One code smell found in the codebase was the dispensable class of GiftCardItem. It was dispensable because it counts as a lazy class, as it doesn’t do much other than have a constructor, so its functionality could easily be implemented into the Items class. In order to refactor the code to remove this class entirely while maintaining functionality, a boolean to check if an item is a giftcard or not was added to the Item class constructor, named isGiftCard. I had to add this boolean to the Items and TaxableItems class constructors. To keep the order calculations correct, the hasGiftCard method in the Order class had to be changed to check through the list of items for the boolean isGiftCard rather than if the item is of type GiftCardItem. Implementing all these changes and ensuring the item3 item in Main is set to be an Item object rather than GiftCarditem, the same output and functionality of the code is maintained.

Another code smell can be found in the Order class. The calculateTotalPrice method is exceptionally long and thus can be considered a Bloater, specifically a long method. To fix this, we can extract some of the operations and put them as standalone methods. For example, we will calculate the discount, tax, apply gift card, and apply bulk discount in another method. This way, the code is more readable and testable.

Another code smell could be found between both the DiscountType enum and the Order class, which used a switch statement to choose the discount behavior from a key. The key in question was the DiscountType enum, which was not extensible to future discount types due to being an enum. This was a classic case of Object Oriented Programming abuse, so we fixed it by changing DiscountType to an interface and implementing the strategy pattern. Each type of discount now has its own class that overrides the new applyDiscount method, which takes in an initial price and a discount amount and returns the discounted price. The Order class’s calculateDiscount method has been replaced with applyDiscount, which calls the applyDiscount method on the passed in Item’s discountType instance and returns its value or returns the item’s base price if its discountType is null.

Another code smell is found in the order class with its method applyDiscount. This method checks the discountType and discountAmount attributes of the Item class, so it makes more sense to move this method inside of the item class itself. This reduces coupling, makes Item not a lazy class, and reduces the size of the order class.

An additional code smell was found in the Item.java class. Here, it speculates that the price, quantity, and other variables associated with it can be changed. However, if these variables were to actually be changed, that would be a whole different Item, so you would have to create a new object. Therefore, these variables should be made final and immutable.