SYSC2004 – Project Report

Milestone 2

Sebastien Marleau
SN: 101155551

Change log:

- Created ProductQuantityPair
    - A basic helper class to relate a Product instance to its related quantity integer
    - used in ProductQuantityCollection

- Created a ProductQuantityCollection class
    - Is inherited by both Inventory and ShoppingCart
    - It's a collection of products and their related quantities
    - Transferred most of the code from M1's Inventory class
    - Uses a HashMap with product ID to ProductQuantityPair instead of the double HashMap
    - Most methods interface with productIDs instead of Product references, a convention kept throughout the whole program
        - addProductToCollection being the only exception, which takes a Product as parameter to add it to the collection
    - Throws IllegalArgumentExceptions when a productID is given that doesn't correspond to any products in the collection
    - Has a getAllProductIds() function that returns a Stream<String>
        - I chose Stream<String> as almost any collection can be reduced to a stream. This is much better that using Set<String>, as if I change the HashMap to something else in the future, I would've had to convert that something else into a Set, or change the return type of this function and change the code in all use cases of it. A good example of abstraction.
    - Chose to not have a getter method for ProductQuantityPairs. Instead there are two, one for the Product and one for the quantity. This is again abstraction so that in the future if the implementation details change and ProductQuantityPair is removed, the classes that use ProductQuantityCollection don't have to be changed.

- Inventory
    - A type of ProductQuantityCollection where product quantities cannot go below 0 (throws an IllegalArgumetnException)
    - Is initialized with some products in the collection by default
- ShoppingCart
    - A type of ProductQuantityCollection where products whose quantities reach 0 or below get removed from the collection.
    - Each instance has a unique ID set by the class on object creation
    - Also has functions to compute cart price and item count.

- StoreManager
    - Now manages both an Inventory and a collection of shopping carts
    - cartIDs are used to communicate with it instead of ShoppingCart references. This adds a layer of abstraction
    - When products are added removed form the carts, the quantities are taken from or added back to the inventory
    - Can display representations of the cart and the inventory to System.out

- StoreView
    - Is assigned a StoreManager instance to display the view of. Remembers the cartID assigned to it by the StoreManager
    - Added the displayGUI() method to start the store's GUI.

Questions

1. **What kind of relationship is the StoreView and StoreManager relationship? Explain.**

It is a one way relationship fom StoreView to StoreManage, where StoreView depends on StoreManager. StoreManager can exist fine on its own, but StoreView needs a StoreManager to display a store.

2. **Due to their behavioral similarity, would it make sense to have ShoppingCart extend Inventory, or the other way around? Why or why not?**

It wouldn't make sense, as inheritance represents a "is-a" relationship. You can't say that a ShoppingCart "is-a" Inventory or the other way around. This inheritance would probably cause problems in the long run, where and when Inventory and ShoppingCart may not be so similar.

3. **What are some reasons you can think of for why composition might be preferable over inheritance?These do not have to be Java-specific.**

Inheritance often creates tightly-coupled classes. This is a problem when you may want to modify a base class, but know that it will be a lot of work since you also have to change all the children classes. Composition on the other hand creates a layer of abstraction, where all the clients of the class see are the public methods and attributes. This makes the "coupling" much lighter. Especially with things like interfaces, composition can often do all that inheritance can do, but with less coupling.

4. **What are some reasons you can think of for why inheritance might be preferable over composition? These do not have to be Java-specific.**

Inheritance can be a lot easier to implement than composition, thus requiring less work. For example with the base class of ProductQuantityCollection, it was very easy to implement two different types of this class, in Inventory and ShoppingCart, which both had slight functional changes. All I had to do was extend and override key methods. Composition would have required an extra attribute for the collection, and also the re-implemention of basic methods to add and remove products.
It is also often easier to think in objects, as it is relationships we are familiar with.