**Oregon Institute of Technology**

Decorator Lab Fall  
For Demonstration in class on February 9

**Overview**

In this lab you will work with code that handles orders. The idea is that over time, items are added into an Order instance and when the customer is finished they proceed to the payment phase. Currently, this involves printing out all the items and stating the total cost for all items in the order. However, the company now wants to make the overall cost of the order a function, not only of the items ordered, but also the type of shipping the customer desires. In addition, they see an opportunity to add extra revenue by charging for credit card purchases.

Your task is to enable the company to do this without modifying the existing Order functionality. To do this, yep you guessed it; you will use the Decorator pattern.

### Steps:

1. Open the DecoratorLabStarterCode project, familiarize yourself with the code and run it. It should display a set of items in the Order instance and a total cost for the items in the order. Your challenge is to add support for express shipping. This will add a fixed cost of $4.00 to the price of the order. Since there can’t be any changes to the Order type functionality, you can’t add a bool to the Order type to represent express shipping. Instead you will refactor the code and use the decorator pattern to facilitate adding the cost of shipping.  
     
   Some things to note in the starter code:

I have implemented C# properties instead of C++ style setters and getters in the OrderItem class. For example:  
  
private string productCode;  
public string ProductCode  
{  
 get { return productCode; }  
}  
  
This code allows me to create an instance of OrderItem and then get the value of productCode by accessing the ProductCode property using the dot operator.  
  
OrderItem item = new OrderItem();  
Console.WriteLine(“productCode: {0}”, item.ProductCode);  
  
Look at the Order.PrintOrderItems() method in the Order class for another example. If this is confusing ask for an explanation.

1. The first step in using the decorator pattern is to identify the type you wish to decorate—in this case the Order type. In order to decorate the Order type at runtime, the Order class needs a contract that contains its methods in an abstract form. The client code using the Order instance will be written against this contract. This allows the creation of concrete decorator objects that implement this new abstract Order type.

The steps for refactoring are:

* 1. In the dll, create a new public abstract class called AbstractOrderBase and make all methods public abstract.
  2. Remove the items list from the Order class and add it as a protected data member of AbstractOrderBase.
  3. Derive Order from AbstractOrderBase
  4. Override each of the AbstractOrderBase methods in Order. Each method should implement the original Order logic.

Recompile the code. It should run just the same as before. However, by implementing AbstractOrderBase, we now have a contract that we can use to decorate an Order.

1. Now we can start adding classes that will decorate Order with additional behavior. The class diagram below shows the type hierarchy to implement. The first step is to create a public OrderDecorator type that all Order decorators will derive from. This makes sense since all decorators need to contain a reference to the object they are decorating. It also provides the “pass through” functionality for methods in Order class that are not being decorated. (Refer to the OtherMethods() functionality in the Beverage demo.)



1. Create a public base class for your decorators. Call it OrderDecorator and derive it from AbstractOrderBase. Add a protected data member of type AbstractOrderBase—this is the type being decorated. Add a protected c’tor that takes one parameter of type AbstractOrderBase. Use the c’tor to initialize your protected data member to the value passed in as a parameter.

Implement each of the abstract methods defined in the AbstractOrderBase type. Each method's implementation should simply make the same method call using the protected data member you declared.

The application should compile.

1. Now for the fun part… the decoration code. Create a public express shipping decorator type, call it ExpressDeliveryOrderDecorator. Derive ExpressDeliveryOrderDecorator from OrderDecorator.

Add a c’tor to ExpressDeliveryOrderDecorator that takes an AbstractOrderBase as its only parameter. Pass the parameter to the base class c’tor.

Start by decorating GetTotalCost(). It is very important that you understand that the other methods are handled by the OrderDecorator class. If this doesn’t make sense, please ask in class.

Said another way, the OrderDecorator class does all of the heavy lifting in the Decorator pattern. Once it is implemented, each Decorator only needs to focus on the methods that need decorating.

OK, getting back to GetTotalCost(). In this method, add a $4 delivery charge to the total items cost of the object being decorated and return the accumulated value.   
  
Also, decorate PrintOrderItems(): add a Console.WriteLine that explains to the customer that a shipping cost may apply—call OrderDecorator.PrintOrderItems() to get the pass through behavior (Total Cost of Items), then add Console.WriteLine("Grand Total with Shipping {0:C}", GetTotalCost()); to output to the Grand Total to the console. Note that this CW calls the decorated GetTotalCost() which adds the $4 delivery charge for the Grand Total.  
  
Time to modify Main(). After creating an Order instance, decorate it by creating an ExpressDeliveryOrderDecorator instance and passing in the Order instance as a parameter. Now invoke PrintOrderItems via the Decorator. Compile and run the code, you should now see your order printed out with the shipping cost message and the additional shipping charge in the Grand Total.   
  
Your output should look something like this:  
  
A Shipping Cost May Apply

BroncoHats x 2 @ $1.50 = $3.00  
BroncoGloves x 1 @ $3.00 = $3.00  
BroncoSocks x 6 @ $1.90 = $11.40  
BroncoBanners x 3 @ $8.00 = $24.00  
BroncoFootballs x 4 @ $5.60 = $22.40  
BroncoJerseys x 2 @ $2.30 = $4.60

Total Cost of Items $68.40

Grand Total with Shipping $72.40

1. Further extend the code and create two decorators for paying by credit card. If the customer chooses to pay by Visa there is a $2.00 charge. If they choose to pay by American Express there is a $5 charge. Create two new decorators one for Visa and one for American Express.   
     
   In Main(), decorate the Order instance with the Visa decorator and call PrintOrderItems(). Then decorate the order instance with the American Express decorator and call PrintOrderItems().   
     
   Similar to the ExpressDeliveryOrder decorator, your output should explain to the user the details of the extended behavior.

|  |
| --- |
|  |

Questions:

Write some code in main that demonstrates the pass-through functionality of the OrderDecorator class.

Why is the OrderDecorator c’tor protected?

Draw a sequence diagram that shows the method calls that occur when creating an Order instance, an ExpressDeliveryOrder decorator instance, and a call to the decorator’s PrintOrderItems();