중앙대학교
CHUNG-ANG UNIVERSITY

Object-Oriented Programming

IIKH Project

TEAM 1 Members

20145263 박준홍

20143408 방준석

20201613 유준서

20153479 이동민

20174266 이지호

0. Index

1. Requirements Analysis

1.1 Requirements Specification

The specifications for the IIKH requirements described in the textbook are as follows: (However, it only deals with functional requirements.)

[1] Query recipe database
[2] Adding a database recipe
[3] Modifying and annotating recipes
[4] Adding a plan for meal with multiple recipes
[5] Calculation of the number of materials required according to the number of people
[6] Establishing a daily or more plan
[7] Generating a list of groceries that are included in all menus over a set period of time
[8] Searching for recipes with one or more keywords
[9] Printing out recipe
[10] Annotate for a specific date
[11] Retain recipe data as a file

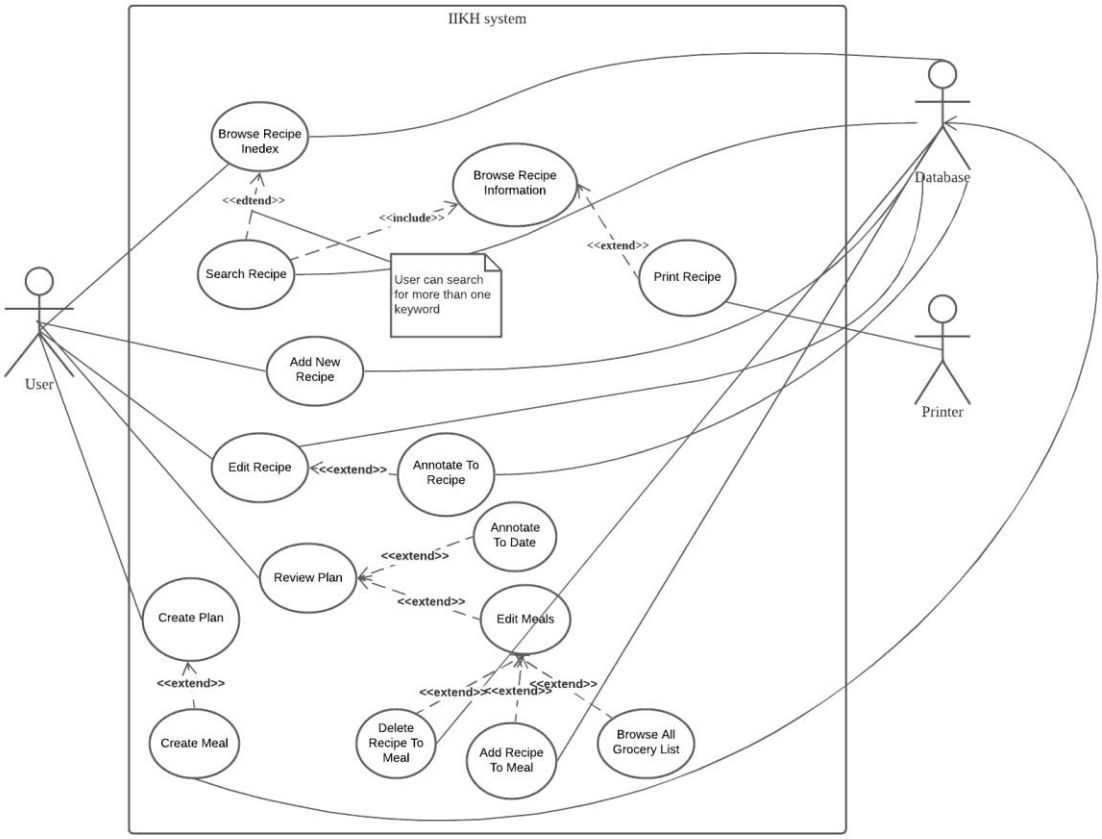
Although the requirements were identified as follows and the component classification and the responsibility sharing of each component were also shared, the requirements were re-analyzed with only reference to the requirements.

1.2 Use-case Diagram

With an understanding of the functions available to the user as an IIKH system, the Use-case was analyzed and diagrammed and used as visual data, improving the understanding of the system as a whole team member.

The process of preparing a user-scale diagram was identified as the most representative, high-level user-scale functions provided on the first screen, and extended by assuming the scenario using them from the user's perspective.

A diagram of these identified use cases is as follows.



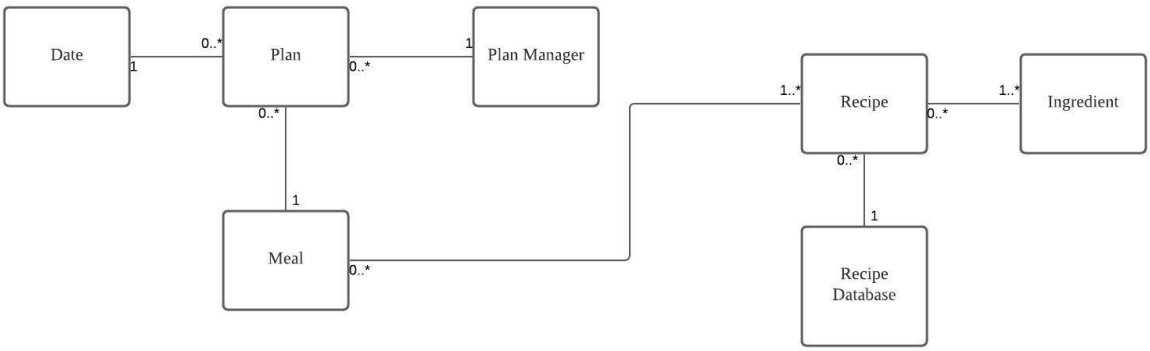
[Figure 1-1] Use-Case Diagram

Top-level Use-cases include recipe index inquiry, recipe addition, recipe modification, plan inquiry, and plan generation.

For example, in the Use-case of recipe index inquiry, you can think of a scenario in which all recipes are searched by the material name on the screen listed, and a use-case called Search Recipe has been identified. In addition, if two or more keywords are given, the restrictions that print recipes containing all keywords are marked as notes.

It also expressed the relationship between returning the results of the recipe search, inquiring about it, and printing the desired recipe directly to the printer.

1.3 Domain Model



[Figure 1-2] Domain Model

The next step in the requirements analysis was to identify the domain of the system. If the user-led behavior was identified through the use-case analysis, the relationship between the objects participating in the action could be identified by creating a domain model.

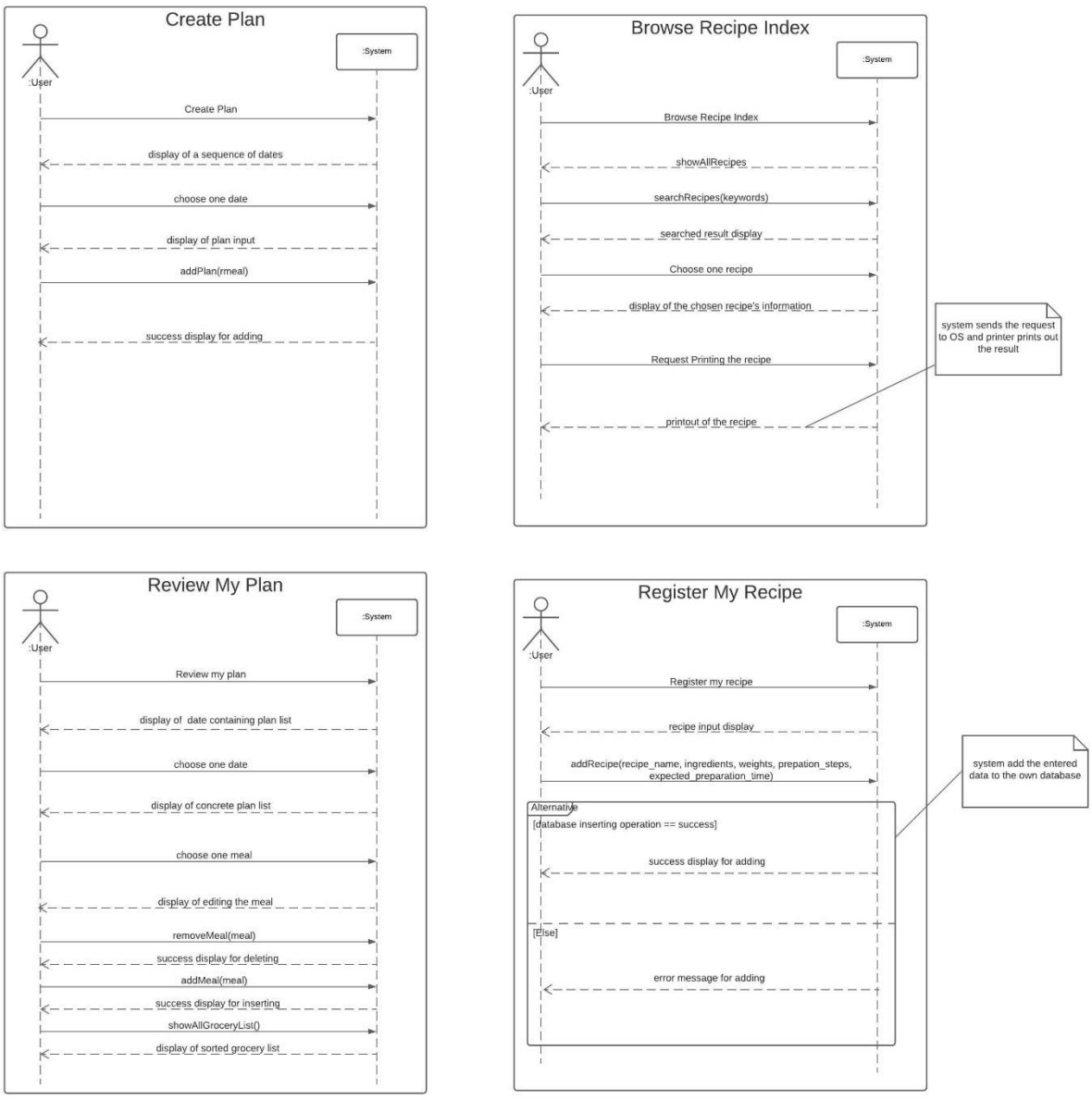
What materialized through this stage was the emergence of another object plan linking Meal and Date. Plan was discussed to combine Date and Meal into a single plan and to allow reference to that date.

1.4 System Sequence Diagram

Once the objects that participate in the system have been identified, it is the step to identify Operation. Based on the use case, input users can express how they interact with the system and understand the user's inputs and the corresponding actions of

the system for the expressions that appear.

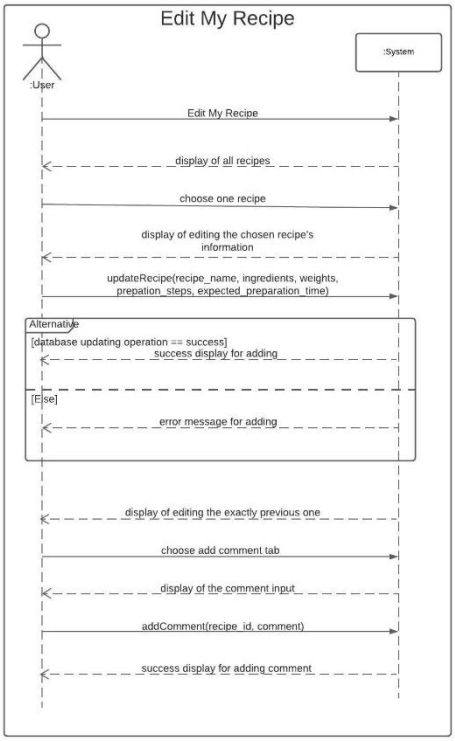
This process was carried out on the top five use cases



[Figure 1-3, 1-4] SSD for plan verification, plan creation, recipe registration, recipe inquiry

Interaction between the user and the system provides an overview of the flow of each use case and the response of the system to the user's input.

For example, operations such as addPlan, removeMeal, addMeal, showAllGroceryList, searchRecipes, and addRecipe were identified, and approximate parameters were identified for each.



[Figure 1-5] Edit Recipe SSD

The same is true of recipe modification. The updateRecipe, addComment, etc. have been identified.

2. Design

2.1 Object-Oriented Design

Responsibilities should be properly distributed for a good design. The responsibility for distributing the identified requirements through a youth case analysis to objects

identified in the Domain Model.

[Recipe] has items containing the recipe (Recipe name, material, weight of each material, preparation process, preparation time). It can also print the information it contains.

[Meal] has a list of recipes to include. For each recipe, the number of people can be set, and the required materials can be sorted out according to the number of people.

[Plan] It contains information by mapping customized meals and dates one-on-one.

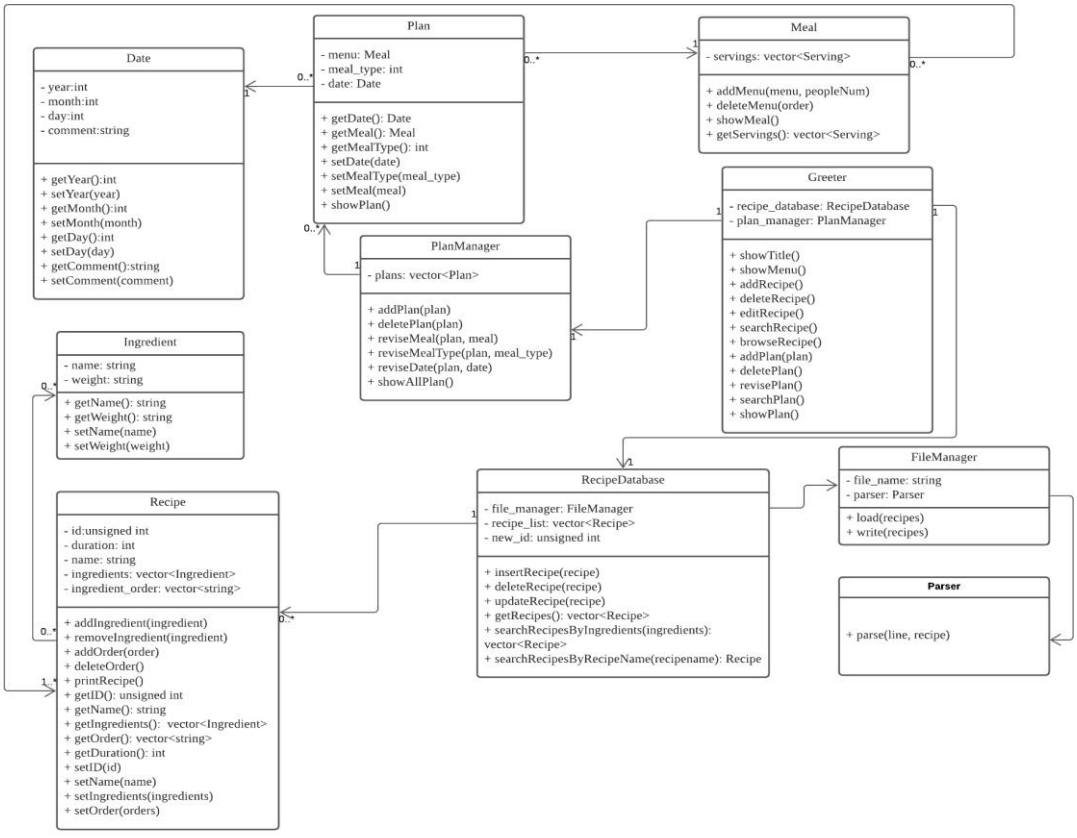
[Plan Manager] is responsible for managing the plan entered by the user. You can enter, delete, modify, or query a plan.

[Recipe Database] is responsible for administering the recipe. It has all the recipes entered and can be stored permanently in the form of a database. You can also enter, delete, modify, and view recipes.

[Date] Has date information. You may include comments about the date.
be distributed with.

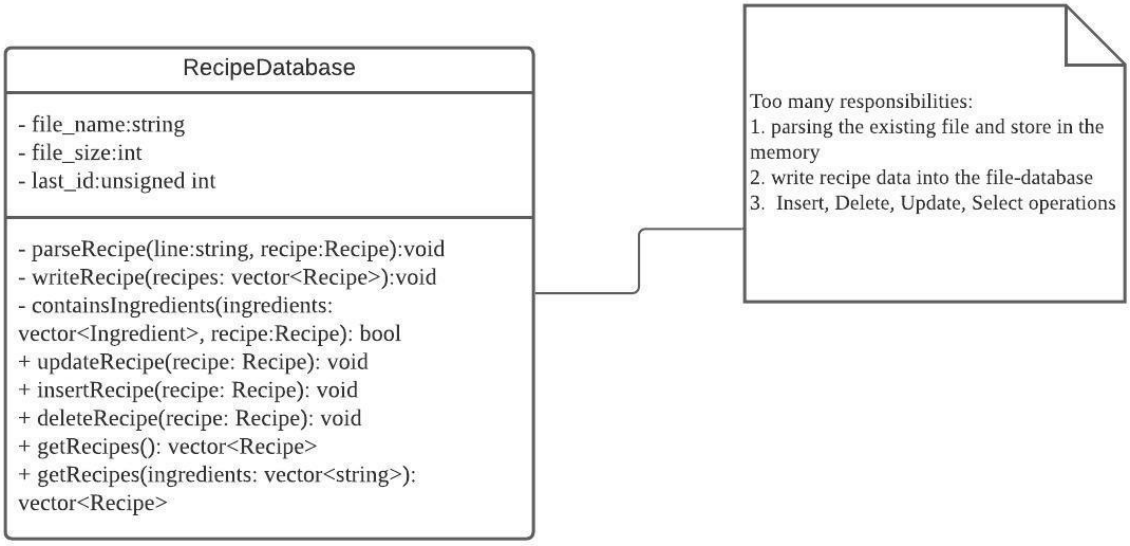
Also, the Greeter class in charge of UI should be designed with this in mind, as it should not affect other classes in charge of logic in the program. Based on this distribution of responsibilities, the class diagram is designed as follows:

2.2 Design Class Diagram



[Figure 2-1] Class Diagram

Greeter, who is in charge of the overall UI of the system, knows RecipeDatabase and PlanManager, which manipulate the data covered by the system. Once a user's input is received, the user's requirements can be achieved through method calls that properly manipulate the data.



[Figure 2-2] initial database design

The initial database design had all the responsibility, from parsing files to manipulating Recipe data to storing and loading files. However, if the way data is parsed or the way data is kept permanently itself, the RecipeDatabase changes, and the Change Propagation requires the effort to change the known Greeter and costs it to retest. In other words, costs are incurred in terms of maintenance.

Thus, the responsibility for handling files is provided only with operations that encapsulate, load and write. In addition, the parsing responsibility is encapsulated with parser and provided only parse methods.

However, loopholes exist in the completed design. Plan will have to have the same structure as RecipeDatabase if the requirement is changed to be kept in a permanent

file. However, any scalability for him is not guaranteed, so revisions must be made throughout the code.

In this case, a higher class can be created that provides a common operation of the database as an interface, which can be addressed by the design implemented by the two Concrete Classes. However, the initial version did not care because it was not specified in the requirements.

3. Implementation

Prior to detailed code explanation, UI was implemented as Command Line Interface.

Specific code implementations are described based on the requirements mentioned in Section 1.

```
/**
 * @class RecipeDatabase
 * Responsible for performing operations in general databases.
 * It has a recipe array as a member variable, so when it was created, the contents of the file were read and cached in memory.
 * Update what is added, deleted, or modified while the program is running
 * Save the contents of the array to a file when the instance is destroyed.
 *
 */
class RecipeDatabase {
private:
    FileManager *file_manager;
    vector<Recipe> recipe_list;
    unsigned int new_id;
public:
    RecipeDatabase();
    ~RecipeDatabase();
    void updateDatabase(Recipe recipe);
    /**
     * @param new_recipe: reference of the recipe instance to insert
     * @param recipe: reference of the recipe instance to delete
     */
    void insertRecipe(Recipe &new_recipe);
    void deleteRecipe(Recipe &recipe);
    vector<Recipe> getRecipes();
    /**
     * @param keywords User-entered ingredient list
     */
    vector<Recipe> searchRecipes_ingredients(vector<string> ingredients);
    Recipe searchRecipes_recipename(string recipename);

};
```

[Code 3-1] Implementation of RecipeDatabase Class

[1] Query recipe database

```
void Recipe::printRecipe()
{
    {
        cout << "------------------"<< endl;
        cout << "| Selected Recipe |" << endl;
        cout << "------------------"<< endl;

        cout << " Name: " << getName() << endl;
        cout << " Cooking Time: " << getDuration() << endl;

        if (getIngredients().size())
        {
            cout << endl;
            cout << " <Ingredients List>" << endl;
            for (int i = 0; i < getIngredients().size(); i++)
            {
                cout << " " << i + 1 << ". " << getIngredients()[i].getName() << "  "
                    << getIngredients()[i].getWeight() << endl;
            }
        }

        if (getOrder().size())
        {
            cout << endl;
            cout << " Cooking Order " << endl;
            for (int i = 0; i < getOrder().size(); i++)
            {
                cout << " " << i + 1 << ". " << getOrder()[i] << endl;
            }

        }

        cout << endl << endl;
    }
}
```

All instances of Recipe created in the recipe database are kept in vector <Recipe>. If you return it to getRecipes and call each printRecipe for the Recipe instances, you can inquire information about all recipes.

[Code 3-2] Recipe Output Function

[2] Adding a database recipe

If the user-entered information is passed to the insertRecipe of the database, it is added to the recipe_list that exists as a member of the database.

[3] Modifying and annotating recipes

Entering the editRecipe entry of Greeter, you can receive input for the recipe you want to modify, and turning over the modified recipe to updateRecipe in the database will modify the information in the recipe_list.

[4] Adding a plan for meal with multiple recipes

```
#include "recipe.h"

class Meal
{
private:
    string name;
    vector <Serving> servings;

public:
    Meal();

    ~Meal();

    string getName() { return this->name; }
    void setName(string name) { this->name = name; }
    // add menus
    void addMenu(Recipe &_menu, const int& people_in = 1);

    // delete menus and ingredients form selected menus
    void deleteMenu(int order);

    // show name of menus and ingredients scaled by num_of_people
    void showMeal();

    vector <Serving> getServings();
};
```

```
typedef struct Serving {
    Recipe menus;
    int num_of_people;
} Serving;
```

```
class           Plan
{
private:

    Meal        meal;
    int         meal_type;
    Date        date;
```

[Code 3-3] Meal, Plan Class and Serving Structure

There can be several recipes in Meal and they are managed with vectors. It also defines a structure called Serving and manages the number of human feces in each recipe. The plan defines the date's metal as one plan by tying it to the date.

[5] Calculation of the number of materials required according to the number of people

```
void Meal::showTotalIngredients() {
    vector<string> ingredients;
    unordered_map<string, int> required_ingredients;

    for (auto each_menu : servings)
    {
        for (auto each_ingredient : each_menu.menus.getIngredients())
        {
            string ingredient_name = each_ingredient.getName();
            int ingredient_weight = stoi(each_ingredient.getWeight()) * each_menu.num_of_people;
            if(!required_ingredients.count(ingredient_name)) {
                ingredients.push_back(ingredient_name);
                required_ingredients[ingredient_name] = ingredient_weight;
            }
            else {
                required_ingredients[ingredient_name] += ingredient_weight;
            }
        }
    }

    cout << "---------------------" << endl;
    cout << "| Required Ingredients |" << endl;

    for(string ingredient_name: ingredients) {
        cout << "| " <<  ingredient_name << ", " << required_ingredients[ingredient_name] << " |" << endl;
    }

    cout << "---------------------\n" << endl;
}
```

[Code 3-4] Material Output by Number of Personnel

It is a code that collects all recipes included in a single meal in the Meal class and outputs the necessary materials.
Each recipe has the necessary ingredients in the form of an Ingridient class, and Meal binds them into a structure called Serving to manage its arrangement.
Each material can be calculated by multiplying the number of people the Serving has.

The materials required for the entire Meal may be duplicated, so use unordered_map to add the required material weight to the previously explored

material name and unexplored material name to unordered_map.

If you can refer to an instance of Meal in any component without output, you can calculate the amount of each material required by referring to the material arrangement in Recipe and by referring to the number of people in Serving.

[6] Establishing a daily or more plan

Plans for days or more can be implemented by adding the same plan instance to the same specified dates.

[7] Generating a list of groceries that are included in all menus over a set period of time

```
vector<Plan> PlanManager::searchPlan(const Date& begin, const Date& end) {
    vector<Plan> ret;
    vector<Plan> sorted_plans = planData;
    // sort by date and meal_type
    sort(sorted_plans.begin(), sorted_plans.end());

    // for lower_bound searching, setting the meal type as 0, which is the lowest number of it
    Plan begin_keyword = Plan(begin);   begin_keyword.setMealType(0);
    // for lower_bound searching, setting the meal type as 10, which is greater than the highest meal type arrangment.
    // then, lower_bound function will return the Plan iterator that is next to the end Date
    Plan end_keyword = Plan(end);       end_keyword.setMealType(10);
    int begin_idx = lower_bound(sorted_plans.begin(), sorted_plans.end(), begin_keyword) - sorted_plans.begin();
    int end_idx = lower_bound(sorted_plans.begin(), sorted_plans.end(), end_keyword) - sorted_plans.begin() - 1;    // we will not contain the iterator of next day

    ret.assign( sorted_plans.begin() + begin_idx, sorted_plans.end() + end_idx );
    return ret;
}


void PlanManager::showIngredientsForPeriods(vector<Plan> plans) {
    vector<string> ingredients;
    unordered_map<string, int> required_ingredients;

    for (Plan plan: plans)
    {
        vector<Serving> servings = plan.getMeal().getServings();
        for (Serving serving : servings)
        {
            Recipe recipe = serving.menus;
            for(Ingredient ingredient: recipe.getIngredients())
            {
                string ingredient_name = ingredient.getName();
                int ingredient_weight = stoi(ingredient.getWeight()) * serving.num_of_people;

                if(!required_ingredients.count(ingredient_name))
                {
                    required_ingredients[ingredient_name] = ingredient_weight;
                    ingredients.push_back(ingredient_name);
                }
                else
                {
                    required_ingredients[ingredient_name] += ingredient_weight;
                }
            }
        }
    }

    cout << "--------------------" << endl;
```

[Code 3-5] Code for printing a list of materials in a menu for a specified period of time

SearchPlan (Date, Date) receives information about the year, month, and date from the user and returns the plan that corresponds to the period in an array if a set period is set.

The output function can receive the plan arrangement again and output a list of materials integrated by algorithms such as [5].

[8] Searching for recipes with one or more keywords

```
vector<Recipe> RecipeDatabase::searchRecipes_ingredients(vector<string> ingredients) {
    vector<Recipe> ret;
    for(Recipe recipe: recipe_list) {
        int matched_num = 0;
        for(string keyword: ingredients) {
            for(Ingredient ingredient: recipe.getIngredients()) {
                if(keyword == ingredient.getName()) {
                    matched_num += 1;
                    break;
                }
            }
        }

        if(ingredients.size() == matched_num) {
            ret.push_back(recipe);
        }
    }

    return ret;
}
```

[Code 3-6] Recipe Search Code

Receive the material name as a factor and return the matching recipes in an array through a full search.

[9] Printing out Recipe

Each recipe has a printRecipe method that can print itself out.

[10] Annotate for a specific date

The Date class has a comment variable as a member and the user is free to comment on the date by entering comments.

[11] Retain recipe data as a file

```
/**
 * @class FileManager
 * Responsible for opening, closing, reading, and writing files.
 * Load the contents of the file when RecipeDatabase is created,
 * and overwrite the contents of the memory with the file when it is destroyed.
 */
class FileManager {
private:
    string file_name;
    Parser *parser;
public:
    FileManager();
    ~FileManager();
    /**
     * @param recipes
     * [load]  Recipe array of RecipeDatabase to load the contents of text files
     * [write] Recipe array of RecipeDatabase to overwrite in text file
     */
    void load(vector<Recipe>& recipes);
    void write(vector<Recipe> recipes);
};

/**
```

[Code 3-7] FileManager code for file management

FileManager, a member variable of RecipeDatabase, is delegated the responsibility to handle files. When the creator of RecipeDatabase is called, he reads and parses the stored recipe data file and puts it in the array as Recipe data.

When the destructor of RecipeDatabase is called, the recipe array updated during the run time is written to the file.

4. Testing

Each team member is responsible for one component and performs the first test of the unit module he implements.

After testing by linking RecipeDatabase and Recipe to Test, Meal and Plan, PlanManager, and Date, integrated testing with the UI component Greeter was conducted to check the execution of the code.

```cpp
int main() {
    try {
        unique_ptr<RecipeDatabase> recipedb = make_unique<RecipeDatabase>();
        vector<Ingredient> ingredients;
        ingredients.push_back(Ingredient("두부", "400"));
        ingredients.push_back(Ingredient("김치", "600"));
        recipedb->insertRecipe("두부김치", ingredients, {"두부를 썬다.", "김치를 썬다.", "접시에 담는다."}, 5);

        ingredients.clear();
        ingredients.push_back(Ingredient("김치", "400"));
        ingredients.push_back(Ingredient("물", "600"));
        ingredients.push_back(Ingredient("다진 마늘", "10"));
        recipedb->insertRecipe("김치찌개", ingredients, {"다진 마늘을 넣고 볶는다", "김치를 썬다.", "물을 끓인다.", "김치를 넣고 끓인다."}, 20);

        recipedb->deleteRecipe(Recipe(10000));
        recipedb->deleteRecipe(Recipe(10001));

        vector<Recipe> recipes = recipedb->getRecipes();
        for(auto recipe: recipes) {
            recipe.printRecipe();
        }

    } catch(const exception& e) {
        cerr << e.what() << endl;
        return 1;
    }

    return 0;
}
```

[Code 4-1] RecipeDatabase Testing Code

5. Compile and Execution

Created based on Linux execution environment. All source and header files are integrated and compiled, and assembly codes compiled in the oobjs directory are stored.

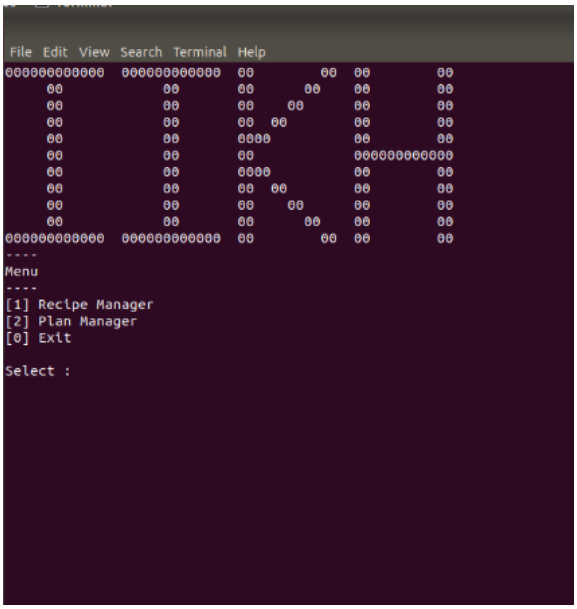
```
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ ls
Makefile  changelog.md  database  includes  objs  srcs
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ sudo make
ar rcs IIKH.a ./objs/main.o ./objs/iikh.o ./objs/greeter.o ./objs/recipe.o ./objs/recipe_database.o ./objs/meal.o ./objs/date.o ./objs/plan.o ./objs/planManager.o
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ ls
IIKH.a  Makefile  changelog.md  database  includes  objs  srcs
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$
```

[Picture 5-1] compiled and generated library files

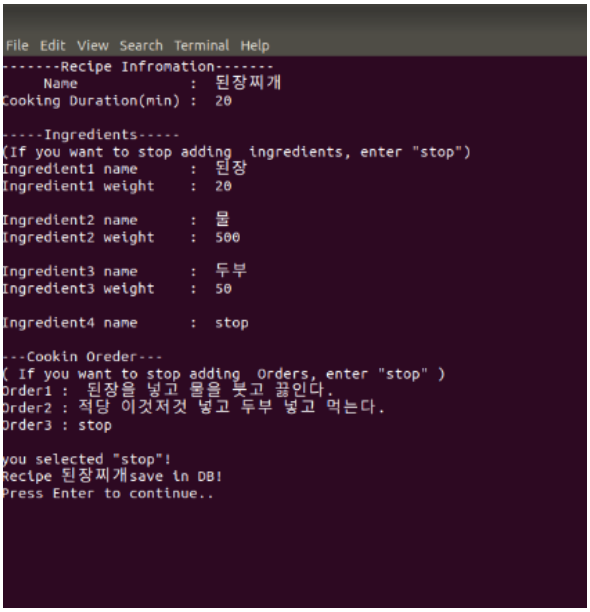
In addition, a static library file named IIKH.a is created in Path where Makefile is stored. Link it to g++ to create an executable.

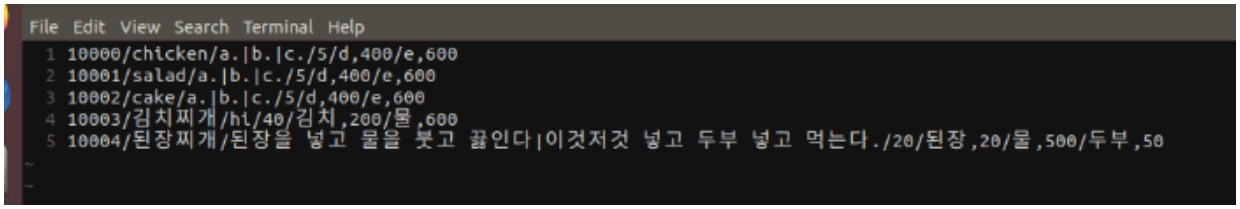
```
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ ls
Makefile  changelog.md  database  includes  objs  srcs
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ sudo make
ar rcs IIKH.a ./objs/main.o ./objs/iikh.o ./objs/greeter.o ./objs/recipe.o ./objs/recipe_database.o ./objs/meal.o ./objs/date.o ./objs/plan.o ./objs/planManager.o
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ ls
IIKH.a  Makefile  changelog.md  database  includes  objs  srcs
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ sudo g++ IIKH.a
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$ ls
IIKH.a  Makefile  a.out  changelog.md  database  includes  objs  srcs
dongmin@dongminlee-Ubuntu:~/OOP/OOP_TEAM1$
```

[Picture 5-2] How the executable was created



[Picture 5-3] Executable is executed





[Picture 5-4] How the recipe was added to the database