Retail Simulator

Use Case Description

This project's purpose is to develop a single-player Retail Simulator game that entertains users during their free time while enhancing their resource management and strategic planning abilities. The game simulates the daily challenges of running a retail store, focusing on key aspects such as inventory management, customer satisfaction, and financial stability. The user will be able to purchase items, confirm purchases, add stock to the store, remove stock from the store, and view the store's sales. They should expect a substantial amount of planning to manage the store's stock, budget and reputation, which ultimately impacts their profitability and ability to pay rent.

The project's goal is to help kids and teenagers, the target audience, learn how to manage resources, handle business operations, and understand the relationship between customer satisfaction and financial success through hands-on experience in a risk-free, entertaining environment. The game is designed to make learning about retail operations engaging, with quick and easy-to-understand gameplay.

Class List

1. **Printable**: A pure abstract class/interface that defines the contract for any printable entity in the project. It contains a single pure virtual function print(), which must be overridden by any derived class. This class allows polymorphism by ensuring that all subclasses implement their own print() version, ensuring a uniform way to display relevant information across different objects.
2. **StoreBase**: A base class that provides a foundational structure for any type of store that shares common attributes and behavior such as the number of different items, a list representing the store's inventory, and a list of item names. It has methods that allow users to manage their store's inventory. The class acts as a framework that simplifies project's expansion if the need to implement more store types arise.
3. **Store**: A specialized class derived from StoreBase, representing the store that the user manages. It has additional attributes that track dynamic aspects of the game, such as the current day in the simulation, the number of customers visiting each day, the store's financial balance, customer satisfaction rating, daily rent and inventory. The Store class handles day-to-day operations, allowing the player to make strategic decisions about inventory, customer service, and finances.
4. **Supplier**: This class represents the external entity where the store can purchase items to restock inventory. It has attributes like cost list, cost reference list and inventory. The Supplier class works closely with the Store to facilitate transactions, enabling the user to manage their store's stock.
5. **Item**: A class representing any general product sold in the store. It contains attributes shared across all item types like the number of items available, price, brand, and production date. This class is the parent class for more specialized item types, such as perishable items.
6. **PerishableItem**: A class that represents products with a limited shelf life. It adds attributes such as expiration date and shelf life measured in days to track when the item will spoil. This class allow for recalculating the remaining shelf life and checking whether an item has expired, ensuring that perishable goods are managed properly within the store.
7. **Egg**: A perishable item representing eggs. It adds attributes such as the animal the egg comes from and the specific type of egg. It allows for management of different egg products in the store. It adds variety into the available Item that the user can manage, increasing the game's complexity
8. **Milk**: A perishable item representing milk. It adds attributes such as the animal the milk comes from and the specific type of milk. It allows for management of different milk products in the store. It adds variety into the available Item that the user can manage, increasing the game's complexity
9. **Meat**: A perishable item representing meat. It adds attributes such as the animal the meat comes from and the specific body part. It allows for management of different meat products in the store. It adds variety into the available Item that the user can manage, increasing the game's complexity

10. **Toys**: A non-perishable item representing toys. It adds attributes such as the type of toy and the minimum age restriction, ensuring that toys are categorized correctly and to be sold to appropriate customers. It adds variety into the available Item that the user can manage, increasing the game's complexity
11. **Soap**: A non-perishable item representing soaps. It adds an attribute, type of soap, allowing for categorization of different soap products in the store. It adds variety into the available Item that the user can manage, increasing the game's complexity.

Data and Function Members

- **Printable:**
  - Methods
    - **print() : void virtual** -> A pure virtual function that derived classes override to display an easy-to-understand summary of their data.

- **StoreBase:**
  - Attributes
    - **numDifferentItem : int** -> The number of unique items in the store (Not the number of unique item classes used).
    - **inventory : map<string, Item*>** -> A map of string and pointers to `Item` objects representing the stock in the store.
  - Methods
    - **get_numDifferentItem() : int** -> returns the value of **numDifferentItem**
    - **get_inventory() : map<string, Item*>** -> returns the **inventory** vector containing pointers to **Item** objects.
    - **set_numDifferentItem(int) : void** -> sets a value for **numDifferentItem**
    - **set_inventory(map<string, Item*>) : void** -> sets the **inventory** with provided map of string and pointers to **Item** objects.
    - **set_itemNameList(vector<string>) : void** -> sets the **itemNameList** vector with provided vector of strings
    - **change_numDifferentItem(int) : void** -> changes the number of different items in the store when provided with positive or negative integer.
    - **addItem(pair<string, Item*>) : void** -> adds an item to the inventory with given pair of string and **Item** pointer
    - **removeItem(string) : void** -> removes an item from the inventory that matches with provided Item name
    - **print() : void override** -> Override the inherited print() and displays an overview of the store's inventory and item details.

- **Store**
  - Attributes
    - **currentDay : int** -> Tracks the current day in the simulation
    - **numCustomer : int** -> The number of customers visiting the store on a given day
    - **balance : double** -> The store's available balance for purchasing items and paying rent.
    - **rating : double** -> The store's rating, influenced by customer satisfaction
    - **target : double** -> The amount of money the store's should have at the end of the day
  - Methods
    - **get_currentDay() : int** -> returns the current day of the simulation
    - **get_numCustomer() : int** -> returns the number of customers for the day
    - **get_balance() : double** -> returns the store's current **balance**
    - **get_rating() : double** -> returns the store's **rating**
    - **get_target() : double** -> returns the **target** for the day's sales.
    - **set_currentDay(int) : void** -> sets the current day
    - **set_numCustomer(int) : void** -> sets the number of customers

        - **set_balance(double) : void** -> sets the store's **balance**
        - **set_rating(double) : void** -> Sets the store's **rating**
        - **set_target(double) : void** -> Sets the financial **target**
        - **print() : void override** -> Override inherited **print()** and displays a summary of the sales made in the day, item that customer wanted but wasn't in stock, store balance and rating after the day.

- **Suppliant:**
    - Attributes
        - **costRefList : vector<double>** -> A fixed list of costs that is used to ensure that daily product's cost change doesn't fluctuate too much from the reference cost
        - **costList : vector<double>** -> A list of costs for all product in the suppliant inventory that changes daily
    - Methods
        - **get_costRefList() : vector<double>** -> returns the reference cost list for items
        - **get_costList() : vector<double>** -> returns the current cost list from the supplier
        - **set_costRefList(vector<double>) : void** -> Sets the reference cost list.
        - **set_costList(vector<double>) : void** -> Sets the current cost list.
        - **print() : void override** -> displays the supplier's current cost list and available item prices.

- **Item:**
    - Attributes
        - **numItem : int** -> The number of this item in the store's inventory.
        - **price : double** -> The price of the item.
        - **brand : string** -> The item's brand name.
        - **productionDate : string** -> The production date of the item.
    - Methods
        - **get_numItem() : int** -> returns the number of items.
        - **get_price() : double** -> Returns the price of the item.
        - **get_brand() : string** -> Returns the brand of the item.
        - **get_productionDate() : string** -> Returns the production date.
        - **set_numItem(int) : void** -> Sets the number of items.
        - **set_price(double) : void** -> Sets the item's price.
        - **set_brand(string) : void** -> Sets the brand name.
        - **set_productionDate(string) : void** -> Sets the production date.
        - **change_price(double) : void** -> Changes the price of the item
        - **print() : void override** -> Displays the item's details like brand, price, and production date.

- **PerishableItem:**
    - Attributes
        - **shelfLifeInDays : int** -> The number of days the item is good for.
        - **expirationList : string** -> A list with length **shelfLifeInDays** in which each position represents the days left till expiry (0 to **shelfLifeInDays)**, with the value at each position being the amount of PerishableItem with that expiration.
    - Methods
        - **get_expirationList() : int*** -> Returns the expiration date list
        - **get_shelfLifeInDays() : int** -> Returns the shelf life of the item in days
        - **set_expirationList(int*) : void** -> Sets the expiration date list
        - **set_shelfLifeInDays(int) : void** -> Sets the shelf life
        - **recalculate_expirationList() : void** -> Update the expiry status of the PerishableItem based on the current day
        - **print() : void override** -> Displays the item's perishable details like expiration date and shelf life.

- **Egg:**

- Attributes
  - **fromAnimal : string** -> The animal the egg came from
  - **eggType : string** -> The type of egg (e.g., organic, free-range).
- Methods
  - **get_fromAnimal() : string** -> Returns the animal type
  - **get_eggType() : string** -> Returns the egg type
  - **set_fromAnimal(string) : void** -> Sets the animal type
  - **set_eggType(string) : void** -> Sets the egg type
  - **print() : void override** -> Displays egg details such as type and origin

- **Milk:**
  - Attributes
    - **milkSource : string** -> The animal that produces the milk (e.g., cow, goat)
    - **milkType : string** -> The type of milk (e.g., skim, whole).
  - Methods
    - **get_milkSource() : string** -> Returns the source of the milk
    - **get_milkType() : string** -> Returns the type of milk.
    - **set_milkSource(string) : void** -> Sets the milk source.
    - **set_milkType(string) : void** -> Sets the type of milk.
    - **print() : void override** -> Displays the milk's source and type

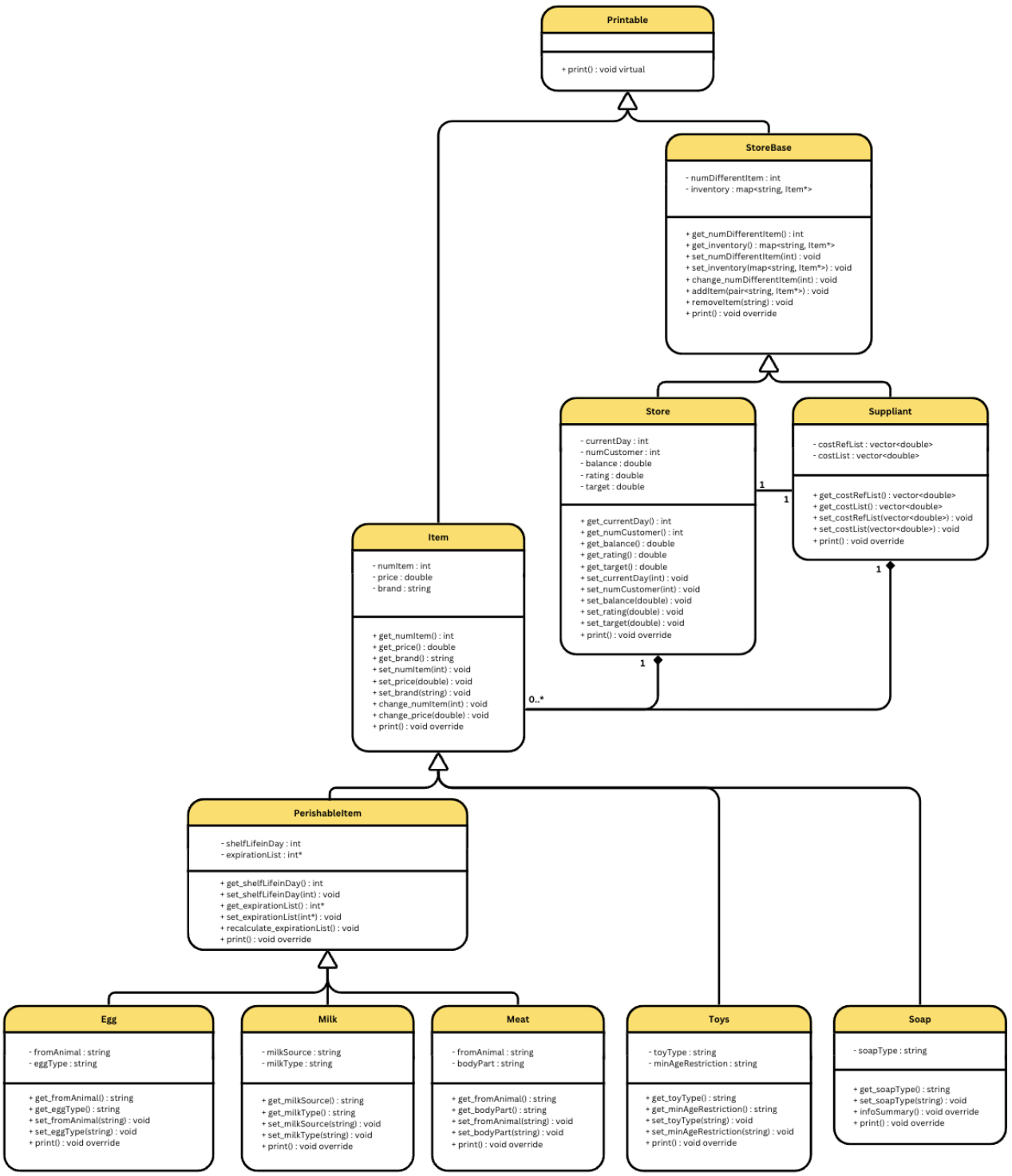
- **Meat:**
  - Attributes
    - **fromAnimal : string** -> The animal the meat came from.
    - **bodyPart : string** -> The body part of the animal (e.g., thigh, breast).
  - Methods
    - **get_fromAnimal() : string** -> Returns the animal type
    - **get_bodyPart() : string** -> Returns the body part
    - **set_fromAnimal(string) : void** -> Sets the animal type
    - **set_bodyPart(string) : void** -> Sets the body part
    - **print() : void override** -> Displays the animal and its type of meat

- **Toys:**
  - Attributes
    - **toyType : string** -> The type of toy (e.g., action figure, puzzle)
    - **minAgeRestriction : int** -> The minimum age restriction for the toy
  - Methods
    - **get_toyType() : string** -> Returns the toy type
    - **get_minAgeRestriction() : int** -> Returns the age restriction.
    - **set_toyType(string) : void** -> Sets the toy type
    - **set_minAgeRestriction(int) : void** -> Sets the age restriction
    - **print() : void override** -> Displays the toy type and its minimum age restriction for purchase and usage.

- **Soap:**
  - Attributes
    - **soapType: string** -> The type of soap (e.g., shampoo, conditioner, body wash, block)
  - Methods
    - **get_soapType() : string** -> Returns the **soapType**

- **set_soapType(string) : void** -> Sets the **soaptype** with provided input
- **print() : void override** -> Displays the soapType

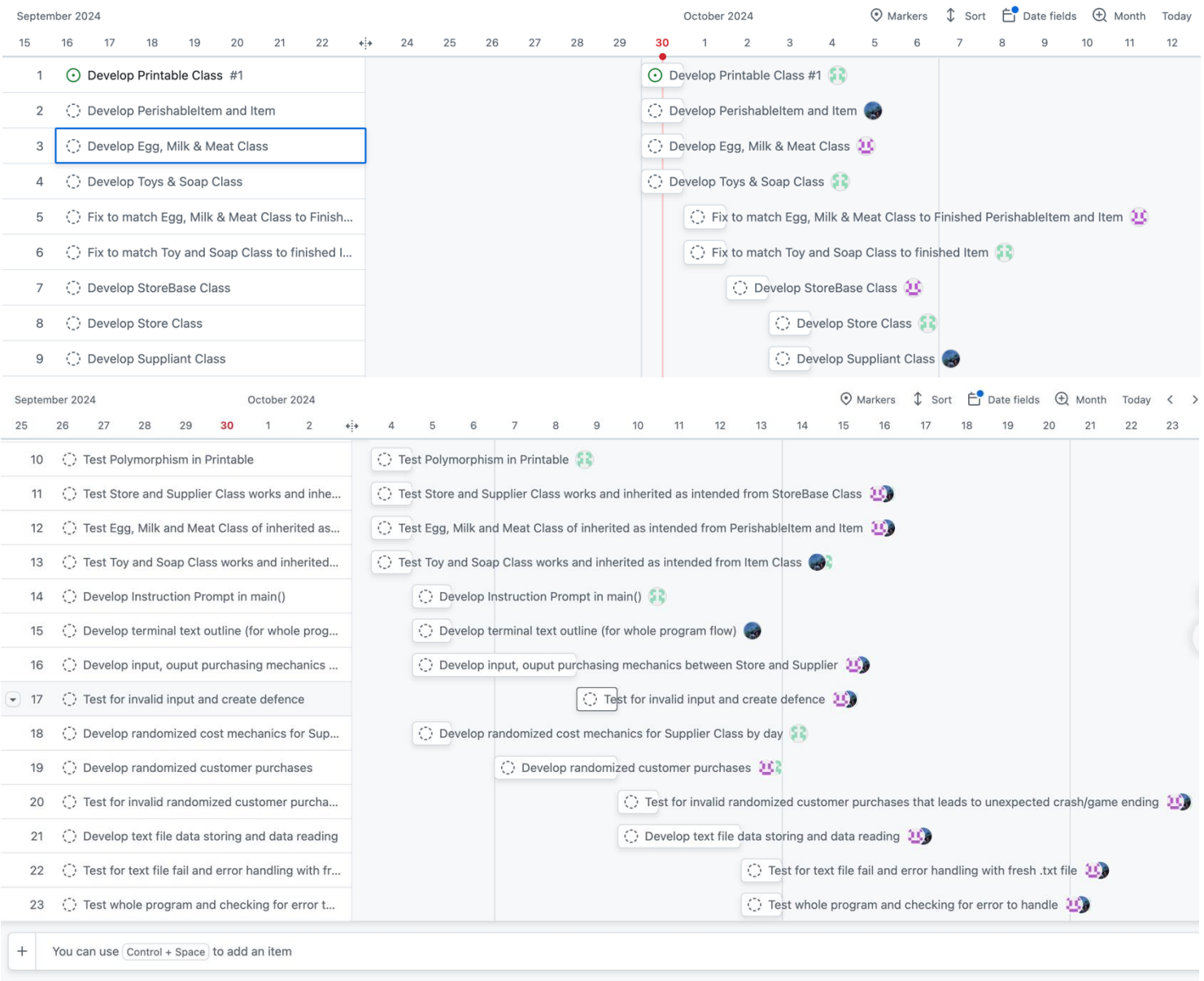
Relationships between Classes



- There's a distinct and purposeful 3 inheritance between Item, PerishableItem and Egg, Milk & Meat. Item is the base class with attributes that all Item, in general, would have including numItem, price and brand. PerishableItem is a type of item that has these attributes in addition to information regarding shelf life and expiration date. Egg, Milk & Meat is a type of PerishableItem with their own unique information such as bodyParts and milkSource.
- Item cannot exist without Store while a Store can exist without Items. With the scope of the project where the user only manages one Store, a Store can have multiple Items but an Item can only be in one Store. The same can be said with the composition relationship between Item and Supplier.

- With the scope of the project being the user managing one store in interacting with a supplier that always has stock to sell, there's an association between Store and Supplier in which a Store can only have a Supplier and vice versa.

Project Task List and Timeline



1. Develop Printable Class (**Hung Hang Yep**)
2. Develop PerishableItem and item (**Thai**)
3. Develop Egg, Milk, & Meat Class (**Steve**)
4. Develop Toys & Soap Class (**Hung Hang Yep**)
5. Fix to match Egg, Milk & Meat Class to Finished PerishableItem and Item (**Steve**)
6. Fix to match Toy and Soap Class to finish Item Class (**Hung Hang Yep**)
7. Develop StoreBase Class (**Steve**)
8. Develop Store Class (**Hung Hang Yep**)
9. Develop Suppliant Class (**Thai**)
10. Test Polymorphism in Printable (**Hung Hang Yep**)
11. Test Store and Supplier Class works and inherited as intended from StoreBase Class (**Hung Hang Yep, Thai & Steve**)
12. Test Egg, Milk and Meat Class of inherited as intended from PerihableItem and Item Class (**Steve & Thai**)
13. Test Toy and Soap Class works and inherited as intended from Item Class (**Thai & Hung Hang Yep**)

14. Develop Instruction Prompt in main() (**Hung Hang Yep**)
15. Develop terminal text outline (for whole program flow) (**Thai**)
16. Develop input, output purchasing mechanics between Store and Supplier Class (**Steve & Thai**)
17. Test for invalid input and create defense (**Hung Hang Yep, Thai & Steve**)
18. Develop randomized cost mechanics for Supplier Class by day (**Hung Hang Yep**)
19. Develop randomized customer purchases (**Steve & Hung Hang Yep**)
20. Test for invalid randomized customer purchases that leads to unexpected crash/game ending (**Hung Hang Yep, Thai & Steven**)
21. Develop text file data storing and data reading (**Thai & Steve**)
22. Test For text file fail and error handing with fresh.txt file (**Thai & Steve**)
23. Test whole program and checking for error to handle (**Thai & Steve**)

User Interaction Description

The user will entirely interact with the program through a text-based terminal, where concise and reasonable spacing, for readability, instructions and information will be displayed on the game mechanics and logic. The program will display and allow the user to see the simulated day information, store status along with information on available stocks from supplier to make their purchase. User will be prompted to provide numerical/text inputs to purchase items with updates on their balance after every purchase made. If an invalid purchase is made, either due to lack of money or invalid item, a message will be displayed in the terminal to explain the issue and request the user to input with provided options. The program will display the automatic sales made and a summary of the store's status after the day, showing whether the user's store is still operatable.

Unit Testing and Debugging Plan

The program will be tested using unit testing and a wide range of different input to check for edge cases for debugging to ensure that the program runs and outputs as expected.

Unit Test

There will be unit tests for every method to ensure they work as intended. For example, we would run getters methods and check if the returned datatype and data is as its default value, if not, we will output an error message through error handling. In another instance, we would check if the setters methods changed class's variable with input value, if not, we will output an error message. Taking the addItem(Item*) function as an example, we will check if an Item* is added to the inventory when inputting an Item*, if not, we will output an error message.

Input/Output Test

For input/output testing, we will simulate errors related to floating-point precision, infinite loops, and incorrect memory allocation by testing the program with unexpected input data files and edge case scenarios. This will help us identify and handle errors effectively during runtime. An example of this would be unexpected file format or unexpected datatype such as inputting a string into set_shelfLifeInDay(int shelfLifeInDay) which expects an int.

After finding errors through testing, we will handle these invalid inputs to ensure the program doesn't crash and runs as intended (criteria for success). For example, we would debug the issue of infinite loop by incorporating a looping condition that only maintains the loop until a certain condition is met. We would know that this program

has been debugged when the loop stops when intended too during testing. All functions will be tested separately and tested once more when they're combined in the main file.

Integration Test

During development, we'll do integration testing to ensure that new codes run as intended with the previous working program. For example, we would test if Store class (child class) correctly inherit from StoreBase class (parent class).

Makefile will be included to guide the user on how to compile the project and a README file will be included to describe how to interact with the program and explain how data is stored in the text file (record game status).