin is logged in then 'ApproveRecipes', 'ConfigurePassword', 'Delete Recipes' are provided. We have chosen this pattern as it provides protection for the tools and also easy to add new tools without having if else statements.




*UML for Protection Proxy pattern*

In the similar way, the classes "RecipeDescriptionController", "HomeController", "ApproveController" loads the view according to the user. when User opens a recipe few functionalities like rating, follow, edit are controlled according to the user Logged In. If no user is

logged in then the recipe view just gives the Information related to recipe. If a user is logged in and if he/she opens a recipe which they have uploaded, they are provided with the option to edit their own recipes and If they open some other recipe which is not uploaded by them, they are allowed to rate and follow.

**Template Method:**

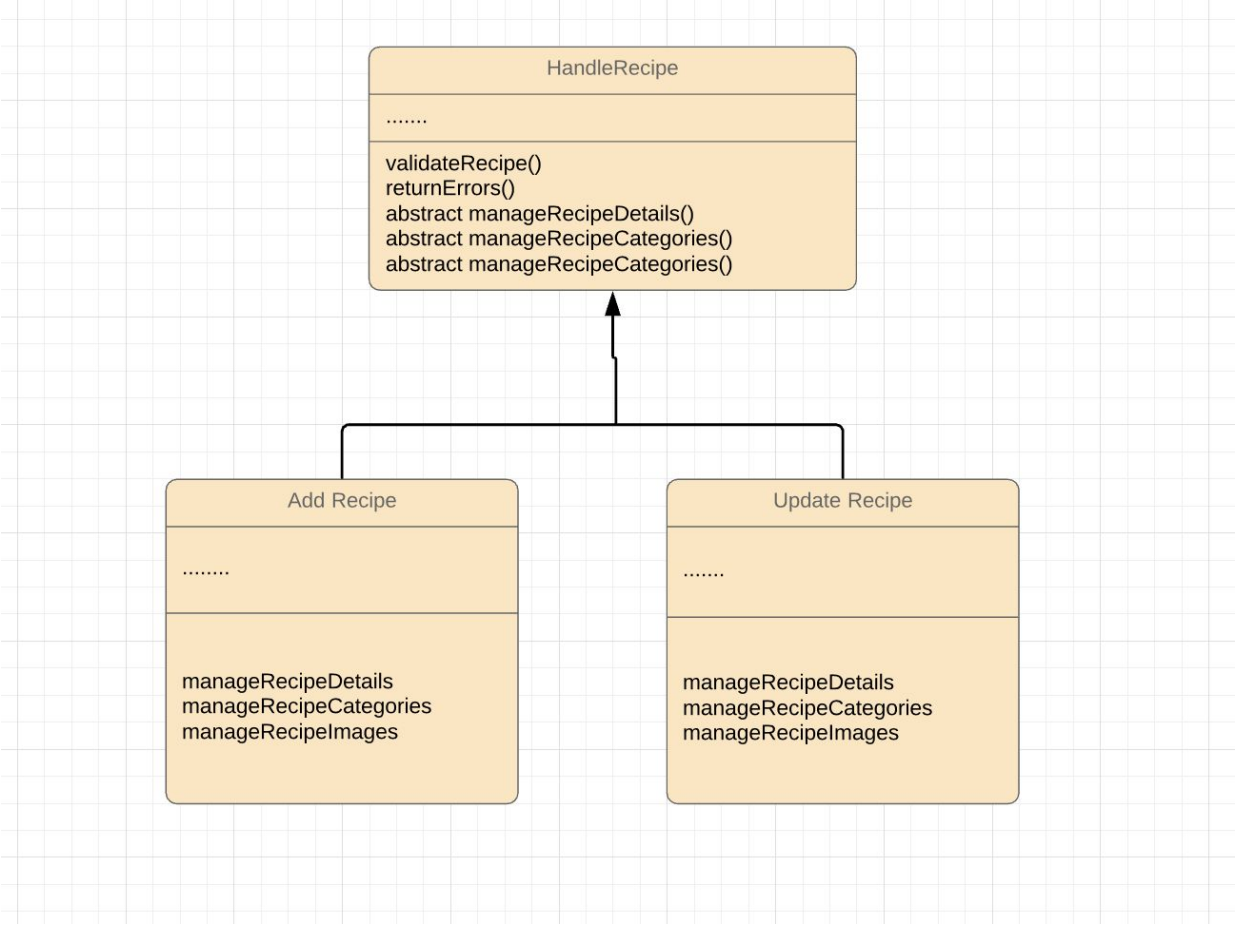
Template method is used to define an algorithm/steps that are needed to be followed to perform add and update recipes and also to remove the duplicate code and steps that are involved in both the processes. The Handle Recipe class is an abstract class serves as the template to Add / Update algorithms, Both the processes involves:

1.Validate the Recipe
- Return errors in Mandatory fields ( if any )

3.Manage the Recipe Details

4.Manage the Recipe Categories

5.Manage the Recipe Images




*UML for Template Method design pattern*

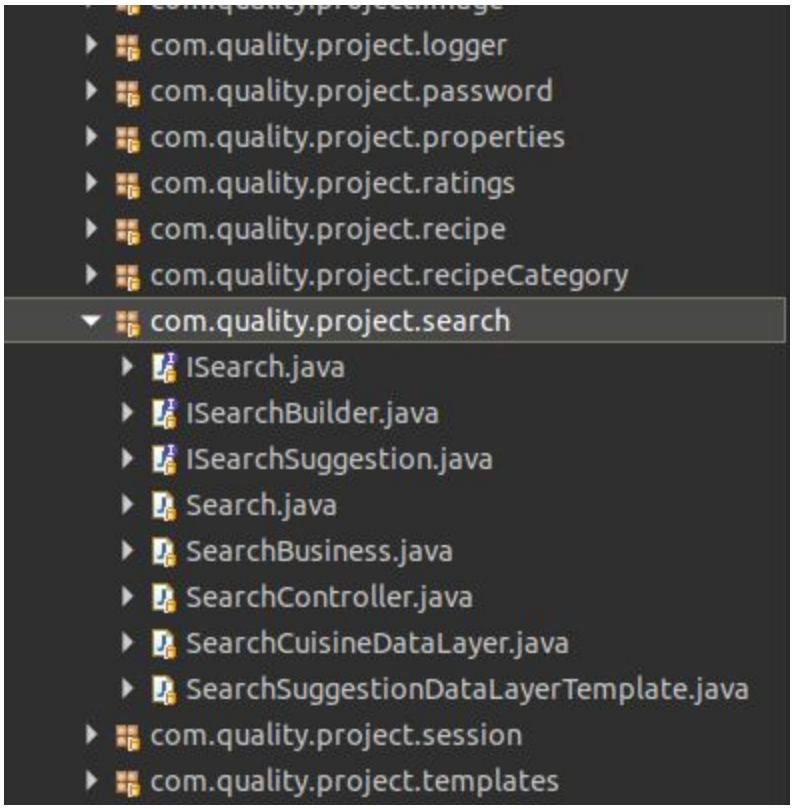
So Validation process is same for both add and edit, so it is defined in the HandleRecipe Class and rest three are abstract which have child classes that defines the methods in AddRecipe

Class and UpdateRecipe class. Thus, Template method suits in this scenario where the steps involved are same but the implementations are different. Using this pattern we are able to remove the duplicate code and retain the algorithm to add and update recipe.

**Factory Method:** Simple Factory IRecipeFactory Class is created to make two different types of Recipes, which is used across the project to create recipe objects.

# Separation of business / presentation / data layers

In order to implement the structure of the project, we used a modularized approach where each module contained the business layer, presentation layer and data layer. We were able to combine this concept with MVC pattern, where model contained the business logic and various interfaces and abstract classes required for implementation. View would request a resource from controller and controller would interact with model. A screenshot for the various layers is shown below:




We were able to achieve modularized approach with each module placed separately. Moverover, we were able to divide the various layers by adding a suffix after each class. For example, in case of search business layer, we have added SearchBusiness making it easy to recognize that it's a business layer class and it would contain the main logic for search implementation.
For data layer, we have added a dataLayer suffix for each class. For example, in case of search cuisine data layer, we have added a suffix data layer to signify that its a data layer and would

contain the interaction with the database. Moreover, if in case it's implementing a pattern like Search suggestion is implementing a template pattern, we have added template as a suffix to make it visible that its a part of data layer which implements a template pattern.

# Naming and spacing conventions

Naming conventions were as follows:

- **Classes** should start with capital letter
- **Interface** should start with I
- **Variables** should have camelCasing
- **Methods** inside class should start with small case and follow camelCase
- All **constants** in class should be in capital letters(full word)
- **Avoided boolean** in parameter, in such case, make two different functions for true and false cases and use them to provide better encapsulation

Spacing conventions: **Tabs**

# Coding strategy

- Avoiding dependency of one class with another by using **dependency injection**. We have tried to make code structure such that the object is not dependent on the class, rather, it's dependent on the interface, such that any change in the module would depend on the interface and not the concrete class itself.
- We followed **SOLID** principles throughout our application. For example, We made sure that a single class is responsible for functionalities related to particular feature. For example,instead of adding all the database interaction within dBConnector.java, we tried to make it exclusive to each module, that way dBConnector.java is just containing connection related features but nothing else.
- **Test-driven development** was followed wherever possible. We started off by writing the unit test cases for modules and then added the functionality for each module. This type of approached helped us when we were refactoring the code, making sure that we do not alter the functionality of the code.
- **Design Patterns** were implemented wherever possible. Such type of approach helped us create a proper structure for the application and helped us reuse the code whenever possible.

# Refactoring performed

- **Form Template Method:** Initially, we started by implementing CRUD operations for each module separately, which was replaced by a template pattern making the code more centralized.
- **Introduce Parameter Object:** Instead of passing values to the function we are passing the object binded with the interface, making it dependent on the interface and taking the value from that object, thereby reducing the number of parameters to be passed and binding it to the interface.
- **Self-encapsulating field:** For each class that has variables, we have created getters and setters to access those values.
- **Encapsulation:** We have changed public member variable to private thereby limiting the access to a single class only.
- **Duplicated Code:** In lot of places we have removed Duplicated code by using different methods like Extract Class, Extract Method.

A list of technical debt of the current state performed and how could these be resolved?
Technical debt is as follows:

- **Dependency on classes:** A lot of dependency has been removed, however, there are some places where the dependency of one class exists on another.
- **Parameterized Method:** There are still some places in the project where we are passing values to the parameter instead of passing objects.

# Contribution:

## List classes written by each group member:

**Lakshmi Suresh Kumar Gangumalla**

- com.quality.project.databaseConnector
  - DBConnector.java **(Singleton)**
- com.quality.project.email
  - EmailService.java
  - EmailServiceSubject.java **(Observer)**
  - IEmailServiceSubject.java
- com.quality.project.logger
  - ErrorLogger.java
  - FatalLogger.java

    - InfoLogger.java
    - Logger.java **(Chain of Responsibility)**
    - LoggerInstance.java **(Singleton)**
    - WarnLogger.java
- com.quality.project.password
    - IConfigurePassword.java
    - ConfigurePassword.java
    - IPasswordParam.java
    - PasswordParam.java
    - IPasswordValidator.java
    - PasswordValidator.java
    - PasswordConfigDBConnector.java
- com.quality.project.properties
    - AdminProperties.java
    - ApplicationProperties.java
- com.quality.project.session
    - ISessionHandler.java
    - SessionHandler.java **(Singleton)**
- com.quality.project.user
    - IAdminLogin.java
    - AdminLogin.java
    - IRegistration.java
    - Registration.java
    - IUser.java
    - User.java
    - IUserLogin.java
    - UserLogin.java
    - UserDatabaseManager.java
- com.quality.project.JUnit.registrationAndLoginTest
    - IUserMockFactory.java
    - UserMockFactory.java **(Singleton)**
    - LoginTest.java
    - RegistrationTest.java
- com.quality.project.JUnit.passwordConfigTest
    - PasswordValidatorTest.java
    - IPasswordMock.java
    - PasswordMock.java **(Singleton)**
- com.quality.project.JUnit.sessionHandlerTest
    - SessionHandlerTest.java

**Naveen Kusakula**

- Com.quality.project.recipe
    - HandleRecipe.java **(Template Method)**
    - AddRecipe.java **(Template Method)**
    - UpdateRecipe.java **(Template Method)**
    - DeleteRecipe.java
    - IDelete.java
    - IRecipe.java
    - Recipe.java
    - IRecipeValidator.java
    - RecipeValidator.java
    - GetRecipesForUser.java
    - IRecipeCRUDDataLayer.java
    - RecipeCRUDDataLayer.java (Sai Chand)
    - IRecipeFactory.java
    - RecipeFactory.java **(Factory  Method)**
    - FetchRecipesDataLayer.java (Sai Chand, Navneet)
- com.quality.project.recipeCategory
    - AddNewCategory.java
    - FetchCategoriesDataLayer.java
    - ICategoryValidator.java
    - CategoryValidator.java
    - IRecipeCategoriesCRUDDataLayer.java
    - RecipeCategoriesCRUDDataLayer.java (Sai Chand)
- com.quality.project.accessControl
    - AccesControl.java
    - AccesControlProxy.java **(Proxy Pattern)**
    - IAccessControl.java
- Com.quality.project.image
    - DeleteImageFile.java
    - IDeleteImageFile.java
    - IImageValidator.java
    - IImageCRUDDataLayer.java
    - ImageCRUDDataLayer.java (Sai chand)
    - ImagesToBeSavedForRecipe.java **(Singleton)**
    - ImageValidator.java
    - UploadImageFile.java
    - IUploadImageFile.java

- com.quality.project.JUnit.AccessControlTest
    - AccessControlTest.java
- com.quality.project.JUnit.AddRecipeValidatorTest
    - IRecipeMockFactory.java

    - RecipeMockFactory.java
    - RecipeValidatorTest.java
- com.quality.project.JUnit.CategoryValidatorTest
    - CategoryMockFactory.java
    - ICategoryMockFactory.java
    - CategoryValidatorTest.java
- com.quality.project.JUnit.ImageValidatorTest
    - IImageFileMockFactory.java
    - ImageFileMockFactory.java
    - ImageValidatorTest.java

**Navneet Prakash Singh**

- com/quality/project/templates/ICRUDTemplate.java **(Template Pattern)**
- com.quality.project.search.ISearch.java
    - ISearchBuilder.java **(Builder Pattern)**
    - ISearchSuggestion.java
    - Search.java
    - SearchBusiness.java
    - SearchController.java
    - SearchCuisineDataLayer.java
    - SearchSuggestionDataLayerTemplate.java **(Template Pattern)**
    - AdvanceSearchDataLayer.java

- com,quality.project.recipeCategory.
    - CategoriesBusiness.java
    - Category.java
    - CategoryDataLayerTemplate.java
    - ICategory.java
- com.quality.project.ratings.IRatings.java
    - RatingDataLayerTemplate.java
    - Ratings.java
    - RatingsBusiness.java
- com.quality.project.JUnit.Category.CategoriesMockFactory.java
    - CategoriesTest.java
    - ICategoriesMockFactory.java
- com.quality.project.JUnit.Search.ISearchMockFactory.java
    - SearchMockFactory.java
    - SearchTest.java

**Srisaichand Singamaneni**

- Com.quality.project.image
    - IImages.java
    - Images.java

    - IImageCRUDDataLayer.java
- Com.quality.project.password
    - PasswordEncrypt.java
- Com.quality.project.recipe
    - IRecipe.java (Naveen)
    - Recipe.java (Naveen)
    - RecipeCRUDDataLayer.java
    - FetchRecipesDataLayer.java (Naveen, Navneet)
- Com.quality.project.ratings
    - RatingDataLayerTemplate.java
- Com.quality.project.Admin
    - ApproveController.java
    - AdminApprove.java **(Uses Observer to send Notifications)**
- Com.quality.project.followers
    - FollowersDataLayer.java
- Com.quality.project.view
    - LoadUserView.java **(Proxy Pattern)**
    - LoadAdminView.java
- com.quality.project.JUnit.passwordConfigTest
    - PasswordEncryptTest.java

**List of Views created by each group member:**

**Lakshmi Suresh Kumar Gangumalla**

- login.jsp
- registration.jsp
- configurepassword.jsp

**Navneet Prakash Singh**

- common.css
- common.js
- common.footer.jspf
- common.header.jspf
- common.navigation.jspf
- Home.jsp (Naveen)
- search.jsp

**Naveen Kusakula**

- addrecipe.jsp
- approverecipes.jsp
- categoryresults.jsp
- home.jsp (Sai chand, Navneet)
- userpage.jsp

**Srisaichand Singamaneni**

- adminApproveRecipe.jsp
- recipedescription.jsp
- home.jsp (Navneet)

## List of presentation layer files written by each group member:

**Lakshmi Suresh Kumar Gangumalla**

- com.quality.project.password
    - ConfigurePasswordController.java
- com.quality.project.user
    - RegistrationController.java
    - LoginController.java
    - LogoutController.java

**Naveen Kusakula**

- Com.quality.project.recipe
    - AddOrUpdateController.java
    - DeleteRecipeController.java
- Com.quality.project.admin
    - RecipesTobeApprovedController.java
- Com.quality.project.home
    - HomeController.java (SaiChand, Navneet)
- Com.quality.project.user
    - UserPageController.java

**Navneet Prakash Singh**

- com.quality.project.home.HomeController.java (Sai Chand, Naveen)
- com.quality.project.search.SearchController.java

**Srisaichand Singamaneni**

- com.quality.project.home
    - HomeController.java (Navneeth, Naveen)
- com.quality.project.recipe
    - RecipeDetailsController.java

## List of stored procedures written by each group member:

**Lakshmi Suresh Kumar Gangumalla**

- sp_addLog (to post logs into database)
- sp_addUser (to add into database)
- sp_fetchUser (to get user from database)
- sp_passwordConfig (to get password configuration from database)
- sp_updatePasswordConfig (to update password configuration in database)

**Naveen Kusakula**

- sp_addNewRecipe(adding the Recipe details  into the recipes table)
- sp_updateRecipe(update Recipe details into the recipes table)
- sp_getRecipeId(get RecipeId for all the recipe name)
- sp_getRecipesToBeApproved(get Recipes that are to be approved from recipe table)
- sp_getRecipesByUser(get Recipes for the User Id from recipe table)
- sp_getRecipeWithSameRecipeNameAndDifferentRecipeID(get recipe with same recipe Name and Different recipeId from recipe table)
- sp_getRecipesForCategory(get recipes related to the category from categories and recipes table)
- sp_deleteRecipeDetails(delete recipes details from recipe table)
- sp_getAllRecipes(get all recipes from recipe table)
- sp_getAllCategoriesList(get all categories from categories table)
- sp_getCategoriesWithRecipes(get categories with recipes associated from recipes, recipe_categories, categories tables)
- sp_DeleteRecipeCategories(Delete categories related to the recipe in recipe categories table)
- sp_InsertRecipeCategories(Insert categories related to the recipe in recipe categories table)
- sp_addImageForRecipe(insert new image related to the recipe in recipe_images table)
- sp_DeleteImagesForRecipe(delete recipes related to the recipe in recipe_images table)

**Navneet Prakash Singh**

- sp_addCategory (adding the category in the category table)
- sp_fetchCategory (fetching all the categories from the category table)
- sp_removeCategory (removing a particular category from the category table)
- sp_updateCategory (updating a particular category against the id)
- sp_addSearchSuggestion (check if a suggestion does not exists for user, the add suggestion against the user id)
- sp_fetchAdvanceSearchResults (fetch all the results on the basis of advance search criteria)
- sp_fetchCountForRecipes (fetch count for recipes against a particular recipe)
- sp_fetchDistinctRatings (fetch all the distinct ratings of all recipes)

- sp_fetchsearchResults (fetch search results against a particular keyword)
- sp_fetchSearchSuggestion (fetch search suggestion against a particular keyword)
- sp_newArrivals (fetch all recipes according to the order in which they were created)

**Srisaichand Singamaneni**

- sp_fetchRecipe (fetch details of the Recipe using recipe Id)
- sp_insertRating (Insert the rating provided by the User for the Recipe)
- sp_fetchRatingByUser (fetch rating given by user for that particular Recipe)
- sp_fetchCategoriesForRecipe (fetch categories belonging to the Recipe)
- sp_getFollowing (fetch the list of all the people the current user is following)
- sp_followUser (Insert entry for following a particular user)
- sp_fetchFollwersforUser (fetch the list of all the followers for the user)
- sp_fetchImagesForRecipes (fetch all the images related to the recipe)

**Application Link :**

***https://csci5308prod.herokuapp.com/home***