Online Shopping Using Object Oriented Design Methods

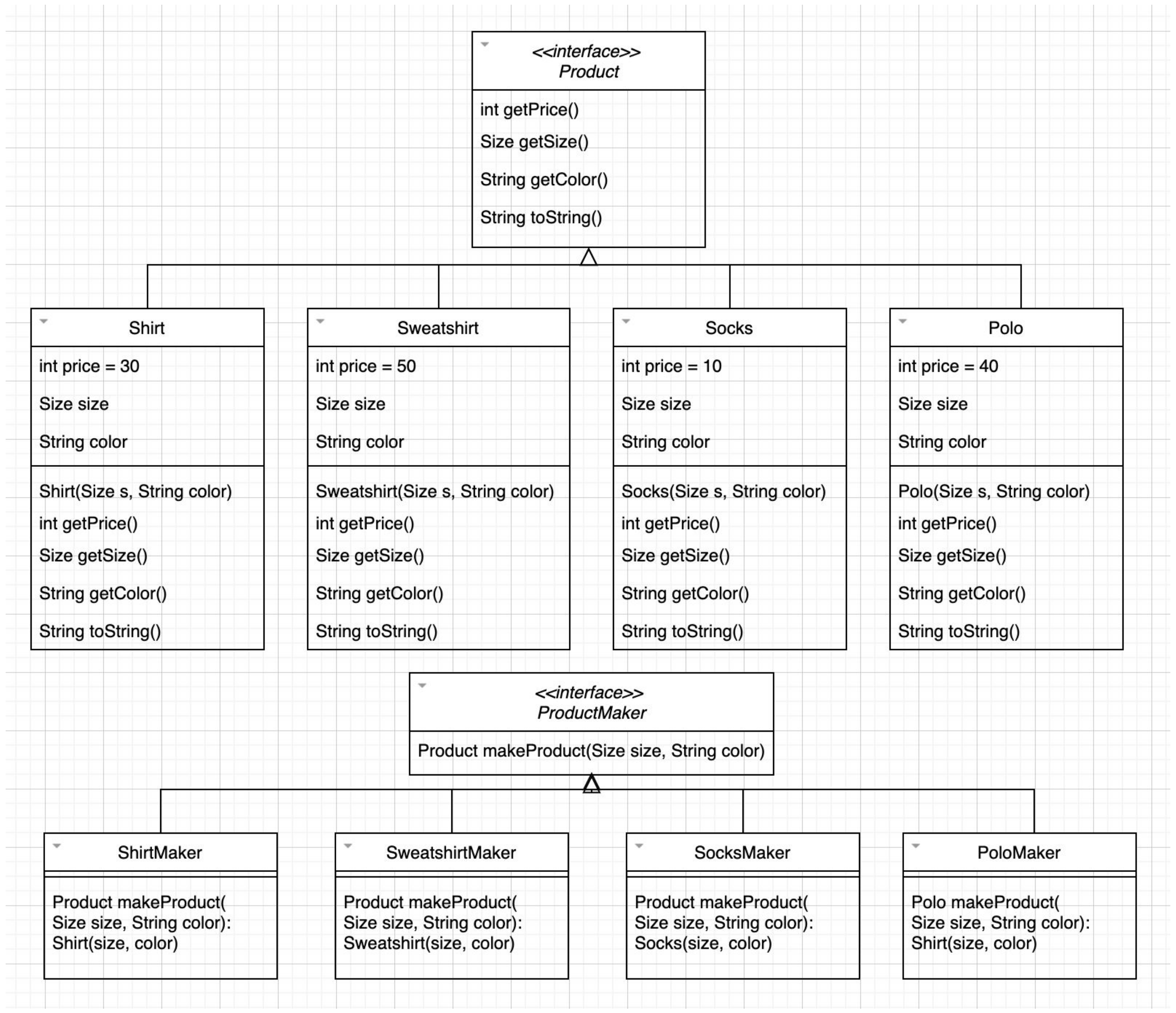
Joseph Porpora

My project is modeled after an online shopping experience. The use of products and maroon and gold coloring is to echo the experience one might have at the Boston College bookstore's online website.

To model the products in my bookstore, the creational pattern I used was the factory method. The factory method seemed ideal here to separate the object creation from the object methods. As the products do not need to be built in steps and do not necessarily have a high "cost" to build each time, the factory method seemed like the ideal creation method. The file Product contains the Product interface, as well as the products that implement. In my pseudo-BC bookstore, there are four products on sale: shirts, sweatshirts, socks, and polos. Each product has three attributes: price, size, and color. The getters for these values are all inherited from the product interface. Size is an enumerated type, located in the file Size. The supported sizes are S, M, L, and XL. The color of the product is assigned at creation. In my attached examples I only use "Gold" and "Maroon" for simplicity, though more could easily be added based on the "store owner's" wishes. Each product has a price specific to its subclass; shirts are 30, sweatshirts are 50, socks are 10, and polos are 40.

The ProductMaker interface and the classes that extend it are located in the file ProductFactory. ProductFactory has one method called makeProduct, which has inputs of Size and a string denoting the color and returns a Product. The four subclasses are ShirtMaker, SweatshirtMaker, SocksMaker, and PoloMaker. They all implement makeProduct, and the four subclasses return a Shirt, Sweatshirt, Socks, and Polo respectively. The following UML diagram represents the related nature of the products and the product factories.

Products and Product Factories UML Diagram

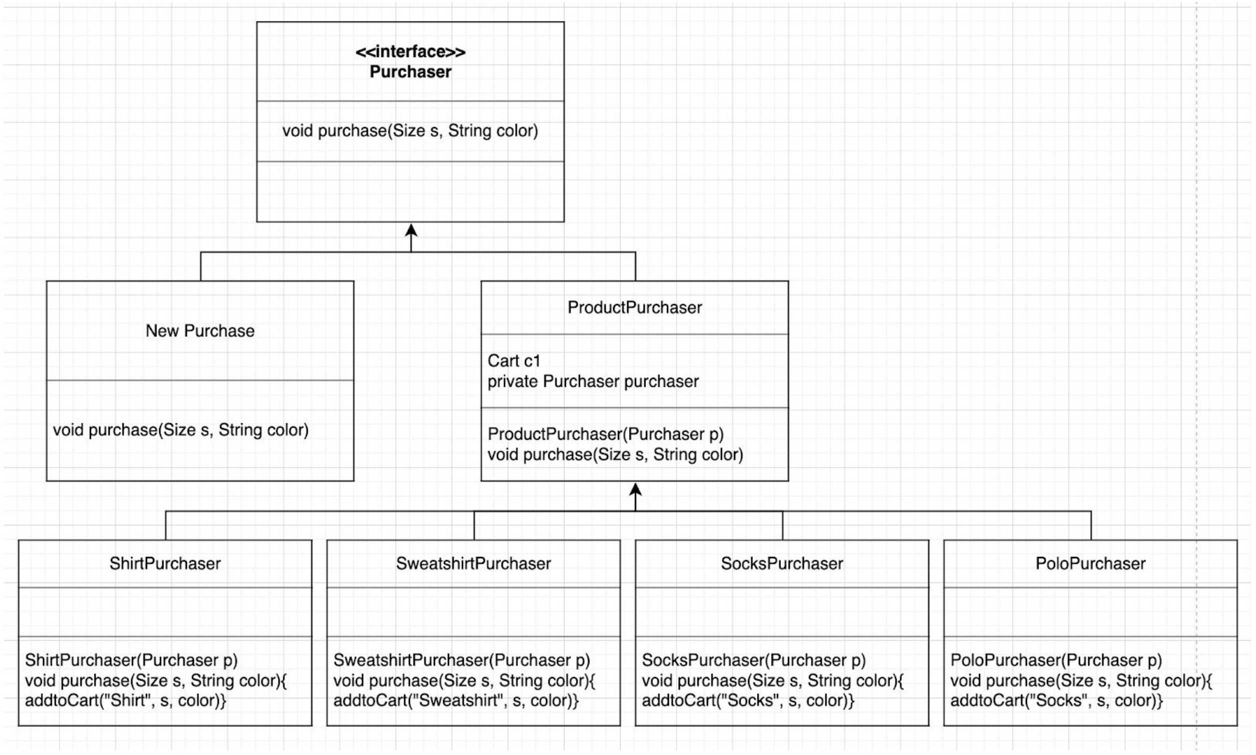


Though not necessarily implemented using object oriented programming patterns, the files Stock and Cart contain code which further embellishes the online shopping process. Instead of the purchase of an item leading to its creation for the customer's carts, items are created and added to stock to mirror real-world practices. There are four arraylists of products which denote the stock of each type of product available at the store. The method addtoStock takes in what product to produce, what size, and what color, and calls on the respective Product

Factory to create the object. This new product is then added to its respective stock arraylist. There is also a definition of addtoStock which takes in an int as well to produce multiple of the same object. The method getStock returns the stock so we can check what items are in stock. The Cart file includes other methods related to the shopping cart. getPrice will return the carts price by iterating through the items ordered. addtoCart will remove an item from stock and add it to cart or give a message saying the specified item is not in stock. getOrder and showOrder are used mainly for testing purposes to view the items in the cart.

The decorator pattern was used to model the process of adding items to the shopping cart, and is contained in the file CartDecortator. I thought cart decorator would be the best for an online shopping interface as the decorator is used when you want different combinations of objects. As there could be infinite combinations of items in a cart (as long as the stock is sufficient), I thought decorator would be best suited to this. Additionally, as there is no functional difference between a cart with a shirt added first then a sweatshirt and a cart with a sweatshirt added first then a shirt, the decorator seemed ideal. The interface Purchaser has two subclasses which implement it: NewPurchase and ProductPurchaser. NewPurchase simply prints out that a new order has begun, while ProductPurchaser creates a new Cart by using the blank constructor from Cart. The classes ShirtPurchaser, SweatshirtPurchaser, SocksPurchaser, and PoloPurchaser all implement ProductPurchaser. Each subclass calls purchase and takes the input size and color and passes it into addtoCart, as well as the type product to purchase. The following UML diagram represents the related nature of the interfaces and classes in CartDecorator.

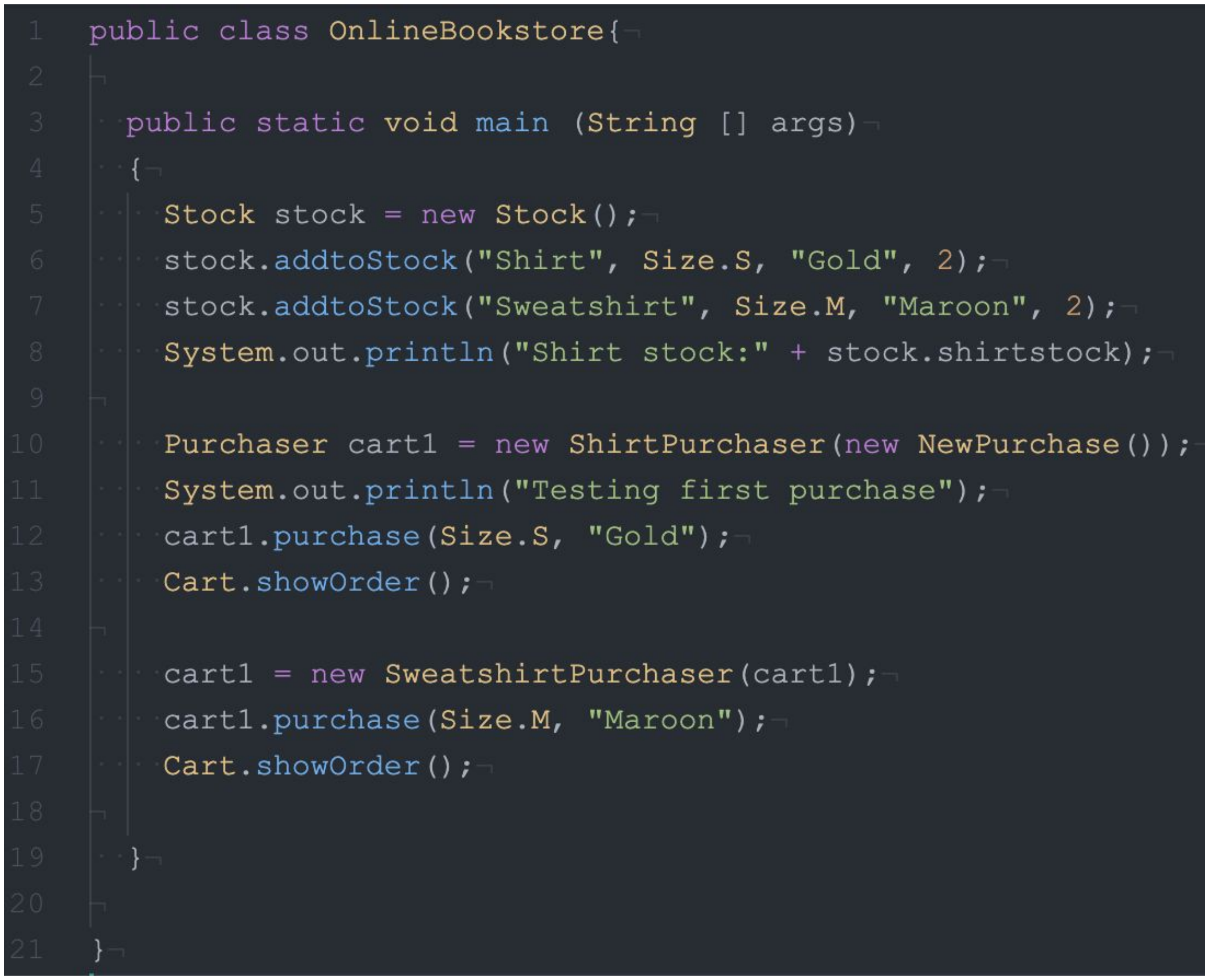
Cart Decorator UML Diagram



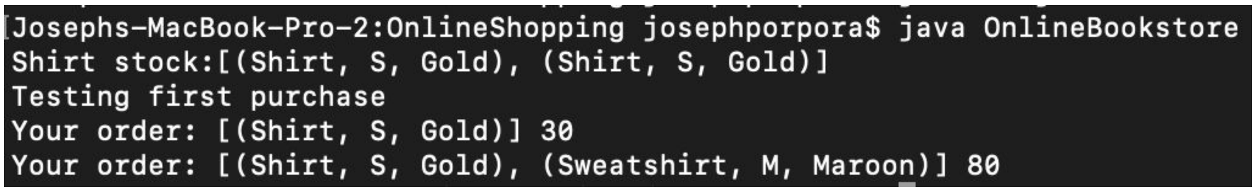
<<interface>>
Purchaser
void purchase(Size s, String color)
New Purchase
void purchase(Size s, String color)
ProductPurchaser
Cart c1
private Purchaser purchaser
ProductPurchaser(Purchaser p)
void purchase(Size s, String color)
ShirtPurchaser
ShirtPurchaser(Purchaser p)
void purchase(Size s, String color){
addtoCart("Shirt", s, color)}
SweatshirtPurchaser
SweatshirtPurchaser(Purchaser p)
void purchase(Size s, String color){
addtoCart("Sweatshirt", s, color)}
SocksPurchaser
SocksPurchaser(Purchaser p)
void purchase(Size s, String color){
addtoCart("Socks", s, color)}
PoloPurchaser
PoloPurchaser(Purchaser p)
void purchase(Size s, String color){
addtoCart("Socks", s, color)}

The following images give examples of running the code through the driver program OnlineBookstore. The first few lines of each example can be imagined as actions of the store owner, while the rest can be imagined as the actions of the customer. First, various items are added to stock, showing what the store owner does behind the scenes. Then, the customer creates their order.

Example 1:



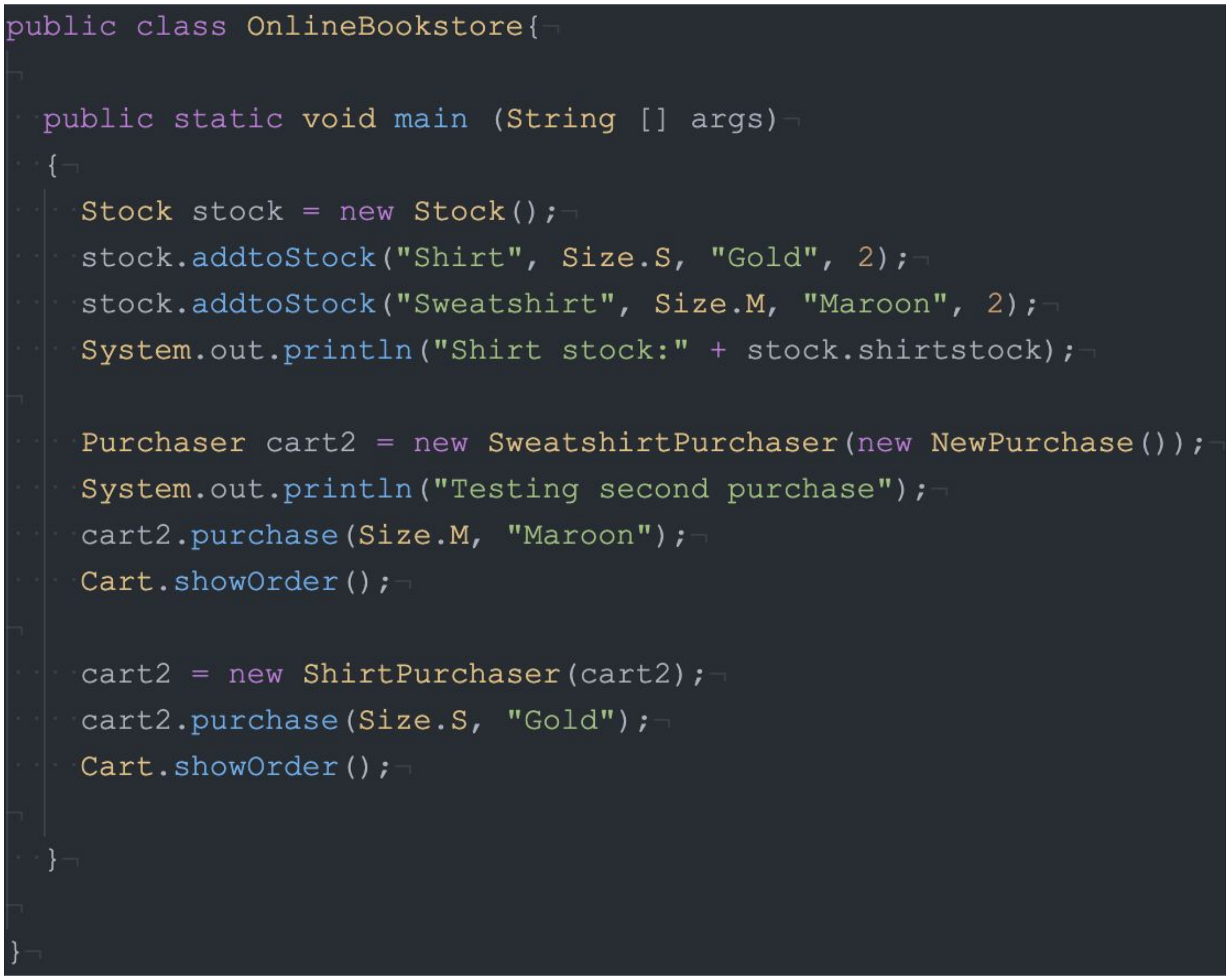
```java
public class OnlineBookstore{

  public static void main (String [] args)
  {
    Stock stock = new Stock();
    stock.addtoStock("Shirt", Size.S, "Gold", 2);
    stock.addtoStock("Sweatshirt", Size.M, "Maroon", 2);
    System.out.println("Shirt stock:" + stock.shirtstock);

    Purchaser cart1 = new ShirtPurchaser(new NewPurchase());
    System.out.println("Testing first purchase");
    cart1.purchase(Size.S, "Gold");
    Cart.showOrder();

    cart1 = new SweatshirtPurchaser(cart1);
    cart1.purchase(Size.M, "Maroon");
    Cart.showOrder();

  }

}
```



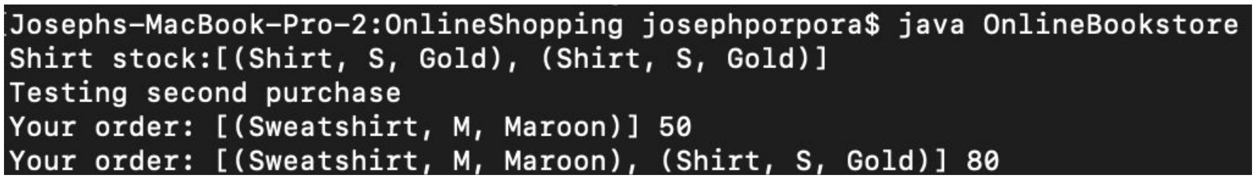
```
[Josephs-MacBook-Pro-2:OnlineShopping josephporpora$ java OnlineBookstore
Shirt stock:[(Shirt, S, Gold), (Shirt, S, Gold)]
Testing first purchase
Your order: [(Shirt, S, Gold)] 30
Your order: [(Shirt, S, Gold), (Sweatshirt, M, Maroon)] 80
```

In this example, 2 small gold shirts are added to stock, as well as 2 medium maroon sweatshirts. Cart1 is created as a new shirtpurchaser, and a shirt is purchased. Cart1 is then cast as a sweatshirt purchaser and then a sweatshirt is purchased. The cart contains these two items, and has the combined total of 80.

Example 2:



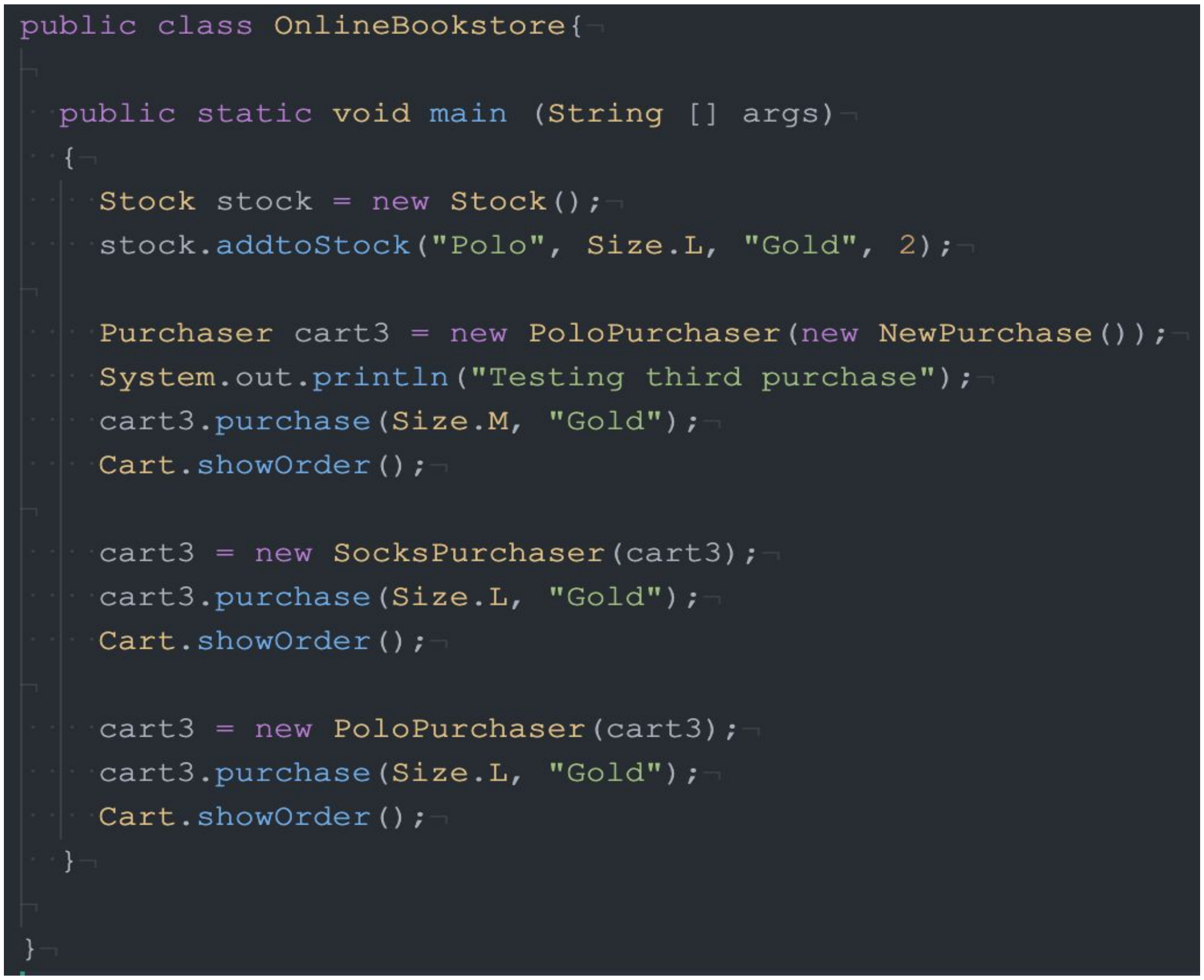
```
public class OnlineBookstore{

  public static void main (String [] args)
  {
    Stock stock = new Stock();
    stock.addtoStock("Shirt", Size.S, "Gold", 2);
    stock.addtoStock("Sweatshirt", Size.M, "Maroon", 2);
    System.out.println("Shirt stock:" + stock.shirtstock);

    Purchaser cart2 = new SweatshirtPurchaser(new NewPurchase());
    System.out.println("Testing second purchase");
    cart2.purchase(Size.M, "Maroon");
    Cart.showOrder();

    cart2 = new ShirtPurchaser(cart2);
    cart2.purchase(Size.S, "Gold");
    Cart.showOrder();

  }

}
```



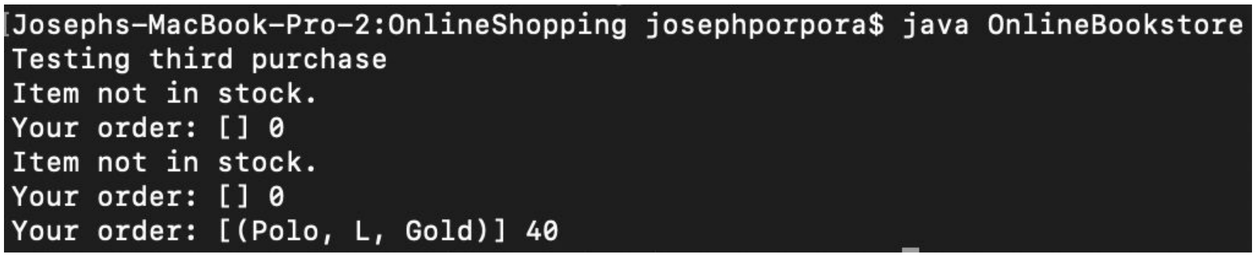
```
Josephs-MacBook-Pro-2:OnlineShopping josephporpora$ java OnlineBookstore
Shirt stock:[(Shirt, S, Gold), (Shirt, S, Gold)]
Testing second purchase
Your order: [(Sweatshirt, M, Maroon)] 50
Your order: [(Sweatshirt, M, Maroon), (Shirt, S, Gold)] 80
```

In this example, the same items are added to stock and purchased as example 1, but in a different order. FIrst a medium maroon sweatshirt is purchased, then a small gold shirt. This cart is functionally the same as the one in example 1, which illustrates why the decorator pattern is suited to this use. Online shoppers need to be able to add anything to their cart in any order. The combination and ordering does not functionally change almost anything about the cart, except the price.

Example 3:



```
public class OnlineBookstore{

  public static void main (String [] args)
  {
    Stock stock = new Stock();
    stock.addtoStock("Polo", Size.L, "Gold", 2);

    Purchaser cart3 = new PoloPurchaser(new NewPurchase());
    System.out.println("Testing third purchase");
    cart3.purchase(Size.M, "Gold");
    Cart.showOrder();

    cart3 = new SocksPurchaser(cart3);
    cart3.purchase(Size.L, "Gold");
    Cart.showOrder();

    cart3 = new PoloPurchaser(cart3);
    cart3.purchase(Size.L, "Gold");
    Cart.showOrder();
  }

}
```



```
Josephs-MacBook-Pro-2:OnlineShopping josephporpora$ java OnlineBookstore
Testing third purchase
Item not in stock.
Your order: [] 0
Item not in stock.
Your order: [] 0
Your order: [(Polo, L, Gold)] 40
```

In this example, we highlight the realistic issues a shopper could run into. First, the shopper tries to add a medium gold polo to their cart. Because there are only large gold polos in stock, they are given an error message. Then they try to purchase a large gold pair of socks. Because there are no socks in stock so they are given in an error message. Lastly, they resign to buying a large gold polo, which is added to cart.