Decorator Pattern

Estimated time for completion: 60 minutes

Overview:

In this lab you will see how to apply the decorator pattern to enable you to write code that is closed for modification but still open to extension.

Goals:

- Develop and put to use the Decorator pattern

Lab Notes:

Part 1: Using a Decorator

For this part of the lab you will work with a piece of code that currently handles orders. The idea is that over time items are placed into an order and then when the customer is finished they proceed to the payment phase. Currently this just involves printing out all the items and stating the total for the order. However the company now wants to make the overall cost of the order a function not only of the items in the basket but also the type of shipping the customer desires. In fact they see lots of opportunity to add extra revenue by charging for credit card purchases too. Your task is to enable the company to do this by not greatly affecting the existing code. To do this, yep you guessed it; you will use the Decorator pattern

1. Open the solution, familiarize yourself with the code and run it. It should display a set of items in the order and the total for the order. You have first been asked to add support for express shipping; this will add a fixed cost of £4.00 to the price of the order. Since there will be minimal changes to the Order type you will not add a Boolean to the Order type to represent express shipping instead you will refactor the code to allow use of the decorator pattern.
2. The first step in using the decorator pattern is to identify the type you wish to decorate, in your case this is the Order type. The Order type is currently a concrete type containing the implementation. In order for you to efficiently augment the Order type at runtime you need the Order type to implement a contract that contains all its methods but in an abstract form. The client code using the order will be written against this contract (abstract class or interface), this will then allow you to create other concrete objects that can wrap up any objects that implement this new abstract order type.
3. The simplest way to perform this refactoring is to :-
    a. Copy the current Order class, to a new class called SimpleOrder
    b. Make the original Order class an abstract class, and make all its methods abstract
    c. Make SimpleOrder derive from Order
    d. Make each of the SimpleOrder methods override the Order versions
    e. Change the new Order statement in Main to Order order = new SimpleOrder()

4. You should now be able to recompile the code and it should still work as before. All you have done is simply created a contract for your SimpleOrder to implement, but the remaining code works against the contract allowing you now to vary the implementation of the Order object. This will enable you to decorate the Order with a new type as long as it implements this contract, which is what the client code is expecting.

```
public abstract class Order
{
   public abstract void AddItem(string productCode, int quantity, decimal
   cost, decimal weight);

   public abstract void PrintOrderItems();
   public abstract IEnumerable<OrderItem> Items { get; }

   public abstract decimal TotalCost { get; }

}
```

5. You are now in a place to start adding classes that will contain the additional behavior you wish to add to your Order. The image below show the type hierarchy you are aiming for. The first step is to create a Decorator base type that all order decorators will derive from, this makes sense since all decorators will need to contain a reference to the object they are decorating and thus you can write this once. It also has the advantage in that you can implement a pass through implementation of each method, allowing the derived decorator types to only override the methods they are interested in changing.

6. Create the base decorator type and call it OrderDecorator and derive it from Order, add a constructor that takes something of type Order, this is the object you are decorating. Add a protected field that you will use to hold on to this object and initialise it inside the constructor, implement each of the abstract methods defined on the order type. Each method's implementation should simply make the same method call but to the wrapped object, this is very much akin to calling base.method().

```
public class OrderDecorator : Order
{
   protected Order wrappedOrder = null;

   public OrderDecorator(Order order)
   {
      wrappedOrder = order;
   }

   public override void AddItem(string productCode, int quantity, decimal
   cost, decimal weight)
   {
      wrappedOrder.AddItem(productCode, quantity, cost, weight);
   }

   public override IEnumerable<OrderItem> Items
   {
      get { return wrappedOrder.Items;  }
```



```
    }

    public override void PrintOrderItems()
    {
        wrappedOrder.PrintOrderItems();
    }

    public override decimal TotalCost
    {
        get { return wrappedOrder.TotalCost; }
    }
}
```

7. The application should still compile
8. Now create the express shipping decorator type, call it ExpressDeliveryOrder, make it derive from OrderDecorator and simply override the Total property getter. In here you want to get the total from the object you are wrapping up and then add your £4.0 delivery charge and return that figure In addition to that you should perhaps override PrintOrderItems, to also print out the fact that you are adding this amount to the order.

```
public class ExpressDeliveryOrder : OrderDecorator
{
    private const decimal DeliveryCharge = 4.0M;


    public ExpressDeliveryOrder(Order order) : base(order)    {    }

    public override void PrintOrderItems()
    {
        base.PrintOrderItems();

        Console.WriteLine("Express delivery charge of {0:C}" , DeliveryCharge);
    }

    public override decimal TotalCost
    {
        get
        {
        return base.TotalCost + DeliveryCharge;
        }
    }
}
```

9. Modify Main so that after creating the order, you change it into an Express Shipping Order, by simply inserting the following line before you call PrintOrderConfirmation

```
order = new ExpressDeliveryOrder(order);
```

10. Compile and run the code, you should now see your order printed out with the additional shipping charge.
11. Extend the code further to create a decorator for paying by credit card. If the customer chooses to pay by Visa there is a £2.00 charge, and there is a £10 for American Express.

Implement this either as a single decorator with the card type passed as a parameter or as two different decorators, one for Visa and one for American Express

```
public enum CardType { Visa, American };

public class CreditCardOrder : OrderDecorator
{
   private CardType cardType;
   private static Dictionary<CardType, decimal> cardToFeeMapper = new
   Dictionary<CardType, decimal>();

   static CreditCardOrder()
   {
      cardToFeeMapper.Add(CardType.Visa, 3.0M);
      cardToFeeMapper.Add(CardType.American, 10.0M);
   }

   public CreditCardOrder(CardType cardType , Order order) : base(order)
   {
      this.cardType = cardType;
   }

   public override void PrintOrderItems()
   {
      base.PrintOrderItems();

      Console.WriteLine("{0} surcharge of {1:C} " , cardType ,
   cardToFeeMapper[cardType] );
   }

   public override decimal TotalCost
   {
      get
      {
            return base.TotalCost + cardToFeeMapper[cardType];
      }
   }

}
```

12. You should now be able to build orders that have no Express delivery, Express delivery and some additional credit card charge, and an order that has just a credit card charge. Hopefully you can see that if you added additional order variants you will not end up with class explosion as you would have with inheritance nor do you end up with lots of if/else logic.

Part 2: Employee Report

Open the solution and familiarize yourself with the code. It is a relatively simple app that contains a singleton type called HumanResources which provides a method called GetEmployees which in turns returns an enumerable set of Employees. The enumerable set of employees can then be used in a foreach loop. The PrintEmployees method inside Program.cs takes advantage

of this to produce a colorful output of all employee details. Compile and run the app and see the result. Whilst the application is ok, it is not long before requests come out for different reports, like :-

- *I want to see all female employees.*
- *I want to see all employees earning more than $xxx*
- *I want to see all high paid male employees*

In other words you want to filter the employees that appear in the report. One obvious way would be to produce multiple versions of PrintEmployees which will perform the appropriate filtering, but that would lead to cut + pasting of the formatting code. You could also provide multiple methods on the HumanResources type to return the various collections of employees which would mean that you need to keep modifying the type for each type of report. What you will do instead is use the decorator pattern to decorate the enumerable stream of employees with filters, each filter will filter out any employee not matching the filter criteria

1. The first step when using the decorator pattern is to determine the base functionality you are decorating. In this case it is the IEnumerable<Employee>. Filters work against contracts and as IEnumerable<Employee> is already an interface no change to the type definition is necessary.
2. When filtering the next step is to create an abstract base class that will implement the interface being filtered. This abstract base class also provides the functionality required to hold a reference to the object you are decorating. This abstract class will then become the base class for all your decorators, each decorator then does whatever it needs to do before and/or after invoking the functionality of the object it is decorating.
3. Write that abstract base class now and call it EnumerableEmployeeDecorator remember it should implement the IEnumerable<Employee> interface.

```
public abstract class EnumerableEmployeeDecorator : IEnumerable<Employee>
{
   protected IEnumerable<Employee> wrappedObject;

   public EnumerableEmployeeDecorator(IEnumerable<Employee> employees)
   {
      wrappedObject = employees;
   }

   public abstract IEnumerator<Employee> GetEnumerator();

   // Non generic version
   System.Collections.IEnumerator
   System.Collections.IEnumerable.GetEnumerator()
   {
      return GetEnumerator();
   }
}
```

4. The next stage is then to create the appropriate decorator functionality. First create a GenderFilter that will allow you to just print Male or Female employees. Define a class

called GenderFilter and make this derive from your decorator base class. It will need to take a Gender to filter on as a constructor parameter in addition to the object it is wrapping it up which in your case is anything that supports IEnumerable<Employee>. Create a private field to hold onto the gender, and then make use of this inside the GetEnumerator method when deciding if to return an Employee. The code below shows you a basic implementation of GetEnumerator taking advantage of C# v2.0 yield return keyword.

```
public override IEnumerator<Employee> GetEnumerator()
{
   foreach (Employee employee in wrappedObject)
   {
      // Add conditional logic here to decide if to return the employee or
   not
      yield return employee;
   }
}
```

5. You should end up with a Gender filter that looks something like this

```
class GenderEmployeeFilter: EnumerableEmployeeDecorator
{
   private Gender genderToMatch;
   public GenderEmployeeFilter( Gender genderToMatch ,
      IEnumerable<Employee> employees) : base(employees)
   {
      this.genderToMatch = genderToMatch;
   }

   public override IEnumerator<Employee> GetEnumerator()
   {
      foreach (Employee employee in wrappedObject)
      {
            if (employee.Gender == genderToMatch)
            {
                  yield return employee;
            }
      }
   }
}
```

6. Check the code compiles and then go ahead and make use of the filter from inside Main.

```
static void Main(string[] args)
            {
            IEnumerable<Employee> employees =
   HumanResources.GetInstance().GetEmployees();

            employees = new GenderEmployeeFilter(Gender.Male, employees);

            PrintEmployees(employees);
            }
```


7. Now add an additional filter for salaries above a certain level. Test it and make sure you are happy with it.

```
class SalaryFilter : EnumerableEmployeeDecorator
{
   private double employeeSalaryOver;
   public SalaryFilter(double salaryOver , IEnumerable<Employee>  employees)
   : base(employees)
   {
      employeeSalaryOver = salaryOver;
   }
   public override IEnumerator<Employee> GetEnumerator()
   {
      foreach (Employee employee in wrappedObject)
      {
            if (employee.Salary > employeeSalaryOver)
            {
                  yield return employee;
            }
      }
   }
}
```

8. If you want to now produce an additional report that displays all Males earning over a certain amount do you need an additional class? Your answer should be no. Demonstrate that by wrapping your gender filter with your salary filter
9. How can you add support for sorting by Salary? When complete you should be able to see that the decorator pattern enables you to easily add behavior at runtime without the need for modification to existing code and without relying on inheritance.