# Affamato: Phase Four Report

## Team:

Canvas Group: **Falcon**

**Phase Four Team Lead: Justin Henry**

## Members:


- Cameron Clark: cameron.clark0821@utexas.edu \\ github.com/cameronclark0821
- Justin Henry: justinhenry@utexas.edu \\ github.com/justinhenry
- Alex Issa: alex.issa32@utexas.edu \\ github.com/alexissa32
- Julia Rebello: julialrebello@utexas.edu \\ github.com/JLRebello
- Samir Riad: sriad123@utexas.edu \\ github.com/sriad123
- Rooshi Patidar: rooshipatidar@utexas.edu \\ github.com/rooshimadethis


**URL** to Github/Gitlab repo and shared Google docs:

- Github: https://github.com/alexissa32/Affamato
- Google Drive: https://drive.google.com/drive/folders/0ANUp-cLx6lnZUk9PVA
- Slack: https://team-falcon-group.slack.com

**Website URL:** https://www.affamato.xyz/

**Phase IV Report Contents:**

1. Information Hiding
2. Design Patterns and Refactoring

1. Information Hiding

If Affamato was to be developed further there are many changes that could be made and features that could be implemented. For example, more search parameters could be added, more recipes and/or ingredients could be added, more fields in recipes and/or ingredients could be added, and ingredients could be fully integrated with recipes. The current design of Affamato is prepared to deal with these changes.

In the scenario where more search parameters were to be added, the frontend would certainly need to display the extra checkboxes. This information would then need to be relayed to FilterParameters via the search servlet. However, if there were different searches that didn't take all the parameters specified in filter parameters, it wouldn't affect the search.

Alternatively, if more recipes or ingredients were added, nothing would need to be changed, as there is no dependence on a hardcoded size value or set of ingredients in the code. Furthermore, if more fields were added to recipes and ingredients, the application would continue to function, and displaying these additional fields could easily be implemented in the frontend with JSONObject.get as the frontend is doing with the fields already present.

Furthermore, Affamato could fully integrate ingredients and recipes, allowing users to add ingredients from a recipe. The current design makes this feasible; the ingredient information is already contained in the recipes stored in the user's recipe list. The code to use that information to add to a grocery list or pantry would just need to be implemented in the frontend.

Ultimately, the most important modularity that we preserved is that the frontend JSP files do not have to know how recipes and ingredients are structured to add and delete them from user's lists (see pseudocode below). These files only need to know how they are structured to display to the frontend. A potential disadvantage to this would be if some fields were removed from recipe or ingredient, as it would result in the frontend breaking as it tries to display something that doesn't exist. The other disadvantage is that much of the functionality is in Cook. We mitigated the first issue by ensuring that all recipes and ingredients have the fields that we access in the frontend. In the future if Cook were to expand more, it might make sense to refactor it into multiple classes based on its new functionality.

Pseudocode for a servlet:

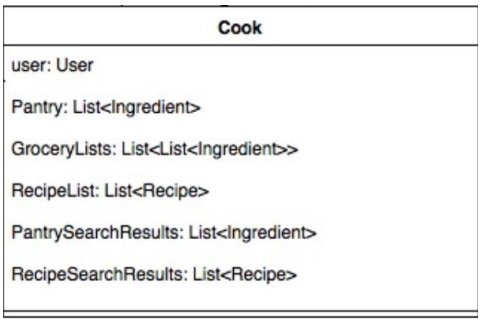
```
FilterParameters parameters = getParameters()

cook.setResults(new Search(parameters).search()) //the cook gets set based on the parameters
                                                 //and the type, the internal logic is hidden to the servlet
```

## 2. Design Patterns and Refactoring

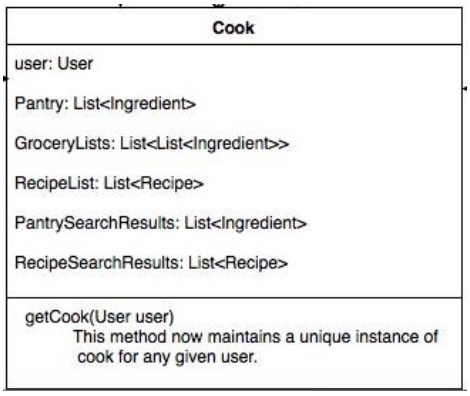
### 2.1 Design Patterns

We changed cook to a singleton to guarantee no two cooks can exist for the same user. Each user needs to have a single cook instance but no more than one. Previously, we were calling the constructor from a servlet; now we are using getCook(user) as a constructor. getCook(user) will either return the instance that already exists or create one if it does not. Additionally, the Cook constructor is now private, and is only accessed by getCook. The advantage of this implementation is that the Cook class ensures that no more than one Cook will be created for each user. The disadvantage is that the Cook constructor is not available outside the Cook class, and now getCook is the defacto constructor which could be confusing.

Before:




After:




We used the Strategy Pattern for search because the ingredient and recipe search had similar algorithmic functionality. The servlet does not need to know how the searches are implemented. It just instantiates the concrete search functionality it needs and calls search(). We decided to encapsulate our algorithms for searching. If we wanted a new search algorithm, we would only need to implement the SearchInterface. Additionally, the search algorithms can be interchangeable and determined at runtime. A disadvantage is that there are more classes to deal with.

Code Before:

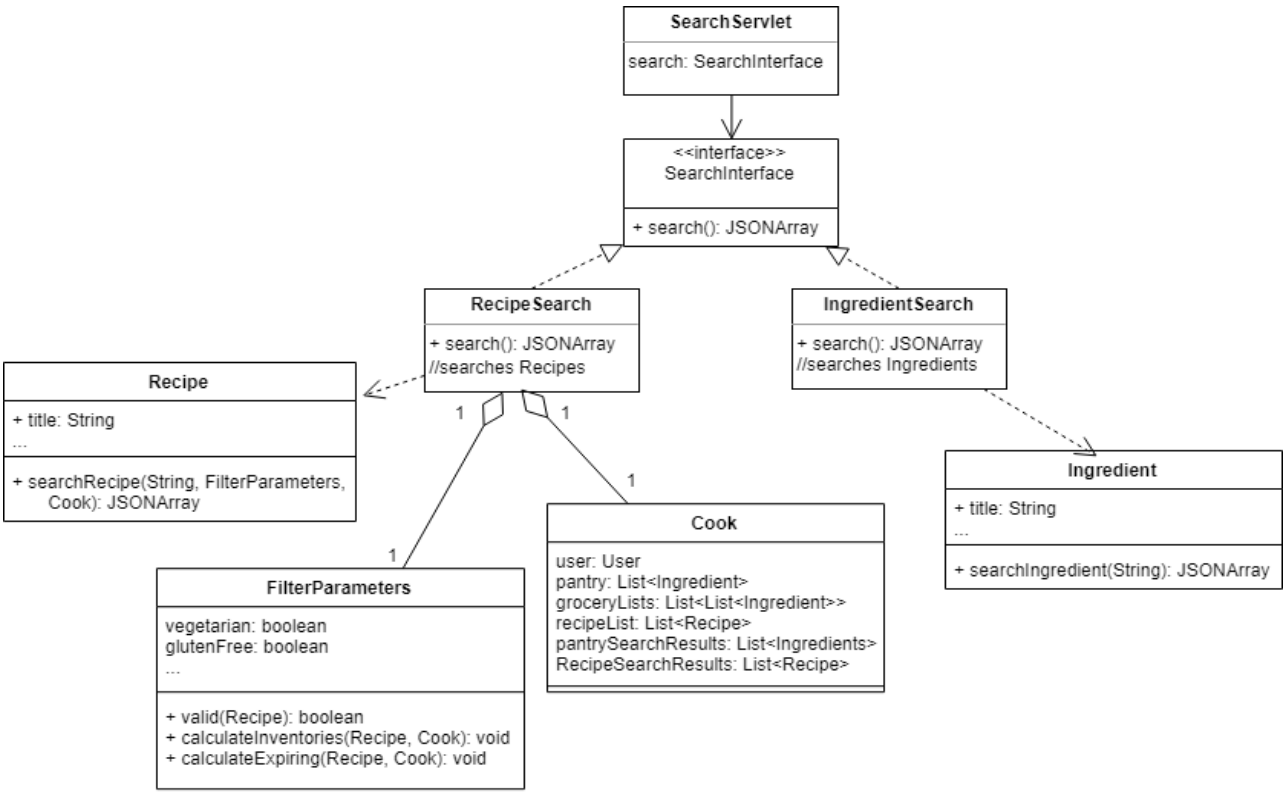
```
JSONArray ja = null;
if(type.equals("recipe")) {
        ja = Recipe.searchRecipe(parameter, param, cook);
        cook.setRecipeSearchResults(ja);
}
```

Code After:

```
  SearchInterface search = null;

 search = new recipeSearch(parameter, filters, cook);
```

```
if(type.equals("recipe")) {
        cook.setRecipeSearchResults(search.search());
}
```

UML Diagram:




2.2 Refactoring

We implemented extract method in the cook class twice. Both refactors removed processing on JSONArrays from remove methods to their own methods.

Refactor 1:
First extract method:

```
boolean noneremoved = true;
//this gets relevant json array inside the initial
for(int i = 0; i < newList.length(); i++) {
        String ingredient = newList.getString(i);
        if(ingredient.equals(Ingredient)&&noneremoved) {
                noneremoved = false;
                //newList.remove(i);
        }
        else {
                replace.put(ingredient);
        }
}
        replace = removeStringFromArray(Ingredient, newList);
```

```
private JSONArray removeStringFromArray(String x, JSONArray pantry) {
for(int i = 0; i < pantry.length(); i++) {
    String ing = pantry.getString(i);
    if(ing.equals(x)) {
        pantry.remove(i);
        break;
    }
}
return pantry;
```

Refactor 2:
Second extract method:

```
for(int i = 0; i < pantry.length(); i++) {
        String ing = pantry.getJSONObject(i).toString();
        if(ing.equals(x)) {
                pantry.remove(i);
                break;
        }
}
pantry = removeJSONFromArray(x, pantry);
```

```
private JSONArray removeJSONFromArray(String x, JSONArray pantry) {
    for(int i = 0; i < pantry.length(); i++) {
        String ing = pantry.getJSONObject(i).toString();
        if(ing.equals(x)) {
            pantry.remove(i);
            break;
        }
    }
    return pantry;
}
```

Refactor 3:
Replace Data Value with Object:

Previously we were passing many Boolean variables to the search method. We used the Replace Data Value with Object refactoring, using the FilterParameters object to pass the parameters to the search methods.

Old code with all Boolean variables:

```
+               boolean vegetarian = Boolean.parseBoolean(req.getParameter("vegetarian"));
+               boolean glutenFree = Boolean.parseBoolean(req.getParameter("glutenFree"));
+               boolean dairyFree = Boolean.parseBoolean(req.getParameter("dairyFree"));
+               boolean ketogenic = Boolean.parseBoolean(req.getParameter("ketogenic"));
+               boolean vegan = Boolean.parseBoolean(req.getParameter("vegan"));
+               boolean quick = Boolean.parseBoolean(req.getParameter("quick"));
+               boolean useInventory = Boolean.parseBoolean(req.getParameter("useInventory"));
+               boolean useExpiring = Boolean.parseBoolean(req.getParameter("useExpiring"));
```

New code with object creation for the parameters:

```
filters = new FilterParameters( req.getParameter("veggie") == null ? false : true,
        req.getParameter("glutenf") == null ? false : true,
        req.getParameter("dairyf") == null ? false : true,
        req.getParameter("keto") == null ? false : true,
        req.getParameter("vegan") == null ? false : true,
        req.getParameter("quickr") == null ? false : true,
        req.getParameter("useinv") == null ? false : true,
        req.getParameter("useexp") == null ? false : true
        );
```

Instead of having a ton of Boolean variables that were highly related, we put them all in one class. Additionally, all the logic related to the parameters is done within the class, so there is information hiding on how the parameters need to be used.