General Idea/Inspo
They say that a good way to find inspiration for a project is to introspect and look at your own needs, solve your own problems, so I tried to create a project that addressed a particular need in my own home.

Overall Analysis of My take:
My ***SmartKitchen***™ allows the user to manually input their culinary essentials into the program via an interactive GUI. Once done, the user is able to continuously update the state of their kitchen. Any time they donate food, consume or throw food out, they notify the program which reacts accordingly to display the updated information and state of their kitchen.
The user is able to design recipes, add items to their grocery lists, add items to their wishlists, view basic data about their recipes and their items, search through their storage for food and check what items are near expiration. Compared to what I initially sought to do, the final product ended up slightly different. I wanted to include more subtle analytics: information that the user probably didn't keep track of consciously (like average price, amount of times they've bought/eaten/thrown a product/recipe, different total metrics) but I quickly found out that the complexity and TIME CONSUMING NATURE of designing and implementing a GUI would make those impossible. Also, since my program was made to serve users, I quickly found myself having to take into consideration many aspects of user friendliness and utilization simplicity/convenience that I hadn't previously considered, all of which took time away from those more ambitious goals. Overall, the project is a functional and still fairly elaborate implementation of my original idea, however it didn't go as far or as in depth as I thought it would.

How?
I implemented an MVC paradigm to communicate between the database and the front-end visual components. Though I put a fair amount of time and thought into the conception and original design, I found myself often having to redesign or rethink certain aspects of the program/how I would implement certain features. The reason being that, as I mentioned, since the program was made to serve a customer, customer satisfaction had to be considered throughout. This ended up putting limitations on things I could/should do, as well as creating new needs for features I hadn't thought about. Also, the limitations of the GUI feasibility forced some features to be omitted for the time being. That said, the capabilities of JavaFX really impressed me; almost every little visual idea I came up with I was able to find documentation for and some built-in code to help with. Also, the process of designing for a customer really taught me how important design and REDESIGN are, and I think I've gained even more knowledge about the product development process. What's different from 1405 however is now I see how the added needs of a customer complicate the process but also make it more enjoyable.

Computational Thinking

Abstraction:

Abstraction of the Problem:
I began by abstracting and taking a birds eye view of my project like so:

> *"The program stores a host of user input, and is open to updated information about that data set. The program visually returns select information from this data set depending on what info the user requests to obtain/see."*
> - Malick Sylla

Abstraction of the Implementation:
Then, I abstracted the steps of my implementation:

1. Designing the GUI using an online editor
2. Complementing the visual design with hand drawn additions for functionality (components, transitions, etc)
3. Creating the Product hierarchy of classes
4. Designing the model
5. Attaching the model to a view(s)
6. Attaching a view(s) to the controller

Abstraction of the Design:
This is where I came up with the structure of my program and its data. I figured that the program would work with some sort of hierarchy; with regards to both the frontend component as well as the backend. I decided to implement a class structure that saw **Product** at the top, with subclasses **Perishable and NonPerishable**. I had a standalone class **Recipe**, and all these classes implemented the **Consumable** interface. I also had a **final class Ingredient,** since this wasn't a class I wanted to be extended.

In terms of the GUI, I had a hierarchy of 10 stages, each with its own **scene**. These scenes were each created with different Panes, however I did not create 9 different panes (I will elaborate below) because in my abstracted view of the problem, I was able to recognize a pattern of functionality and design.

Pattern Recognition:

From the Tutorials, I knew that if poorly designed, GUI's would involve a humongous amount of repeated code. So, from the very beginning pattern recognition was a crucial part of my thought process and my approach to the project.

1. **Add Item Pane**

A major recognition of a repetitive process was in my implementation of the Add Item Pane. The AddItemPane is the GridPane I created to have the user be able to input a new product into the database. This pane is just a simple layout with text fields for user input, as well as labels to prompt the user for the correct information. I included some ActionListeners directly in this pane: the action I was listening for had nothing to do with the data of the program, rather it had to do with the appearance of that specific pane in response to something. This allowed me to declutter my controller, as well as abstract some implementation directly in the pane instead of 4 times for each instance of the AddItemPane in my project. I had the pane take in a boolean input in its constructor, so I could differentiate between the two different types of AddItemPane. Since adding to a grocery required me to know where the user wanted to add the item once it was bought (not the case when the user is adding to the fridge/freezer/pantry, as this would be redundant and frustrating for the user. This is actually a good example of having to trade off brevity and less work on my end for user convenience.) , I needed to display an extra text field and label, but only when it was being used as a grocery input. So, I listened for that possibility with a boolean, and added that aspect accordingly. Yes, this did force a little more knowledge of implementation on whoever is using this, but I assessed it as being very minor in comparison to the time/ressources I saved by not having to create an entirely different pane. Also, since stages have the ability of setting style, I was able to have each AddItemPane have its own style and background color without hardcoding that information in the class itself. The end result: one class was able to be instantiated into 4 different stages, capitalizing on their shared ressemblance and functionality

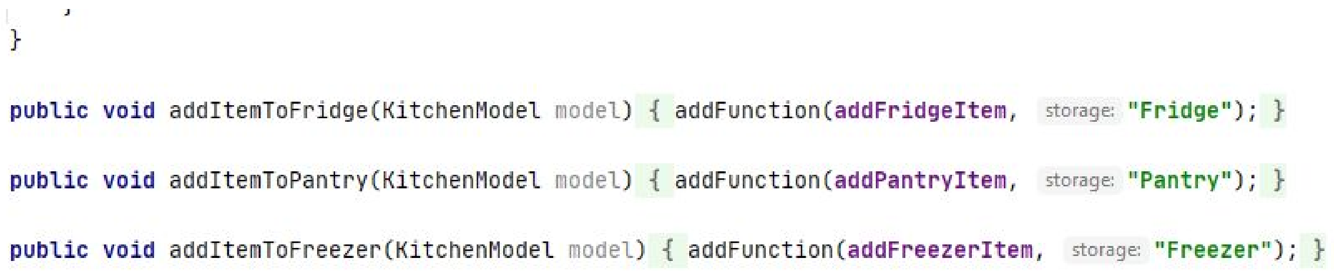
2. **Content Pane**

A very similar process drove my design of the ContentPane class, however this time the process was even more seamless because all 3 uses of the ContentPane had exactly the same functionality and resemblance. Again, I did include some ActionListeners within the class itself,only because they didn't actually handle any data; they listened for aspects of change in the pane itself. Again, this allowed me to abstract that implementation into one class instead of cluttering my controller and view with this little detail. The most interesting and noteworthy thing about the ContentPane was a pattern I was able to recognize, but that I wasn't able to capitalize on. The pane has 7 buttons on it, and my program has 3 stages that use 3 different instances of the pane. So, my controller was having to listen for 21 button events. I didn't like this however, because each of the actions performed by the buttons was essentially the same; they just did it with different information about the model. What I tried to do to remedy that was create one pane (vBox or HBox or Grid) with all the buttons (aka ButtonPane), then have the ContentPane take a Pane object in its constructor, and display it on its own container. That way, my view could create one instance of a ButtonPane containing 7 buttons, and pass that instance to each of the 3 ContentPanes. The effect this would have on my code is that now I'd only listen for 7 buttons, since they were shared by each of the ContentPanes. I figured that dependending on what stage was currently showing, I could tell the view that a specific button had been clicked AND that the fridge was open, so the view would know what data to fetch from the model. This would have reduced the length and clutter of my Controller class immensely, except I ran into a

problem. After trying this solution, I realized that JavaFX didn't actually permit the sharing of 1 component across many containers SIMULTANEOUSLY. So, only the ContentPane that was last initialized would be able to display and "own" the buttonPane. I tried to fix that by reassigning the ButtonPane to a different ContentPane when needed, however this caused problems when that pane was then added to a scene and a stage. Java would not allow this reassignment of a scene and it produced a runtime error. So, after trying different ways and asking for Dave's help, I abandoned the idea and did it the "long" way. This actually helps the clarity of my code if I do say so myself; that drastic of an abstraction would not have been obvious to those reading my code, so for maintainability sake I think this was actually the best move.
.

4.Add Function and Search Function

To add an item to the model's storages, the user could either add to the fridge, the freezer or the pantry. I was able to recognize that the process for these 3 was pretty similar, so I had one private function in the view take care of the adding process depending on the location specified by 3 public helper functions that the controller interacted with. This lets whoever maintains the controller to easily differentiate which function they want to make use of (addItemToFridge/freezer/pantry), effectively making use of **abstraction** as well.



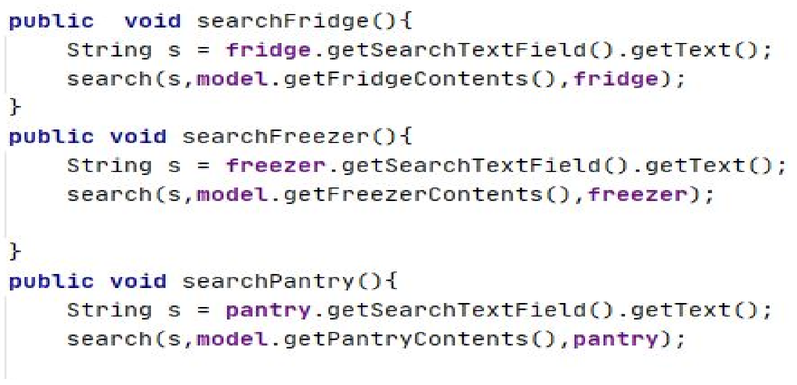
```
}

public void addItemToFridge(KitchenModel model) { addFunction(addFridgeItem,  storage: "Fridge"); }

public void addItemToPantry(KitchenModel model) { addFunction(addPantryItem,  storage: "Pantry"); }

public void addItemToFreezer(KitchenModel model) { addFunction(addFreezerItem,  storage: "Freezer"); }
```

I applied the same process to my search function:



```
public  void searchFridge(){
    String s = fridge.getSearchTextField().getText();
    search(s,model.getFridgeContents(),fridge);
}
public void searchFreezer(){
    String s = freezer.getSearchTextField().getText();
    search(s,model.getFreezerContents(),freezer);

}
public void searchPantry(){
    String s = pantry.getSearchTextField().getText();
    search(s,model.getPantryContents(),pantry);
```

5. Inheritance

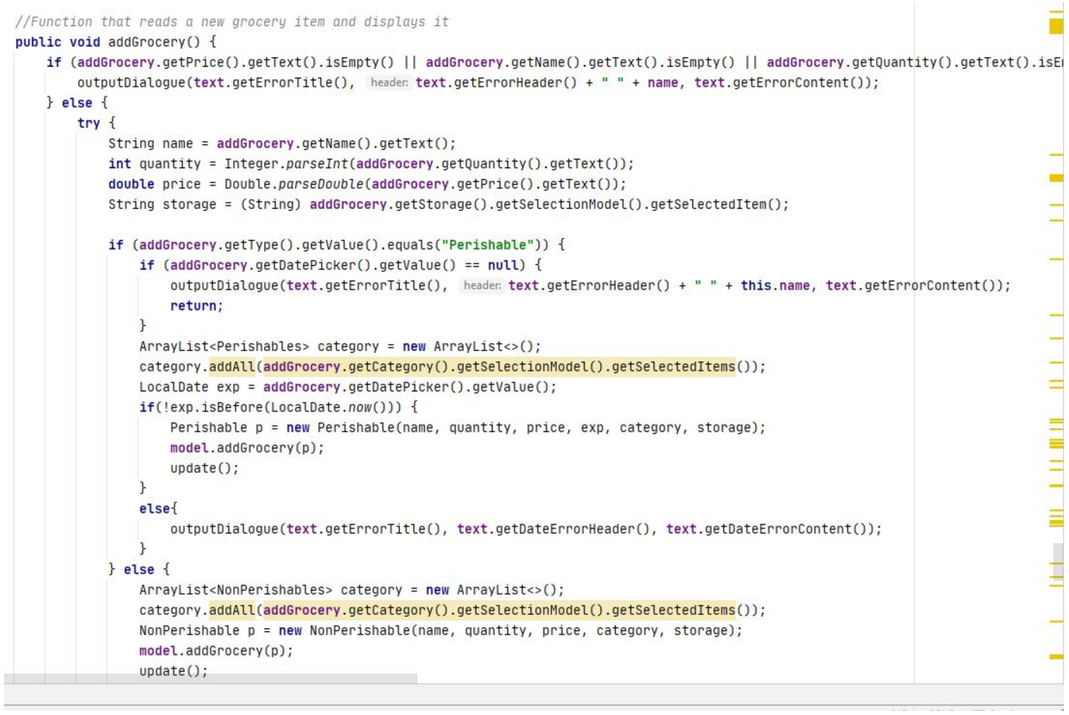
Recognizing the “is-a" relationship between my backend classes as described above was the basis of the hierarchical structure. I was able to have 2 subclasses extend a superclass - effectively capturing shared functionality with less code. This also had **polymorphic applications**, especially since I capitalized on the common “consumable” nature of my data. They implemented the **Consumable** interface, and as you’ll see in the code were frequently type casted or treated as one another for various purposes (such as having an ArrayList<Consumable> fridgeContents - allowing both recipes and products to be displayed easily etc).

Decomposition and Algorithm Design

I find that algorithm design is intertwined with every step of the computational thinking process, since most of your thought is just baby steps towards coming up with an algorithm. In particular, decomposition of my problem heavily implied algorithm design, since by decomposing a problem I was essentially creating a smaller problem that now needed an algorithm. The majority of the more complex algorithms were contained in my model and view.

View

A lot of the algorithms in the view relied heavily on sequential processing. Essentially, I would decompose a problem such as “adding a new product” into the steps that needed to happen first, and those that followed. Largely, my algorithms would read data from the GUI by calling on the getMethods in the appropriate container, and creating objects from this data and passing it to the model as you see here:



```
//Function that reads a new grocery item and displays it
public void addGrocery() {
    if (addGrocery.getPrice().getText().isEmpty() || addGrocery.getName().getText().isEmpty() || addGrocery.getQuantity().getText().isE
        outputDialogue(text.getErrorTitle(),  header: text.getErrorHeader() + " " + name, text.getErrorContent());
    } else {
        try {
            String name = addGrocery.getName().getText();
            int quantity = Integer.parseInt(addGrocery.getQuantity().getText());
            double price = Double.parseDouble(addGrocery.getPrice().getText());
            String storage = (String) addGrocery.getStorage().getSelectionModel().getSelectedItem();

            if (addGrocery.getType().getValue().equals("Perishable")) {
                if (addGrocery.getDatePicker().getValue() == null) {
                    outputDialogue(text.getErrorTitle(),  header: text.getErrorHeader() + " " + this.name, text.getErrorContent());
                    return;
                }
                ArrayList<Perishables> category = new ArrayList<>();
                category.addAll(addGrocery.getCategory().getSelectionModel().getSelectedItems());
                LocalDate exp = addGrocery.getDatePicker().getValue();
                if(!exp.isBefore(LocalDate.now())) {
                    Perishable p = new Perishable(name, quantity, price, exp, category, storage);
                    model.addGrocery(p);
                    update();
                }
                else{
                    outputDialogue(text.getErrorTitle(), text.getDateErrorHeader(), text.getDateErrorContent());
                }
            } else {
                ArrayList<NonPerishables> category = new ArrayList<>();
                category.addAll(addGrocery.getCategory().getSelectionModel().getSelectedItems());
                NonPerishable p = new NonPerishable(name, quantity, price, category, storage);
                model.addGrocery(p);
                update();
```

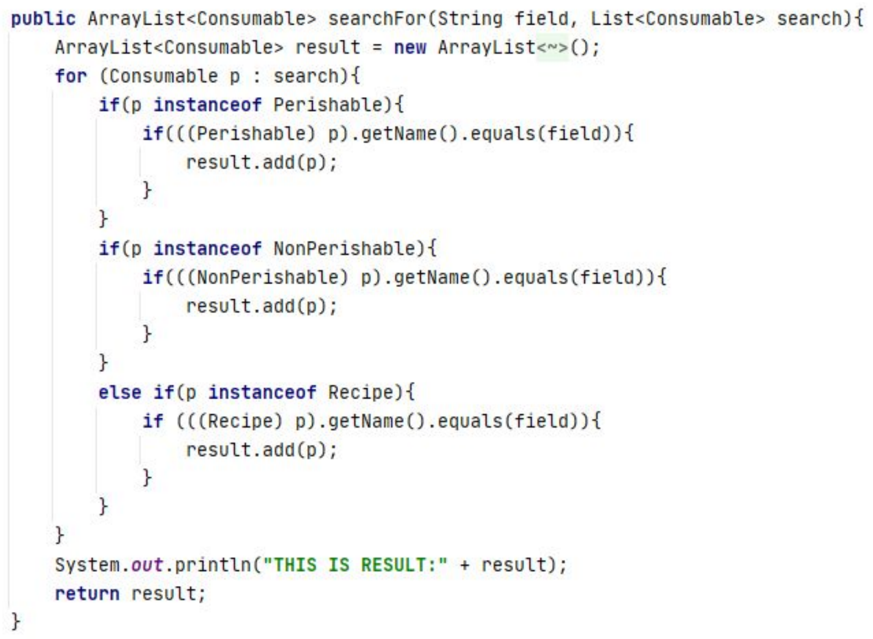
Now part of the sequence was error checking; there were specific points in the flow of reading data that had the possibility of generating runtime errors, so they had to be addressed. Errors like null pointers, incompatible data type entered, or either a closure of the stage or no entry at all. These were effectively handled with try catches and if statements. After passing data down to the model, the view would need to update to reflect the changes in the model. This was the general sequential blueprint for much of the algorithm design within the view. The more intuitive and logical algorithmic design was in the model

Model

The model handled much of the more complicated processes like cooking, searching and buying. Though I wouldn't consider any of the logic particularly hard, it did entail a fair amount of reflection as to what was the best and clearest way to design the algorithms.

Searching:

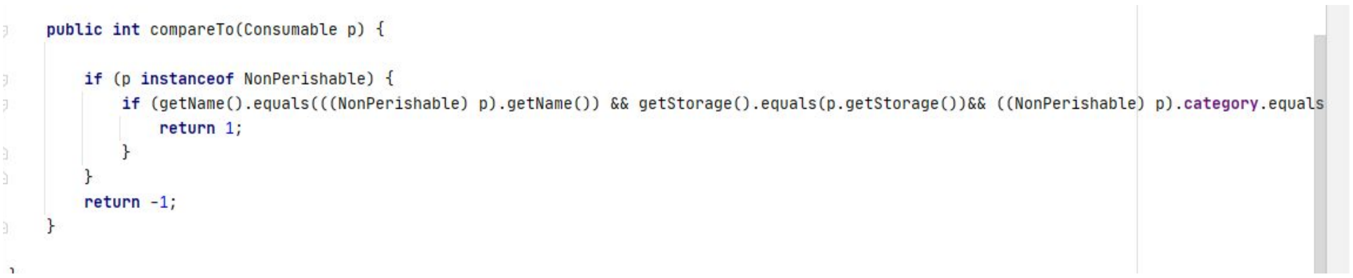
My searching function was the basis for my cooking one as well, and the inspiration behind my buying one, so I'll address it first. I needed a function that could search through the kitchen's contents and display the corresponding results.



```java
public ArrayList<Consumable> searchFor(String field, List<Consumable> search){
    ArrayList<Consumable> result = new ArrayList<~>();
    for (Consumable p : search){
        if(p instanceof Perishable){
            if(((Perishable) p).getName().equals(field)){
                result.add(p);
            }
        }
        if(p instanceof NonPerishable){
            if(((NonPerishable) p).getName().equals(field)){
                result.add(p);
            }
        }
        else if(p instanceof Recipe){
            if (((Recipe) p).getName().equals(field)){
                result.add(p);
            }
        }
    }
    System.out.println("THIS IS RESULT:" + result);
    return result;
}
```

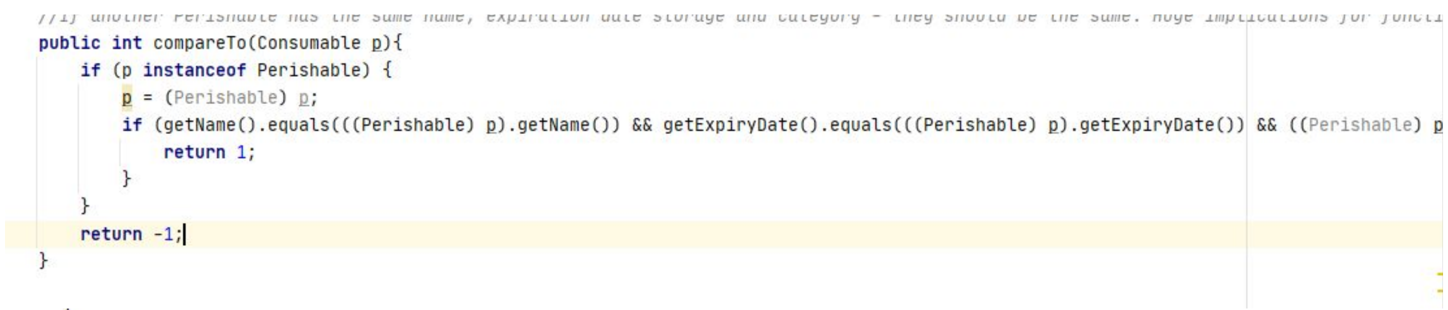
In a way, this was slightly different than searches performed previously in 1405 for example. Since my search function was searching for a string within a list of objects, and also looking for all applicable values, I had to brute force it in O(n). I simply didn't have a way of sorting the consumables, and parsing through them efficiently. That was mainly due to not having quite enough time to implement a different strategy: Abstract Data Types (I will talk about this later). It got me thinking about how a sort would work, and then I realized I should have CompareTo's for my classes.I wasn't too happy with the current search, so I will work towards improving it in my own time.

<u>Buying</u>
My buying function has a few things to take into consideration. It involved removing an item from the grocery list, and adding to the appropriate storage. However, this couldn't be done blindly, since if the product was already in that storage area, the user was essentially refilling their product and it shouldn't be added as a new product; rather the quantity of the product had to be increased instead (**though there is a flaw to this I just realized, I will discuss it below).** This was important to avoid the fridge displaying the same item multiple times as different object's - this would be an inconvenience/frustration for the customer. I needed to check if any of the storage lists ***contained*** the product being added. Well, using the built in **contains(object o)** method, the answer would always be no, since they were technically two different objects. The reflection above made me realize I had to change what it meant for two Products to be equal. More specifically, what it meant for two subclasses of products to me equal: Perishable and non Perishable. So, I implemented a compare to in each subclass of Product:

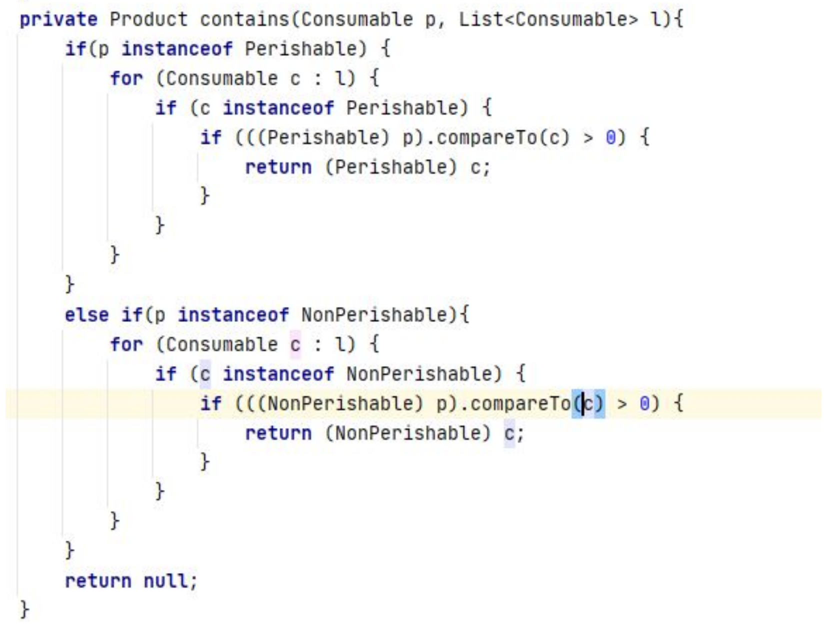


```
public int compareTo(Consumable p) {

    if (p instanceof NonPerishable) {
        if (getName().equals(((NonPerishable) p).getName()) && getStorage().equals(p.getStorage())&& ((NonPerishable) p).category.equals
            return 1;
        }
    }
    return -1;
}
```



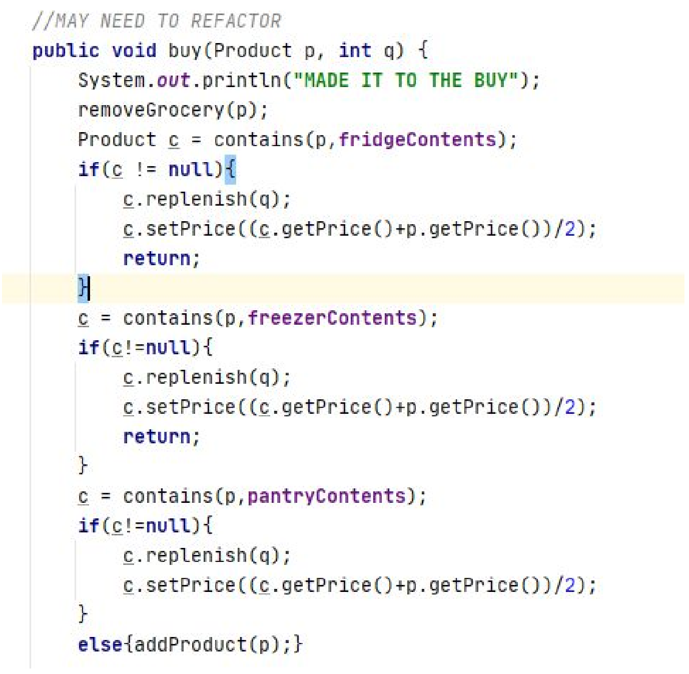
```
//if another Perishable has the same name, expiration date storage and category - they should be the same. Huge implications for functi
public int compareTo(Consumable p){
    if (p instanceof Perishable) {
        p = (Perishable) p;
        if (getName().equals(((Perishable) p).getName()) && getExpiryDate().equals(((Perishable) p).getExpiryDate()) && ((Perishable) p
            return 1;
        }
    }
    return -1;
}
```



```
private Product contains(Consumable p, List<Consumable> l){
    if(p instanceof Perishable) {
        for (Consumable c : l) {
            if (c instanceof Perishable) {
                if (((Perishable) p).compareTo(c) > 0) {
                    return (Perishable) c;
                }
            }
        }
    }
    else if(p instanceof NonPerishable){
        for (Consumable c : l) {
            if (c instanceof NonPerishable) {
                if (((NonPerishable) p).compareTo(c) > 0) {
                    return (NonPerishable) c;
                }
            }
        }
    }
    return null;
}
```

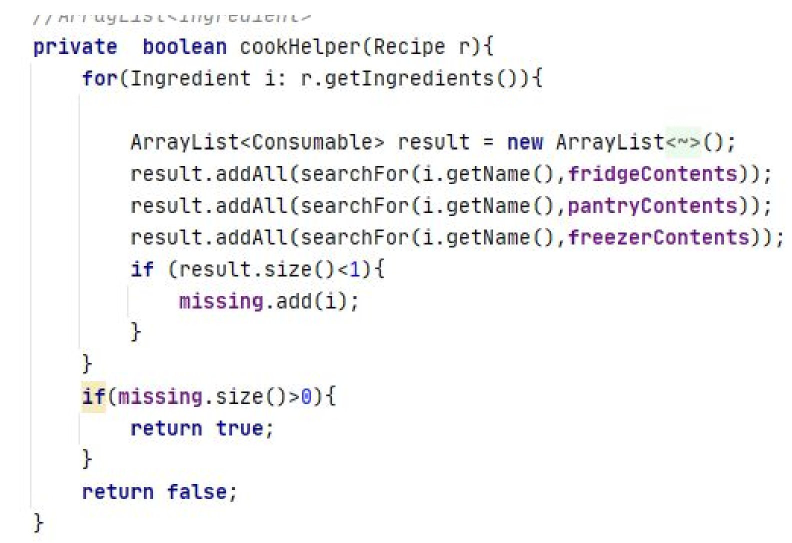
Then, I was able to check if the new product was already in one of the storages(making it not a new product, just a refill). I also needed its match returned so I could update its quantity directly instead of creating another search for it.

Finally, my buy function would simply make use of contains to determine whether or not a value had been returned, and handle that outcome appropriately.


```java
//MAY NEED TO REFACTOR
public void buy(Product p, int q) {
    System.out.println("MADE IT TO THE BUY");
    removeGrocery(p);
    Product c = contains(p,fridgeContents);
    if(c != null){
        c.replenish(q);
        c.setPrice((c.getPrice()+p.getPrice())/2);
        return;
    }
    c = contains(p,freezerContents);
    if(c!=null){
        c.replenish(q);
        c.setPrice((c.getPrice()+p.getPrice())/2);
        return;
    }
    c = contains(p,pantryContents);
    if(c!=null){
        c.replenish(q);
        c.setPrice((c.getPrice()+p.getPrice())/2);
    }
    else{addProduct(p);}
```

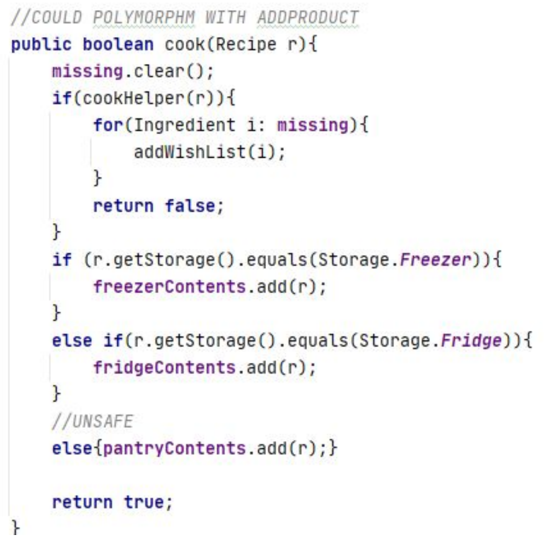
Cooking

My cooking function dealt with putting a recipe into the appropriate storage, just as if a customer had prepared it. However, I didn't want to prepare the recipe if the user didn't have the necessary ingredients. I also wanted to display what ingredients were missing, and add them to the wanted section of the grocery list. To do this, I would again need to search for the ingredients within the kitchen, so I made a helper function that made use of the search function above:


```java
//ArrayList<Ingredient>
private  boolean cookHelper(Recipe r){
    for(Ingredient i: r.getIngredients()){

        ArrayList<Consumable> result = new ArrayList<~>();
        result.addAll(searchFor(i.getName(),fridgeContents));
        result.addAll(searchFor(i.getName(),pantryContents));
        result.addAll(searchFor(i.getName(),freezerContents));
        if (result.size()<1){
            missing.add(i);
        }
    }
    if(missing.size()>0){
        return true;
    }
    return false;
}
```

If the search function returned no hits, the ingredient was added to my **missing** list. Then, if the **missing** list was greater than 1, it returned true.



```
//COULD POLYMORPHM WITH ADDPRODUCT
public boolean cook(Recipe r){
    missing.clear();
    if(cookHelper(r)){
        for(Ingredient i: missing){
            addWishList(i);
        }
        return false;
    }
    if (r.getStorage().equals(Storage.Freezer)){
        freezerContents.add(r);
    }
    else if(r.getStorage().equals(Storage.Fridge)){
        fridgeContents.add(r);
    }
    //UNSAFE
    else{pantryContents.add(r);}

    return true;
}
```

The cook function would begin by clearing the missing list (thereby eliminating all remnants of previous recipes), and then call its hepler (since it directly populated missing). Missing items were added to the wishList, and if nothing was missing then the function would move the items to their appropriate storing places. The function returned a boolean to let the view know whether to proceed if the cooking was successful, or notify the user they were missing items if the cooking was not successful

Problems/Pitfalls/Improvements/**Extensions** of my Implementation

1. <u>Use of Abstract Data Structures</u>

I noticed too late that there was a great opportunity for the use of my own Abstract Data type and ADS. I could have made a collection of consumables that I could have been able to sort, upon adding them by use of their compareTo's (which I should have made Consumable implement) . I could have changed what it meant for my List to contain an item etc. This would have made my search process much more efficient, and allowed for an extension on the search component to be more seamless. It could also have led to an extension where I sorted the kitchen based on different metrics like price, quantity, value, purchase frequency, characteristics and more.

2. <u>Searching</u>

I am not satisfied with my search at all, and not only for the reasons I mentioned above. I do feel like it could have been optimized if I didn't rush it, but also I feel like it fails to account for certain real life situations. However, part of this is due to the tricky nature of the content I am trying to model. What it means for a product to be equal to another in real life is a very fluid concept; a product can have plural identities. It would have been better then for my compareTo to compare based on product type not name, if I was cooking for example. I don't care what kind of milk chocolate I have for my Chocolate Chip Cookies, I just need milk chocolate. In my fridge I may have some from Hershey's that I identify by the brand, or some generic milk chocolate brand from Costco that I just refer to as milk chocolate. In this case, they both share the same function but are identified differently, which hurts the searching process for both the recipes and the general searching. However, in a different situation, let's say when adding to a grocery list, it is in fact the name of the product that matters, not its type. I want to buy more Hershel's chocolate, and its quantity should be increased, not that of all the milk chocolate I own (since Hershel's is my identifier). THere are even more scenarios like this that make it impossible to account for a perfect search every time, however as an extension I would definitely expand what it means for an object to be another DEPENDING on the situation, and then perform the search on this basis.

There is also the fact that my search is VERY key sensitive. That is an improvement I could easily make with a little bit more time, so it's a very attainable improvement/extension. I would also like to be able to dynamically search: have the results pop in as the user types, instead of just at the end when they click search. This would increase the amount of searches required, but it would definitely add to the user experience.

Misc

I could have had one enum with just more values, and then depending on what type of product was being added, only display certain enum values. This would have lessened the need for typecasting and the dichotomy between categories, and maybe make some things a little easier.

I should add a remove recipe button or key listener, so the user can remove from their cookbook.

My remove-product function should not search through the entirety of the storages, it should just rely on the storage type of the incoming Consumable (like the addProduct function does). Since java does not throw errors for removing a non-existent element from an array, this comes with no costs.

The cook function should also return missing if there aren't enough of a specific item. Currently, if there is any occurence of that item it is declared not missing, though the quantity required by the recipe could be more than the user has on hand.

Expiration dates aren't changed when an item is refilled directly. Basically, when the user moves an item directly from the storage to the grocery then buys it, the product's expiry date isn't changed. I'm not sure if this was a subtle omission though it seems glaring to me. It again reflects the nature of what I am modelling, since this problem also heavily involves what it means for two products to be the same in the eyes of the user (is expiration date a differentiator between two otherwise identical products? Sometimes or always?)