



Decorator Design Patterns

by Adesh Srivastava · Aug. 08, 18 · Java Zone · Tutorial

Java-based (JDBC) data connectivity to SaaS, NoSQL, and Big Data. Download Now.

In an object-oriented paradigm, decorators are one of the largely used structural design patterns. This pattern changes the functionality of objects at runtime without impacting its blueprint or the existing functionality.

While working with these patterns, we might confuse decorators with an inheritance. However, there is a difference between the two. Decorators are used when we have to add and remove responsibilities from an object dynamically. Whereas, inheritance can do the same, but not at run time.

Decorator patterns simply allow you to dynamically add functionality to a specific object instead of an entire class of objects.

The best thing is that we can bind one object with the decorator and wrap their result with another decorator.

Why Decorators?

Let us assume that we are building a simple design to keep the data and model of the available mobile phones in the market. Here, we have a number of mobile phones to take into account with various models and overlapping specs. How can we implement the design?

One way to do that is by using the traditional inheritance. We will create a base `Phone` class and multiple subclasses for different phone combinations in specs and the model.

Here is the catch — we cannot create a number of mobile phones through inheritance, especially when we know specs will overlap for multiple

phones. We need to think of something better. Decorator design patterns have the answer.

Implementation

Suppose we want to have different types of mobiles phones.

Let us create a `Phone` interface and `BasicPhone` class (having basic features) to implement it.

```
1  public interface Phone {  
2  
3      public void printModel();  
4  
5  }  
6  public class BasicPhone implements Phone {  
7  
8      @Override  
9      public void printModel() {  
10         System.out.println("Basic Phone");  
11     }  
12 }
```

Now, we will create a `PhoneDecorator` class that implements the `Phone` interface, having one `Phone` object as its property.

```
1  public class PhoneDecorator implements Phone {  
2  
3      public Phone phone;  
4  
5      public PhoneDecorator(Phone phone) {  
6         this.phone = phone;  
7     }  
8  
9      @Override  
10     public void printModel() {
```

```
11         this.phone.printModel();
12     }
13 }
```

It has a HAS-A relationship with the interface `Phone` as the `Phone` object must be accessible to all the child decorators.

Now, let us create two more classes extending `PhoneDecorator` — say `AndroidPhone` and `IPhone`.

```
1 public class AndroidPhone extends PhoneDecorator {
2
3     public AndroidPhone(Phone phone) {
4         super(phone);
5     }
6
7     @Override
8     public void printModel() {
9         super.printModel();
10        System.out.println("Adding Features of Android");
11    }
12 }
```

Similarly,

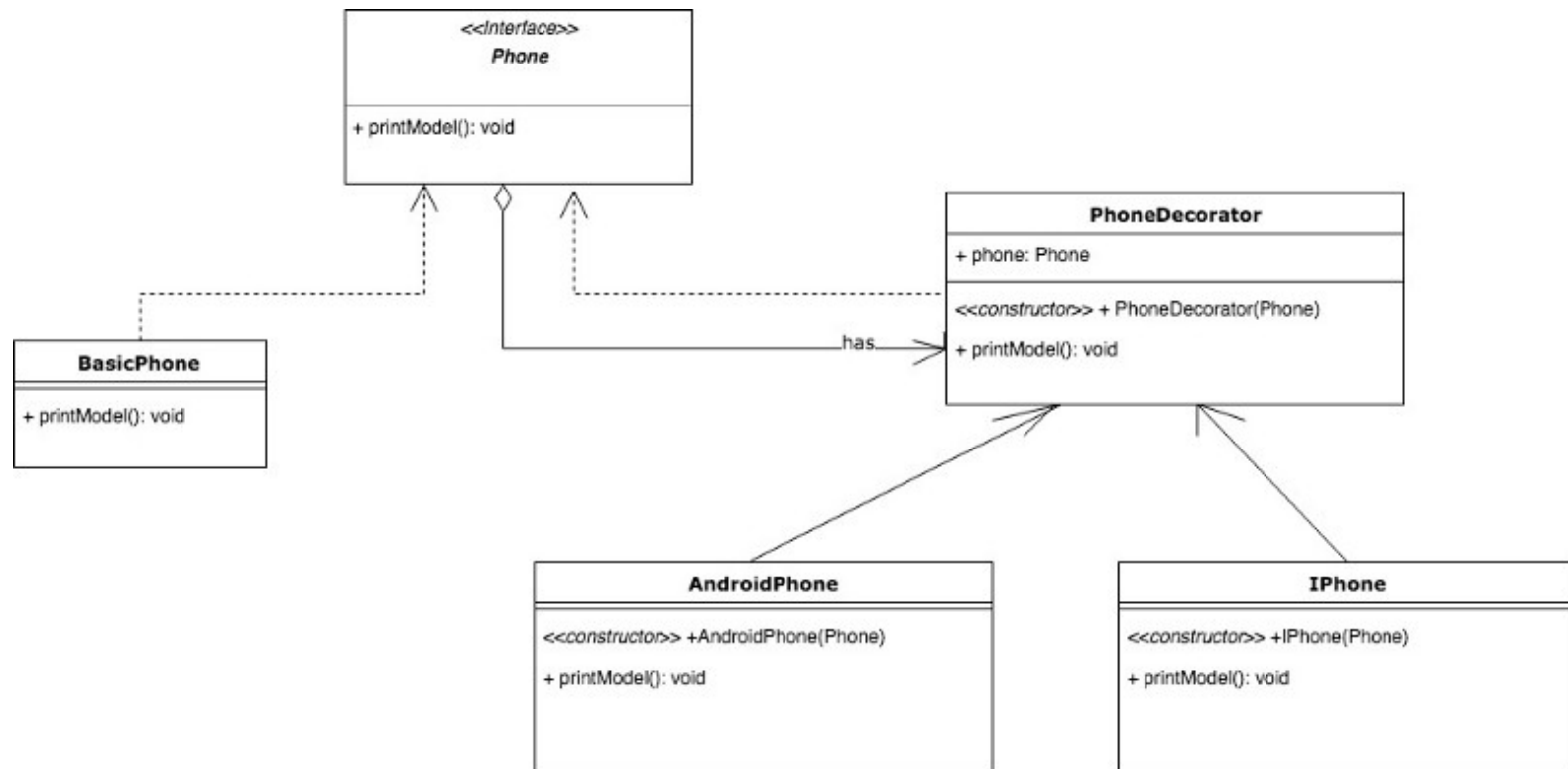
```
1 public class IPhone extends PhoneDecorator {
2
3     public IPhone(Phone phone) {
4         super(phone);
5     }
6
7     @Override
8     public void printModel() {
9         super.printModel();
```

```

10      System.out.println("Adding Features of iPhone");
11  }
12  }

```

We are almost done! Let us checkout the class diagram:



Now, let's test it!

```

1  public class Test {
2
3      public static void main(String[] args) {
4          System.out.println("Test 1\n");
5          Phone phone = new AndroidPhone(new BasicPhone());
6          phone.printModel();
7          System.out.println("\nTest 2\n");

```

```
8         Phone phone1 = new iPhone(phone);
9         phone1.printModel();
10     }
11 }
```

Here is the output:

```
1  Test 1
2  Basic Phone
3  Adding Features of Android
4  Test 2
5  Basic Phone
6  Adding Features of Android
7  Adding Features of iPhone
```

Here we can see how the **phone** object can behave dynamically, rendering property of the `BasicPhone` in the first case. Here, `phone = new AndroidPhone(new BasicPhone())` renders properties of the `BasicPhone` first and, then, the `AndroidPhone`. However, the same `phone`, when passed to `new iPhone(phone)`, shows properties of `BasicPhone` and `AndroidPhone`, followed by the `iPhone`. Essentially, you can write wrappers over wrappers without touching the basic class blueprint.

Conclusion

Now, we know that decorators are used when we dynamically assign the behaviours to objects without messing up the code that uses these objects. Inheritance also does the same, but in a static manner.

Wearing clothes is an example of using decorators. When we are cold, you wrap yourself with a sweater. If it is still cold, we can wear a jacket on top. If it is raining, we can put on a raincoat.

All of these are not part of us. Therefore, we can easily remove them whenever they are not needed. One of the applications of decorators can be found in `java.io` package (1 & 2). For example :

```
1  BufferedInputStream bis=new BufferedInputStream(new
2  FileInputStream(new File("abc.txt")));
```

Though decorators have their own cons like every other good thing, such as presence of lots of small classes and problems in configuring multi-wrapped properties, these are good to know because of their ability to compose complex objects into simple ones, instead of having monolithic, single point agenda classes. If used intelligently, they might come handy in your next LLD!

Happy Coding!

Connect any Java based application to your SaaS data. Over 100+ Java-based data source connectors.

Like This Article? Read More From DZone



Top 5 Online Courses to Learn Java Design Patterns in 2019



How Functional Programming Will (Finally) Do Away With the GoF Patterns




On One Aspect of Design Patterns: Flexibility



**Free DZone Refcard
Java Containerization**

Topics: DESIGN PATTERN - STRUCTURAL , JAVA , SOFTWARE DEVELOPMENT , DECORATOR DESIGN PATTERN , OBJECT ORIENTED PROGRAMMING , TUTORIAL

Published at DZone with permission of Adesh Srivastava . [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

Java Partner Resources

Will your own user auth or integrate pre-built identity management? [Whitepaper]

cta

|

Get insight into your code with IntelliJ IDEA.

JetBrains

|

Java Ecosystem 2018 Infographic by JetBrains

JetBrains

|

Building Reactive Microservices in Java: Asynchronous and Event-Based Application Design

Red Hat Developer Program

|