Introduction

Our inventory management application serves as a tool for automating inventory checking through the use of the observer pattern. It allows a store (an observer) to subscribe to a certain warehouse (subject) in order to get the most updated information regarding inventory items in that warehouse. The idea is to allow stores to be notified of the status of items in warehouses in terms of shortage and excess. The two main actors of our inventory management system are the store and the warehouse. In order to demonstrate the core functionalities of our application, we have written the main.cpp from the perspective of an administrator who has access and can act as both the warehouse and the store. Our application supports the following main functionalities:

- A store keeps track of items in a warehouse by subscribing
- A store is able to order items from different warehouses
- The warehouses could have different types of items
- Different types of items could have different behaviors

Architecture and Technologies

The architecture of our application is based upon the three chosen design patterns: strategy, observer, and factory method. Our choice of an inventory system application was due to our desire to implement and clearly showcase the benefits of utilizing design patterns. The store (observer) to warehouse (subject) idea was a clear way to showcase how observerers reacted to changes in a subject. The factory and strategy design patterns were able to be showcased through the various items and their features created to be stored in our warehouses.

Regarding the actual application, it is based off of the C++11 language since it is a language all four of our group members are familiar with. The user interface of the application is command-line based. All classes are designed while following several coding design principles detailed in the section below. *Figure 1* is our UML class diagram for the entire application.

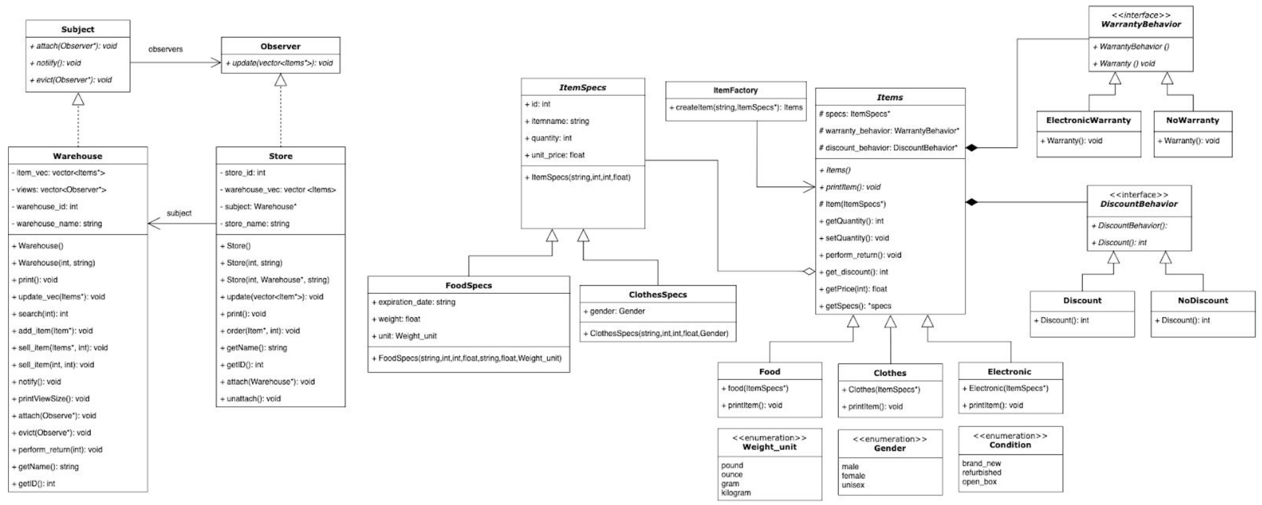

Figure 1. UML Class Diagram.

Design Principles

When designing our application the following design principles were followed: loose coupling, information hiding, encapsulate what varies, and favor composition over inheritance. The principle of loose coupling was applied throughout the program to limit interconnects between our store, warehouse, and item classes. This allows us to make desired changes without worrying about unintended consequences. The information hiding and encapsulation principles were both applied to our class designs and is most apparent in our implementation of an item factory class. Classes only contain the necessary public member variables and functions, while most information is kept as private or protected members so that any child of a particular class can access. Our code design favors composition over inheritance as seen through our implementation of the item behaviors. For instance, different items have varying warranties which are implemented through interfaces rather than straight inheritance.

Design Patterns

Strategy

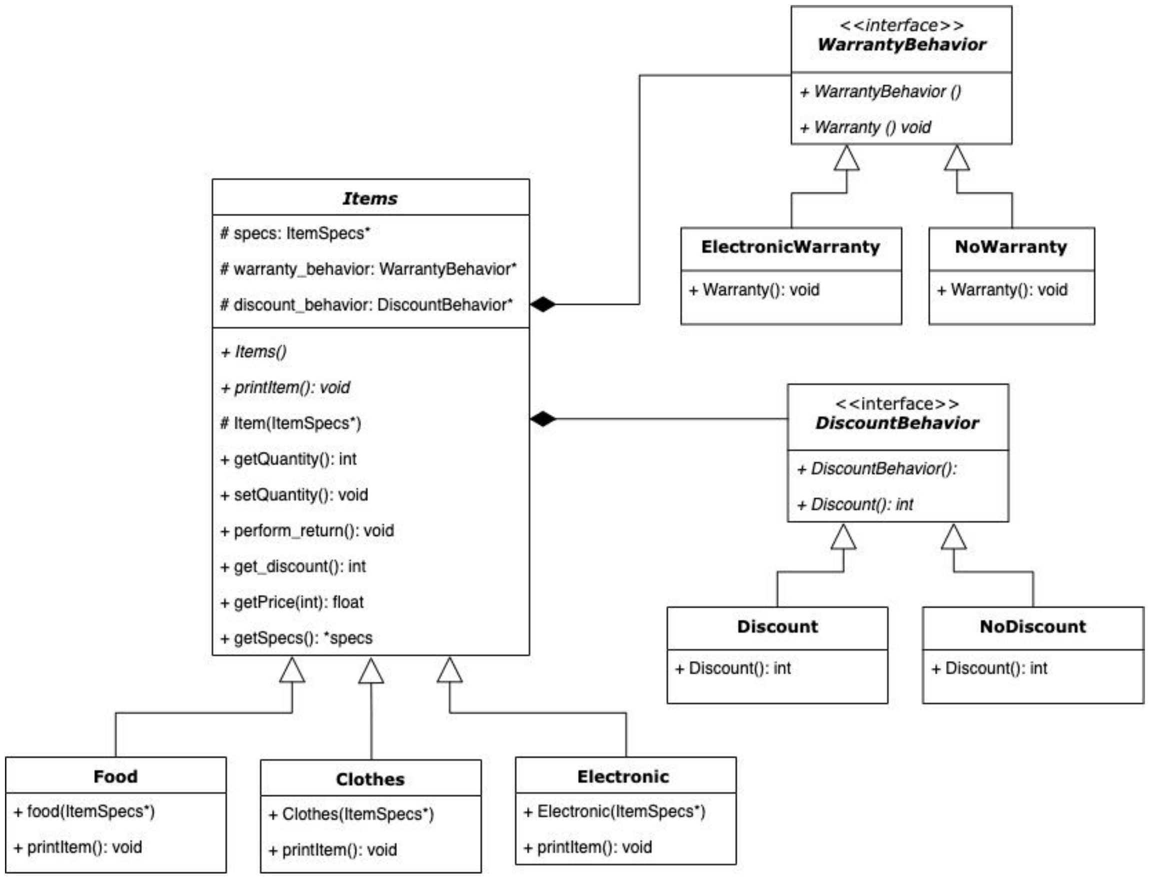

Figure 2. Strategy Pattern.

Our “Items” class is the base class for any type of item. Based on the type of items, they could have different behaviors. For example, it makes sense for electronics or clothes to have behavior on warranty or return policy but the same thing doesn’t apply to food. Instead of using virtual functions and abstract classes, we decided to use the strategy design pattern.

With the strategy design pattern, you encapsulate one or more types of behaviors into a base behavior class and put the actual implementation of the behaviors in its derived classes. For example, we created a warranty behavior class which has one child class for the electronic warranty and one without any warranty or doesn’t allow a return. It is more for a demo purpose, so they only print out different strings.

The benefit of using strategy design pattern over virtual functions is that when you have many different derived classes that share some behaviors and some don't, you only have to implement each type of behavior once. With virtual functions, we would have to implement that for each derived class which would have a terrible performance when scaling up and our codebase would have duplicate code. The benefits of the strategy design pattern in our application might not be very obvious due to the small scale of the application. But if we ever need to increase the scale, this design pattern would make the future implementation easier with less repeated code. For example, if we were to add a new behavior such as a discount behavior for the items, we can just add a new interface class called “DiscountBehavior” to our UML class diagram in *Figure 2.*

Observer

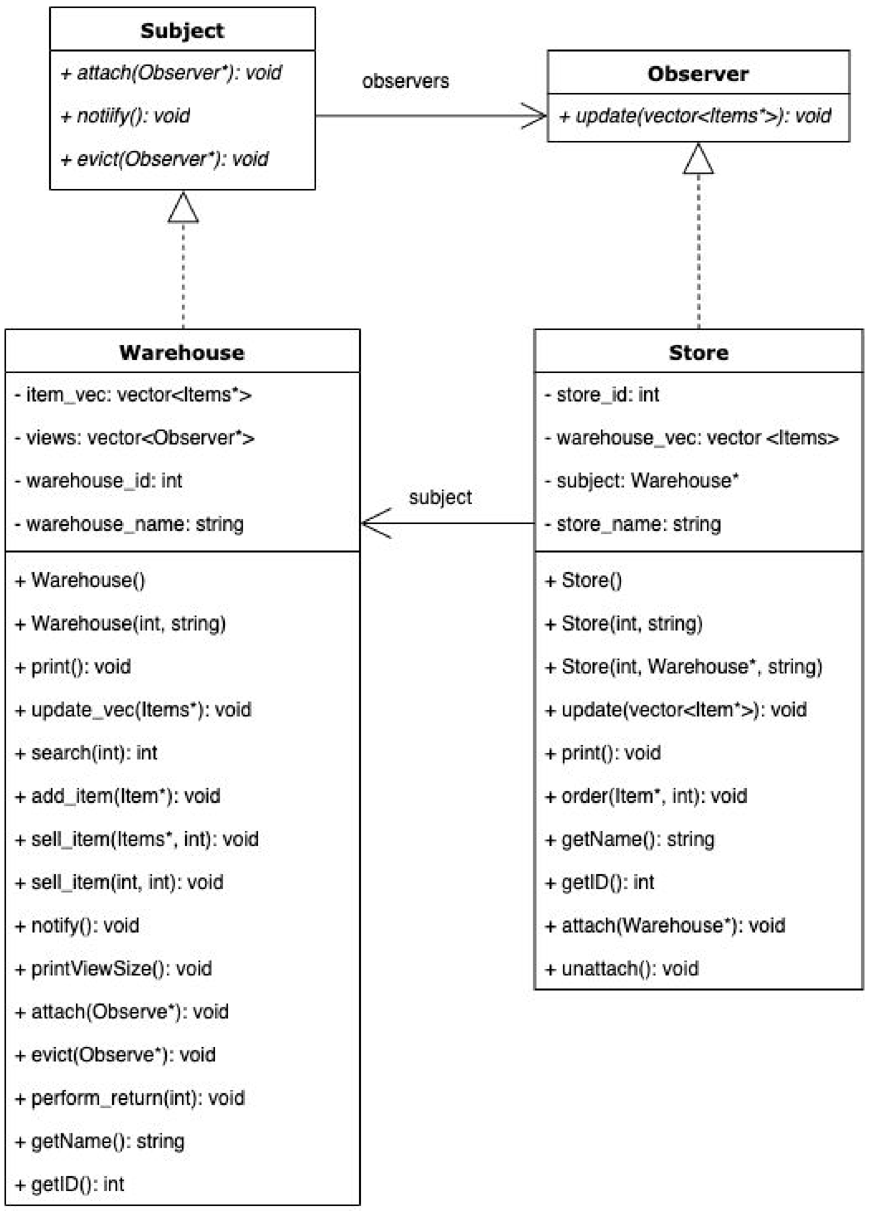

Figure 3. Observer Pattern.

In our application, the “Warehouse” class keeps a list (vector of item pointers) for items inside the warehouse. The “Store” class would need to access this information when ordering. We decided to use the observer design pattern here to make each store observe the list of items in a particular warehouse. Whenever that list is updated by the warehouse, it would notify any store that is observing the list. One warehouse could be observed by any number of stores, and a store could only observe one warehouse in our current implementation. Due to the limit of time, we were not able to convert this one-to-many observer relationship into many-to-many. But we believe that it is possible. The benefit of this design pattern would be more obvious under a many-to-many model. Imagine if a store could order from 100 warehouses, it would need to request the 100 item lists from each warehouse whenever it needs the information. That could hurt the performance even more when the number scales up to 1000 or even more. With the observer design pattern, warehouses would notify all stores observing it whenever there is an update on its item list which ideally would be when the system is idle so that the performance would be so much better when stores need the item lists from all warehouses it is observing.

Factory Method

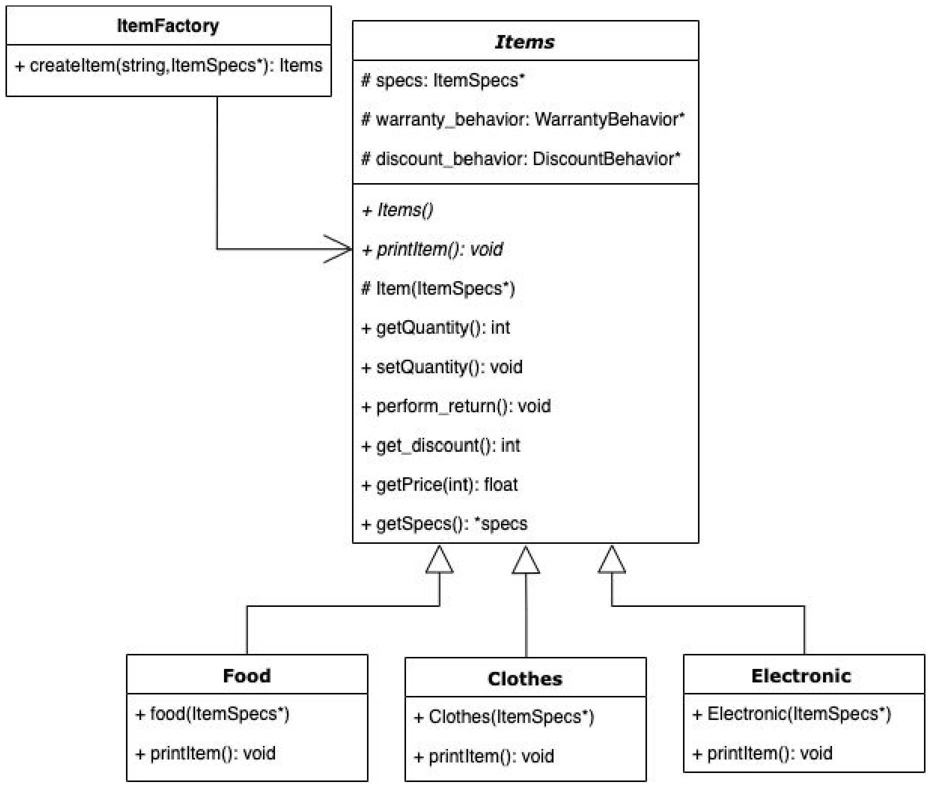

Figure 4. Factory Method Pattern.

As mentioned before, we have an "Items" base class and several derived classes for different types of items. We decided to apply the factory method design pattern to the "Items" class to handle instance generation. We created an item factory class which would take in an item spec and a string indicating what type of item it should be and the factory will return a new instance of item of that type containing the item spec.

The benefit of using the factory method design pattern instead of simply expanding the constructor of the base class is that whenever we need to add a new item type in the future, we won't need to modify the base class. The implementation of an application usually involves more than one developer. Changing the implementation of a base class might need update in several developers' code. With the factory class, this would not happen because the instance generation is encapsulated inside the factory class and we only need to update that class.

Conclusion

In terms of meeting our own expectations of the application, we were overall satisfied with the result. Upon brainstorming for the project, we realized that our initial plan was too ambitious (which we later reduced the application coverage to a few key functionalities) and we ran into the “paralysis of analysis” problem several times. Therefore, we were aiming for a Minimum Viable Product (MVP) that is able to showcase the three design patterns and our application of the design principles. As for demonstrating the application, we mainly focused on operations like attaching (subscribing) a store to a warehouse, adding new items, notifying the stores that are subscribed to a warehouse, unattaching (unsubscribing) a store, and perform return to show the warranty behavior. In terms of the operation of placing an order from a warehouse to a store, we simply created methods to check whether an item is in stock and decrement the quantity without dynamically creating a storage unit to show the transfer of those purchased items. Our design by no means is the best way of implementing an inventory management system; however, we believe that it is optimal and it is something that could be expanded and improved upon.