**UI and Domain Layer Separation Report**

**Nicole Burns**

# Domain Layer

The classes in the Domain Layer are Recipe, RecipeSearch and RecipeController. The Recipe class was built to enable creation of a Recipe object which could have certain attributes such as name, directions, ingredients, and total time to cook. At a very base level, it also has methods within it that allow retrieving and defining these attributes for a given recipe. The RecipeSearch class is built on top of Recipe and allows for (1) interaction with the storage .txt file (e.g. reading and writing to the file) and (2) generates the query results for use in the UI. Finally, the RecipeController sits on top of the RecipeSearch class and acts as a layer of insulation between RecipeSearch and the ApplicationUI (note that in a similar way, RecipeSearch acts as insulation between Recipe and RecipeController).

# UI Layer

The ApplicationUI Class is the UI Layer of the Recipe Management program. It pulls all needed logic information from the RecipeController class, which in turn receives that information from the RecipeSearch class. The ApplicationUI class does contain logic specifically related to movement through the UI.

# Discussion on Separation

Separation between the two layers was difficult for a few reasons that all boil down to the same concept. First, the ApplicationUI is made up of various text windows that allow reading of and writing to the text file. Second, depending on what the user does within the UI layer, conditional logic is applied in a few places. Finally, there are a few fields in the UI (i.e. a combobox dropdown) that need to dynamically update during the execution of the program depending on user interactions. Ultimately, all three of these items can be summarized simply: deciding what logic belonged to the UI layer and what logic belonged to the domain layer was challenging.

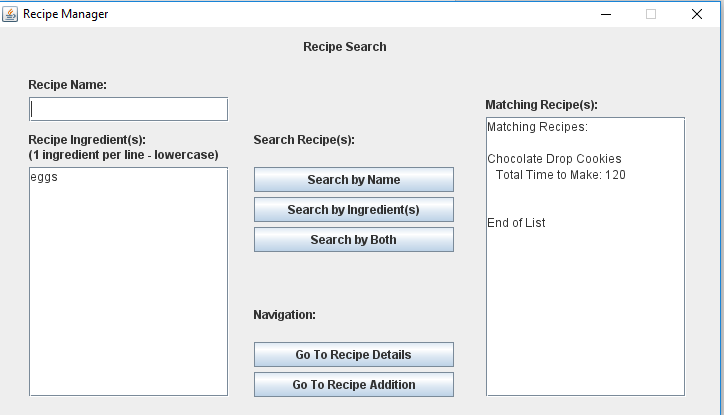
Let’s take a closer look at the challenges posed by having read and write areas in the UI Layer.

Figure - Text Areas

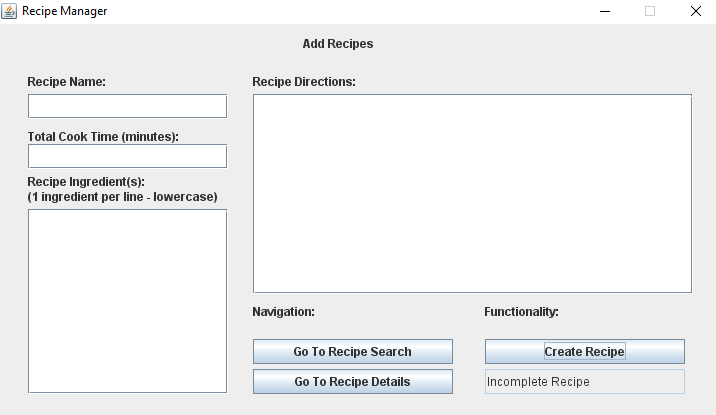
In Figure 1, there are several locations where a user can specify search parameters such as Recipe Name and Recipe Ingredients. These are free text fields that take in what the user has written and based on the search type selected (from the search buttons in the middle), will output a list of matching recipes. In this case, a decision was made to have the text field positioning and properties specified in the UI layer, and then have actionlisteners applied to the buttons to call various methods from the RecipeController class. These methods, in the domain layer, would then take the user input, and return a String for the Matching Recipe(s) report. The UI layer then picks up the string that is returned from the domain layer and sets the text of the Matching Recipe(s) field to that returned string. In this case, the separation of logics is such that consuming and processing the user input belongs to the domain layer, while user specification of what the domain layer should output (e.g. type of search) and when (i.e. when a button is clicked) belongs to the UI layer.

Figure - Incomplete Recipe

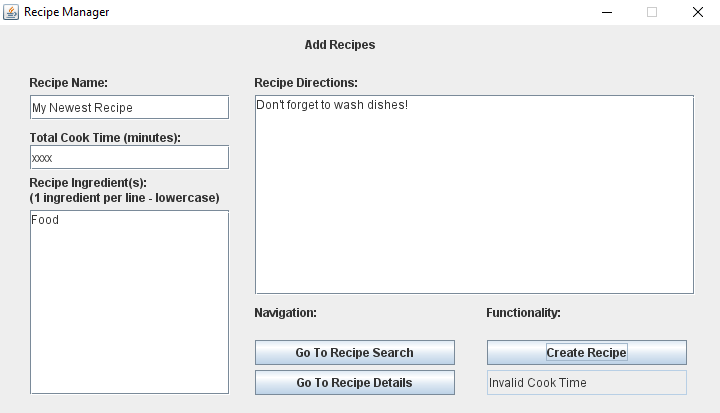
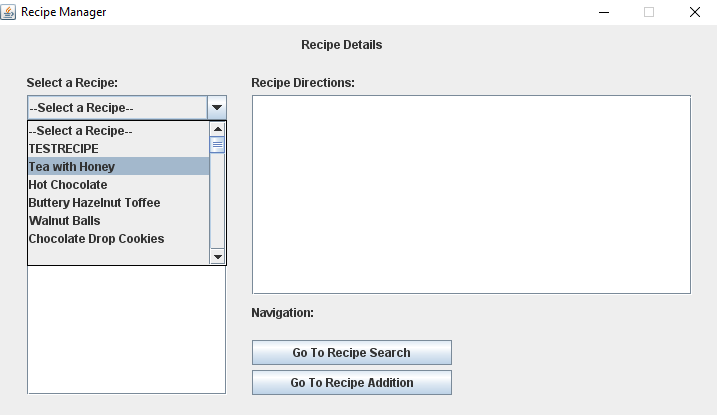
The Add Recipes panel is another place where separation of UI vs. Domain was particularly difficult. In this panel, I wanted to enforce requirements on (1) which fields needed to be filled out (e.g. all of them), (2) what kind of data (e.g. numeric vs. alphanumeric) was allowed for each field, and (3) ensure that a recipe couldn’t be created if it had the same name as an existing recipe. In all of these cases, the logic that prevents creation of the recipe and displays errors messages (i.e. in Figure 2 and Figure 3) actually belongs to the Domain layer, not the UI layer. The reason for this is that the requirements themselves are in place to ensure the integrity of the Domain Layer code (e.g. a few searches/methods pull back information based on the recipe name, and if multiple recipes with the same name existed, it would throw errors in the code), therefore it made sense that to decrease coupling, any error handling to enforce those Domain Layer requirements should be addressed within the Domain Layer directly. The output of that error handling is then passed back as a String notification to the UI Layer (e.g. with the messages “Incomplete Recipe”, “Invalid Cook Time”, etc.)

Figure - Invalid Cook Time



Finally, there was a bit of difficulty surrounding separation of logics for the combobox on the Recipe Details panel (Figure 4). In the “Select a Recipe:” field, the dropdown list is based on all of the existing recipes in the system. So each time a new recipe is added (i.e. in the Recipe Addition panel) this list needs to be refreshed. The separation was as follows: logic that determined what needed to go into the list was handled in the Domain Layer, and logic that actually refreshed the representation of the list in the UI through the combobox is handled in the UI Layer. The reason for this separation is that in order to populate the list, the relevant method needs to access attributes of the RecipeSearch class, by keeping that method’s logic in the Domain Layer instead of the UI Layer, some additional measure of encapsulation is achieved because the UI Layer can’t directly access attributes of the domain layer.

Figure - Dropdown List

Each of these three challenges were handled using the same guidelines: if the logic needed to address what objects the user saw (e.g. buttons), user input areas (e.g. editable vs. not), or navigating through the application, it was handled in the UI Layer. All logic that revolved around determining the contents of what was displayed, and logic that involved any reading from or writing to the text file was handled in the Domain Layer.