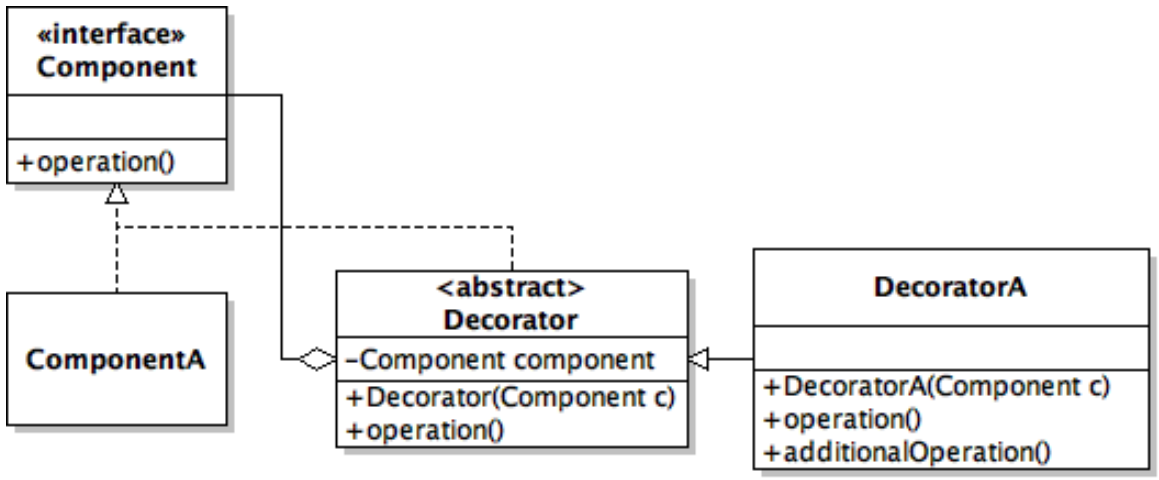
CECS 277 – Lecture 22 – Design Patterns

Decorator

The Decorator pattern is a Structural design pattern that allows you to dynamically attach additional behaviors and responsibilities on to your objects. It is a common alternative to subclassing and is commonly used when there would be an explosion of subclasses from having many different combinations of options for your classes (ex. subclasses of pizza where each has a different combination of toppings). It can also be a useful alternative if you are unable to modify the original class. Once an object has been decorated, it is no longer an object of the base type, and subsequently, cannot call any methods of that class. However, it is still an object of the Component type and can call any methods from there.

Decorator UML – The Decorator pattern has a Component interface (or abstract class) with methods that the base object(s) need to override. The abstract Decorator class implements the Component interface and has an instance of the Component that is initialized through the constructor. The overridden methods in the Decorator are then called on the Component object. All concrete Decorator classes extend from the Decorator and should override the Component's methods, these should call the Decorator's (super) version along with any additional behavior that Decorator should do.



Decorator Template Code:

```
public interface Component {
   public void operation( );
}
public class ComponentA implements Component {
   public void operation( ) {
      System.out.print( "CompA->" );
   }
}
public abstract Decorator implements Component {
   private Component comp;
   public Decorator( Component c ) {
      comp = c;
   }
```

```
   public void operation( ) {
      comp.operation( );
   }
}
public class DecoratorA extends Decorator {
   public DecoratorA( Component c ) {
      super( c );
   }
   public void operation( ) {
      super.operation( );
      additionalOperation( );
   }
   public void additionalOperation( ) {
      System.out.print( "DecorA->" );
   }
}
public class Main{
   public static void main( String [] args ) {
      Component a = new DecoratorA( new ComponentA( ) );
      a.operation( );
   }
}
/* Output:
CompA->DecorA->
*/
```

Decorator Example Code:

```
public interface Pizza {
   public double cost( );
}
public class PizzaSmall implements Pizza {
   @Override
   public double cost( ) {
      System.out.println( "Small Pizza....$5.00" );
      return 5;
   }
}
public class PizzaLarge implements Pizza {
   @Override
   public double cost( ) {
      System.out.println( "Large Pizza....$7.00" );
      return 7;
   }
}
public abstract class PizzaDecorator implements Pizza {
   private Pizza pizza;
```

```
   public PizzaDecorator( Pizza p ) {
      pizza = p;
   }
   @Override
   public double cost( ) {
      return pizza.cost( );
   }
}
public class Mushroom extends PizzaDecorator {
   public Mushroom( Pizza p ) {
      super( p );
   }
   public double cost( ) {
      display( );
      return .5 + super.cost( );
   }
   public void display( ) {
      System.out.println( "Mushroom......$0.50" );
   }
}
public class Pepperoni extends PizzaDecorator {
   public Pepperoni( Pizza p ) {
      super( p );
   }
   public double cost( ) {
      display( );
      return 1 + super.cost( );
   }
   public void display( ) {
      System.out.println( "Pepperoni.....$1.00" );
   }
}
public class Main {
   public static void main( String [] args ) {
      Pizza p = new Pepperoni( new Mushroom( new PizzaSmall()));
      System.out.println( "Pizza Order" );
      System.out.println( "-----------" );
      System.out.println( "Total.........$" + p.cost( ) );
   }
}
/* Pizza Order
   -----------
   Pepperoni.....$1.00
   Mushroom......$0.50
   Small Pizza...$5.00
   Total.........$6.50
*/
```