Participating Team Members:
Taylor West
Dat Mai
Nathan Zheng
Jay Patel
Saddiq Rupani

Explanation for merging of GiftCardItem:
Dispensable (Lazy Class): Item is a superclass of GiftCardItem. However, these two classes have no real need to be a superclass and subclass of one another since they are so similar to each other. GiftCardItem has no unique attributes to it and therefore can have its hierarchy be collapsed. We can then just simply merge the Item and GiftCardItem into just the Item class by itself. This then requires some tweaking of the hasGiftCard method in the Order class and the constructor in the main method.

Explanation for extracting classes from calculateTotalPrice() method:
Bloater (Long Method): The calculateTotalPrice method was refactored to address the "Long Method" code smell by breaking it down into smaller methods. First, the discount logic was extracted into an applyDiscount method, which calculates any discount based on the item's discount type. This makes the code more readable by isolating the discount-related logic. Similarly, tax calculation was moved into a calculateTax method, handling tax exclusively for items that are taxable. Each item's subtotal, including discounts and tax, is now calculated in a calculateItemTotal method, which centralizes the item-level computations and allows for an organized approach to processing each item. Finally, the gift card and bulk discount adjustments were moved to an applyFinalAdjustments method, keeping these specific modifications in one place.These changes allow the code to be more readable and make it simpler to implement future changes.

Explanation for new Discount Class:
Bloater (Data Clumps): The item and taxableItem classes were using the parameters discountType and discountAmount. These data items don't make sense without each other, and they're increasing the size of the constructor method. Therefore, it makes more sense to extract these two values. To do this, I created a new class, Discount, with type and amount as parameters. Instead of discountType and discountAmount, item and taxableItem take in a discount object that can then be used to access the type and amount. The way the methods for calculating the total price calculate the discounts must be changed to reflect the new class, as well as the constructors for item and taxableItem in main.

Explanation for moving DiscountType Class into Discount Class:
Couplers (Feature Envy): The DiscountType Class and Discount Class are too closely related to each other. Discount Class creates an instance of DiscountType Class and has methods directly related to getting values from DiscountType Class. Therefore, I merged the class in DiscountType Class directly into Discount Class, effectively getting rid of the DiscountType Class. This means all of the types required by the Discount Class will be inside of the Discount Class.

Explanation for removing switch statement from calculateTotalPrice():
OO Abuser (Switch Statement): Previously, calculating the price in the order class required a switch statement which checked the discount type of the item's discount. Switch statements are not a good use of Object Oriented Programming, and they make it more difficult for the code to be changed later. To fix this, I added an abstract class to the DiscountType enum. I then used override to give PERCENTAGE and AMOUNT their own implementation of the applyDiscount method. Then I removed the switch statement and replaced it with a call to the applyDiscount method through the item's discount's type, so that the correct version of the method would be implemented. I also removed the default switch call, since items are always created with discounts (which have a type and amount), even if the discount is 0.