Decorator Tasarım Deseni

Nedir?

Decorator tasarım deseni, **structural** tasarım desenlerinden biridir. Bir nesneye dinamik olarak yeni özellikler eklemek için kullanılır. Kalıtım kullanmadan da bir nesnenin görevlerini artırabileceğimizi gösterir.

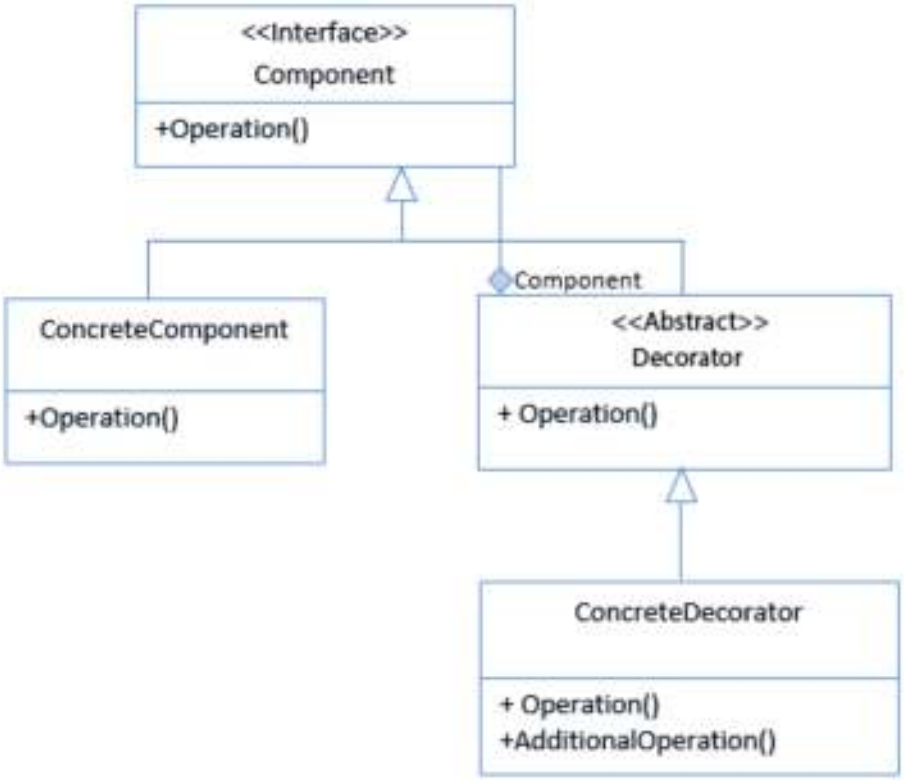
Not: Bir sınıfın nesnesine runtime zamanında eklenen özellikler, bu sınıftan yaratılmış diğer nesneleri etkilemez.

Ne zaman Kullanılır?

Runtime zamanında bir nesneye yeni özellikler eklemek istiyorsak kullanabiliriz.

Nasıl Kullanılır?

Decorator tasarım desenini **Decorator** sınıfları ve **Component** sınıfları şeklinde iki kısma ayırabiliriz. **Component** sınıfları içerisinde **Decorator** süper sınıfı bulunur. Yani şu şekilde bir yapı oluşturmak gereklidir:



Decorator sınıfı **Component** sınıfından türemiştir. Aynı zamanda **Decorator** sınıfı ile **Component** sınıfı arasında **HAS-A** ilişkisi vardır. Bunun anlamı, **Decorator** sınıfı içerisinde **Component** türünden **instance** değişken bulunur (Composition yapısı).

Decorator sınıfı **abstract** veya **interface** olabilir. Somut sınıf kullanmamak gereklidir.

Dinamik olarak özelliklerin ekleneceği nesne, **ConcreteComponent** sınıfından türetilir.

ConcreteDecorator nesnesi, **ConcreteComponent** nesnesine özelliklerin eklenmesi işlemini yapar.

Not: Decorator tasarım deseninde kalıtım, sadece sınıflar aynı türe sahip olsunlar diye kullanılmaktadır. Nesnenin fonksiyonlarını composition ile sağlıyoruz. Eğer kalıtıma bağlı kalsaydık, compile time zamanında nesnenin davranışı belirlenecekti. Böyle bir durumda ise dinamik olarak yeni özellikler ekleyemeyecektik. Bu konuyla ilgili detaylı bilgiler için aşağıdaki bağlantıları inceleyebilirsiniz:

Composition Over Inheritance (http://en.wikipedia.org/wiki/Composition_over_inheritance)
What is composition as it relates to object oriented design? (http://stackoverflow.com/questions/3441090/what-is-composition-as-it-relates-to-object-oriented-design)

Faydaları Nedir?

1. **loosely-coupled** uygulamalar yapmayı sağlar.
2. Runtime zamanında(dinamik olarak) bir nesneye yeni özellikler eklenmesini sağlar.
3. Özellikleri kalıtım yolu dışında composition ve delegation (http://www.softwarevol.com/en/tutorial/Distinguishing-Between-Delegation-Composition-And-Aggregation) ile de alınabilmesini sağlar.
4. **open-closed** prensibinin uygulandığı tasarım desenidir.

Open-Closed Prensibi

Tanım olarak, sınıfların geliştirilmeye açık olması, değiştirilmeye kapalı olması demektir. Bu tanıma göre uygulaması imkansız görünen **open-closed** prensibi, bazı nesneye yönelik programlama teknikleri ile uygulanabilmektedir. Örnek olarak Observer (http://www.softwarevol.com/tutorial/Gozlemci-Observer-Tasarim-Deseni) tasarım deseni. Hatırlarsak Observer tasarım deseninde, yeni Observer sınıfları eklerken Subject sınıfında herhangi bir değişiklik yapmıyorduk. Aynı şekilde Decorator tasarım deseni de bu prensibi uygulamaktadır.

Not: Uygulama geliştirirken, her kısımda bu prensibi uygulamak kodların karmaşıklığını artıracaktır. Eğer uygulamanızda değişmesi muhtemel kısımlar varsa, bu kısımlarda prensibi uygulamak gereklidir.

Örnek Kullanım Alanları

1. **java.io.InputStream, OutputStream, Reader** ve **Writer** sınıflarının tüm alt sınıflarında kullanılır.
2. **java.util.Collections** sınıfı ve **checkedXXX(), synchronizedXXX(), unmodifiableXXX()** metodlarında kullanılır.
3. **javax.servlet.http.HttpServletRequestWrapper** ve **HttpServletResponseWrapper** sınıflarında kullanılır.

Örnek Uygulama

Bu uygulamada decorator tasarım desenini kullanarak basit bir toplama ve çarpma işlemi yapacağız.

Kullanılacak Sınıflar

1. **Component** olarak kullanılacak interface: **Calculator**
2. Somut component olarak kullanılacak sınıf: **ConcreteCalculator**
3. **Decorator** olarak kullanılacak sınıf: **CalculateDecorator**
4. Somut decorator olarak kullanılacak sınıflar: **Sum** ve **Multiply**
5. Test sınıfı

Calculator interface:

```
1  /**
2   * Component interface. Tum decorator siniflari bu interface'i implement etmek zorundadir
3   */
4  public interface Calculator {
5      public double calculate();
6  }
```

Concrete Component:

```
1  /**
2   * Decorator isleminin yapilacagi sinif
3   */
4  public class ConcreteCalculator implements Calculator {
5      private double value=0;
6      public ConcreteCalculator(double value){
7          this.value=value;
8      }
9      @Override
10     public double calculate() {
11         return value;
12     }
13 }
```

Decorator Sınıfı

```
1  /**
```

```
 * Decorator sinifi. abstract tanimlamak zorunlu
 */
public abstract class CalculateDecorator implements Calculator {
    protected Calculator calculator; //eklemek zorunlu

    protected CalculateDecorator(Calculator calculator) {
        this.calculator = calculator;
    }

    @Override
    public double calculate() {
        return calculator.calculate(); //Calculator interfacedeki calculate metodunu cagirmasi zorunlu
    }
}
```

Somut Decorator Sınıfları

```
/**
 * Carpma isleminin yapildigi sinif
 */
public class Multiply extends CalculateDecorator {
    private double value; //carpim katsayisi
    protected Multiply(Calculator calculator,double value) {
        super(calculator);
        this.value=value;
    }

    @Override
    public double calculate() {
        return calculator.calculate()*value; //calculator nesnesi, CalculateDecorator sinifindan geliyor
    }
}
```

```
/**
 * Toplama isleminin yapildigi sinif
 */
public class Sum extends CalculateDecorator {
    private double value;
    protected Sum(Calculator calculator,double value) {
        super(calculator); //cagirmak zorunlu
        this.value=value;
    }

    @Override
    public double calculate() {
        return calculator.calculate()+value; //calculator nesnesi CalculateDecorator sinifindan geliyor
    }
}
```

Test Sınıfı

```
/**
 * Test sinifi
 */
public class Test {
    public static void main(String[] args) {
        //somut component sinifi her zaman en ice yazilir. Somut decorator siniflari
        //bu somut component sinifi sarmakla gorevlidir. Ayni zamanda kendilerini de sarabilirler
        //yani new Multiply(new Multiply()) seklinde de olabilir...
        Calculator calculator=new Multiply(new Sum(new Multiply(new ConcreteCalculator(12),4),4),4);
        double result=calculator.calculate();
        System.out.println(result);
    }
}
```

Somut component sınıfı her zaman en içe yazılır. Somut decorator sınıfları bu **somut** component sınıfı sarmakla gorevlidir. Ayni zamanda kendilerini de sarabilirler. Yani

```
new Multiply(new Multiply());
```

seklinde de olabilir.

Uygulamamızın UML Diagramı