Design Document

Updated Specification

gyst is a web application that organises your household inventory and assists you in your everyday life. Different requirements such as food and household items can be kept track of in terms of expiry dates (if applicable) and quantity. There are two main lists of items defined. First, we have the list of needs (shopping list) of the user. The second list of items is the inventory of the user. Each item in the inventory will have attributes such as expiry date (if applicable), and the quantity. In the beginning, the list of needed items will be initialized, and the user can add items to that list by logging items one by one. The user can also remove items from their inventory or shopping list one item at a time. The user can see their updated inventory and shopping lists. There is an option for the user to save their inventory and shopping list for later use. This website will allow the user to be efficient and organized by decluttering their life.

Changes from Phase 0 to Phase 1

Our most prominent change from phase 0 and phase 1 was how *gyst* interacts with the user. In phase 0, we implemented *gyst* such that a user interacted with it via the command line. In phase 1, we switched over to a website. Our biggest transition (though not fully complete) was from a Command Line Interface to a rather user-friendly web form interface. We focused on the construction and the layout of the website and are planning to connect the functionalities of the interface to use case features through a middle-man such as our controller class ListManager.

Additionally, in phase 0, after the user exited the program, the items they entered into their inventory and shopping list were not saved. In phase 1, we incorporated data persistence/serialization via CSV files so that the user has the option of saving their inventory and shopping list.

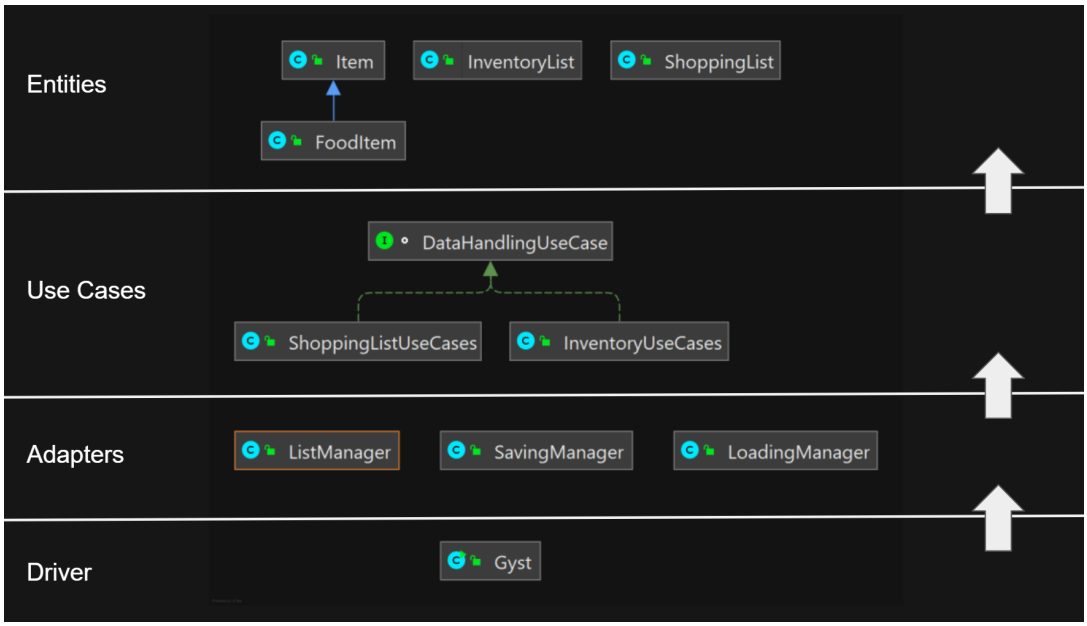
In phase 0, the expiry date was an integer representing the number of days until an item expired. In phase 1, we changed the datatype to the LocalDate class to better reflect the format of an actual expiry date (YYYY-MM-DD). We also implemented a method that returns true if today's date is within a given number of days of the item's expiry date. This method will later be used to notify the user when an item is close to expiration.

Major Design Decisions

We started the second phase of the project by planning to use databases for serialization purposes. Yet, on our way, due to unexpected costs and infeasibilities we had to switch to local csv serialization. This transition came with a dilemma. If we were to update the data source automatically at each change of the properties of the program (InventoryLists and ShoppingLists), we would need to iterate through each line of the csv files to save, process and retrieve data at each call of inventory/shopping list updating methods. On a larger scale, this would require more and more computing power and slow down the process of interacting with the data source. We solved this issue by letting the user save changes after each session, which made the gateway classes make changes on the data source only when the user wants to save the progress/changes. When the program is initialized for the first time, the template csv is loaded to the inventory list and the shopping list. After a series of changes executed by the user, the user can choose/press save. Receiving the save command from the user, the system transfers the data stored in the entities to a parallel csv file and writes it on the template csv files provided in the resources folder.

Another major design decision we came across along the way was whether or not to create concrete class implementations for the gateway layer (interface adapters). Consequently, it led to the discussion of the possibility of changing the location of the instances of the use cases. Our first implementation for the loading and saving functionalities was through a series of methods defined under use case classes and the controller class. After long debates regarding the formation of concrete gateway classes, we decided to create separate gateway classes in order to adhere to clean architecture principles and follow the diagram presented in the Web App Tutorial for handling requests with the presence of gateway. With the help of the creation of gateway classes, our structure perfectly fits the clean architecture principles.

Clean Architecture



In the diagram above, we see where inheritance and interface implementation occurs in each layer. In the entities layer, we see that FoodItem is a child of Item, and it inherits all methods and attributes from the Item class. In the Use Cases layer, we can see that there is a DataHandlingUseCase interface that is implemented by the ShoppingListUseCases class and the InventoryUseCase class.

Scenario Walkthrough

Uncle Drew just came home from the grocery store. He bought 99 bananas and wants to update his gyst inventory. Uncle Drew opens the *gyst* website. The gateway, LoadingManager class, loads the items currently in Uncle Drew's inventory and shopping list from the CSV files. These items are displayed on the website. To add the bananas to his inventory from the website, Uncle Drew enters "bananas" for the input to item name, 99 for the input to quantity, and "2021-11-27" for the input to expiry date. Then, he presses the "Add item" button.

Now, the controller, ListManager, calls a method in the InventoryUseCases class to handle the addition of the item name, quantity, and expiry date to the inventory. This method creates an Item entity with the given item name, quantity and expiry date. Then, it calls a method in the entity, InventoryList, which adds the Item entity to the inventory list. The website is then updated to display Uncle Drew's updated inventory with the 99 bananas. Finally, Uncle Drew presses the "Save" button and the gateway, SavingManager, interacts with the InventoryUseCases and ShoppingListUseCases to save Uncle Drew's inventory and shopping list. Uncle Drew is satisfied and exits the website to go enjoy his 99 bananas.

Our design followed the Clean Architecture principles. We designed *gyst* so that the user interacts with the website. This ensures that the user can navigate through the inventory system easily, without directly interacting with the high level layer elements such as Entities.

In addition, we implemented the use case classes under two categories: ShoppingListUseCases and InventoryListUseCases. If we were to complete the implementation through the classic use case class design, there would be duplicate methods that work in the same manner for both the shopping list and the inventory list. With this choice, first, the controller can directly refer to the corresponding list type. Then, the controller can call the method that is required for the specific use case to execute the use-case specific workflow.

The Dependency Rule is consistently followed when interacting with details in the outer layer. For example, when interacting with the database, only the gateways, LoadingManager and SavingManager, interact with the CSV files. These gateways also only interact with the use cases, InventoryUseCases and ShoppingListUseCases, in order to update or get data from the InventoryList and ShoppingList entities. The gateways do not interact with these entities directly.

SOLID Principles

We consistently checked whether our implementation/design choices adhere to the five SOLID principles of object oriented design.

Our design adheres to the single responsibility principle. For example, the FoodItem class only has the responsibilities related to a food item, such as item name, quantity, expiry date, updating quantity, and comparing expiry date to today's date.

Our design adheres to the open/closed principle and Liskov substitution principle. For example, instead of adding a feature/attribute (that only Foods have) to our Item class, we implemented another class, FoodItem, that extends the Item class and adds the expiry date property.

Our design adheres to the interface segregation principle. For example, we have a DataHandlingUseCase interface that contains one saveList() method. This method is relevant to saving the data of a list (i.e., the inventory and shopping list).

Packaging Strategies

The packaging strategies we considered were packaging by layer versus packing by feature. If we packaged by feature, we would group classes related to inventory together and classes related to shopping list together. However, there are common classes between inventory and shopping list such as the Item class. We found packaging by layer intuitively easier to understand as the layers of clean architecture are clearer to us. Therefore, we packaged by layer.

Design Patterns

Our group implemented the dependency injection design pattern. For example, when adding an Item or FoodItem to an InventoryList object, instead of hardcoding the type Item or FoodItem into the InventoryList, the InventoryUseCases class creates the Item and FoodItem objects and injects them into the InventoryList. This way, we avoid a hard dependency between the Item/FoodItem objects and the InventoryList. We allow both the superclass, Item, and the subclass, FoodItem, to be added to the InventoryList.

If we had more time, we could implement the memento design pattern. Our project involves creating a “shopping list” for inventory items. A user is prompted to type their items one by one in our command-line interface. However, a user may want to undo one of the items they typed.

In order to implement the memento design pattern, we will use the classes InventoryList and ShoppingList as the originator classes, these classes are responsible for saving each item that is added to the inventory. We will then create a memento class that is nested inside the originator. This will allow the originator to access the fields and methods of the memento class even if they are private. The memento class will be immutable and the data will be passed to it once through the constructor. The originator will have a method called PreviousItems that will create new mementos every time a new item is added. We also need to create a caretaker class that will keep track of the originator’s history via an ArrayList. The caretaker class will be able to store and retrieve previous versions of the inventory. We can create undo and redo buttons in order to allow the user to take advantage of this feature.

We could also implement the command design pattern if we had more time. In our current implementation, we let our users create items by letting them choose a name, expiry date, and quantity. In terms of name, even though we need objects with the same name to refer to the same Item object, we are unable to handle typos, therefore, assuming valid input from the user. For example, within the scope of our current implementation: one could add an item called 'bananas' to the shopping list and an item called 'banana' to the inventory list. These two, somehow, should refer to the same type of object, yet there are no relationships between them. This problem seems solvable through the use of Command Design Pattern.
First, we can think of the process of a user creating an Item (through entering inputs: name, quantity, and expiry date) (when applicable) as a user sending a command to the system. Since the process of item creation differentiates only between FoodItem objects and non FoodItem objects, we can create a process/algorithm to handle these two cases separately.

Then, we can design an abstract base class for handling each command provided by the user. With this design pattern, commands can be considered objects with states such as passed, shared, loaded, etc. Therefore, by adding a separate categorization layer with the interface with the execute method, we can make our system understand some commands as valid and some as invalid. We can also handle some exceptions, by letting the user create a different Item with a

name that is invalid to our first layer of categorization if the user insists on having that specific name. By doing so, just like Java's type declarations, we are passing the buck to the user since they insist on handling the case their way. (See that this will not result in a crash in our program, it will only make the user have two items with similar names, maybe they are planning something above the main scope)

Progress Report

We have created a GUI in the form of a website, and we are currently working on connecting our website to our Java code (ex. We have been consulting tutorials and course resources). We are looking into Java Spring Boot, and our main priority is to adapt our code so that it can be read as a Spring Boot project. This will allow it to connect to our website. We are also going to be working on getting push notifications working to notify the user when the quantity of an item is low or if the expiry date of an item is approaching. We are also going to be implementing a method that moves an item from the shopping list to the inventory. We will also be adding the feature of having multiple batches of one item if the items have different expiry dates (ex. If they were bought at different times).

Questions:

- How should we connect our website to Java?
- Should we expect the course to set up a database server for us students to use in the future?

What Worked Well with Our Design:

- We followed the dependency rule of Clean Architecture (as shown in the UML diagram).
- We consistently checked whether our implementation/design choices adhere to the five SOLID principles of object oriented design.
- We organized and packaged our code efficiently and effectively.
- The dependency injection design pattern worked well.

What Each Group Member has Been Working On:

- Ayanaa: Unit tests, design document, researching database material.
- Jennifer: Updated datatype of expiry date from integer to LocalDate class. Revised database.
- Alissa: Creating website and figuring out website and Java connection
- Ali: Databases, local serialization, and applications of cl. arch. principles.
- Aamishi: Unit tests, updating CRC cards, researching website
- Sam: Databases, website, figuring out Action feature on GitHub

What Each Group Member is Going to Work on Next:

- Ayanaa: More unit tests, and implementing the notification system
- Jennifer: Implement login system in Java
- Alissa: Add login system and notifications to website, connecting website to Java
- Ali: Saving/loading information for the login system, assigning lists to users, consistency
- Aamishi: Exception handling, unit tests
- Sam: Move item from shopping list to inventory, login system