

Decoratorパターン徹底解説

クラスを変更せずに機能を追加する

例として、小さな美術写真会社の注文管理に必要な一連のクラスについて考えてみます。この会社では、写真プリントを販売しています。写真プリントには各種のサイズがあり、プリントする紙にはマット紙または光沢紙があり、額縁を付けることもできます。また、額装用のマットはなしでも1つでも複数でもよく、さまざまな色のものが用意されています。さらに、多くのストック写真代理店を通じて、電子的に画像を販売することも行っています。

このデータ・モデルを実装する場合、最初に思いつくのは継承かもしれませんが、用紙の仕上げ、用紙サイズ、額装マットの有無、額縁の有無の組合せすべてについて異なるクラスを準備しようとすれば、クラスが爆発的に増加してしまうのは明かです。さらに、マーケットの状況が変わり、別の変動要素を追加しようとしたときに、すべてが崩壊してしまいます。逆のアプローチとして、1つのクラスですべての組合せを扱うという選択肢もあります。しかし、それではif文だらけで複雑な、わかりにくく管理も難しい、問題があるクラスになってしまうでしょう。

この問題に適した解決策の1つが、Decoratorパターンです。このパターンを使用することで、クラスの外側から既存のコンポーネントに機能を追加できるようになります。これを実現するためには、Decoratorというクラス（通常は抽象クラス）を作り、元のComponent（機能を追加するクラス）と同じインタフェースを実装する必要があります。つまり、Decoratorは、インタフェースまたはクラス定義を必要に応じて使うことで、元のクラスの代わりとして振る舞えることが必要です。Decoratorは、Componentクラスを拡張するとともに、そのインスタンスを保持します。そのため、DecoratorはComponentとの間にhas-a関係とis-a関係の両方を持つことになります。Decoratorのコードは次のようになることが多いでしょう。

```
public abstract class Decorator extends Component {
    protected Component target;
    ...
}
```

この継承関係があるため、Decoratorやそのサブクラスは、元のクラスが使われていた場所で使うことができ、元のクラスにビジネス・メソッドを直接委譲します。このDecoratorは、サブクラス化して使用できます。通常、サブクラスでは、実際のターゲットに委譲するメソッドの一部または全部に「前処理」や「後処理」を付け加えます。サブクラスの存在意義はその点にあります。

先ほどの写真会社の例で、元のComponentに当たるのは、PhotoImageです。このクラスのコンストラクタには、画像の作品名と、販売する画像の原版が格納されているファイルの名前を渡します。以降で紹介するコードは、オンラインで公開しています。

`Decorator`となるクラスは、`PhotoImage`を拡張したもので、写真も含んでいます。続いて、2つのおもな販売方法として、`Print`および`DigitalImage`というデコレータを作成できます。`Print`は実際の写真プリントを表すことから、コンストラクタには用紙サイズ（インチ単位の幅と高さ）を含めます。`Print`には、`Frame`、`Mat`などの`Decorator`を作ることができます。また、必要に応じてパラメータを決めることができます（たとえば、額装マットには色が必要です）。

まず、この一連のクラスの使用例を示します。次のコードは、ImageSales.javaの一部です。

```
// デコレーションなしの画像を作成する
final PhotoImage image = new PhotoImage(
    "Sunset at Tres Ríos", "2020/ifd12345.jpg");
```

```
// 米国のレターサイズ用紙にプリントする
Print im1 = new Print(11, 8.5, image);
addToPrintOrder(im1);
```

```
// 2つ目の画像を19x11でプリントし、額縁と緑色の額装マットを付ける
Print im2 =
```

```
new Print(19, 11,
    new Frame(
        new Mat("Lime Green",
            new PhotolImage("Goodbye at the Station",
                "1968/ifd.00042.jpg"))));
addToPrintOrder(im2);

// 電子的に画像を販売する
PhotolImage dig = new DigitalImage(image, StockAgency.Getty, 135.00);
System.out.println(dig);
```

`addToPrintOrder()`メソッドは、`Print`引数を受け取ります。これによって、型の安全性が一定程度確保されます。今回の簡単なデモでは、このメソッドは`toString()`を呼び出して引数を出力し、委譲メソッドの効果を示すだけのものになっています。

`PhotolImage` (`Component`に相当するもの) のコードを次に示します。

```
/** PhotolImageは、ある時点で撮影した写真を表す。
 */
public class PhotolImage {
    /** 判読可能なタイトル */
    String title;

    /** 実際のピクセルが格納されている場所 */
    String fileName;

    /** 画像ファイルのピクセル数 */
    int pixWidth, pixHeight;

    public PhotolImage() {
        // 空、Decoratorで使用
    }
}
```



```
public PhotoImage(String title, String fileName) {
    super();
    this.title = title;
    this.fileName = fileName;
}

/** 出力可能な説明を取得。toString()よりも詳しくすることも可能だが、
 * いずれにしても、これは委譲メソッドの例
 */
public String getDescription() {
    return getTitle();
}

/** デフォルトのtoString()では、getDescriptionをそのまま使用 */
@Override
public String toString() {
    return getDescription();
}

// setterとgetter...
}
```

DecoratorクラスであるImageDecoratorのコードの一部も示します。

```
public abstract class ImageDecorator extends PhotoImage {
    protected PhotoImage target;

    public ImageDecorator(PhotoImage target) {
        this.target = target;
    }
}
```

```
@Override  
public String getDescription() {  
    return target.getDescription();  
}
```

```
@Override  
public String toString() {  
    return getDescription();  
}  
}
```

デコレータの1つであるPrintクラスのコードの一部も示します。

```
/** Printは、PhotoImageと、プリントする用紙サイズを表す。  
 */  
public class Print extends ImageDecorator {  
    /** 米国のユーザー用に、PrintWidthおよびPrintHeightの単位としてインチを使用 */  
    private double printWidth, printHeight;  
  
    public Print(double printWidth, double printHeight, PhotoImage target) {  
        super(target);  
        this.printWidth = printWidth;  
        this.printHeight = printHeight;  
    }  
  
    @Override  
    public String getDescription() {  
        return target.getDescription() + " " +  
            String.format("(%4.1f x %4.1f in)",  
                getPrintWidth(), getPrintHeight());  
    }  
}
```



```
}
// setterとgetter...
}
```

このコードで重要な部分の1つは、`Print`クラスの`getDescription()`がどのようにしてターゲット・クラスの同名メソッドを呼び出しつつ、独自の機能を追加しているかという点です。Decorator/パターンの「委譲するが機能を追加する」という処理を行っているのが、この部分になります。そのため、メイン・プログラムを実行すると、`Print`オブジェクトから次のような出力が得られます。

Goodbye at the Station, Matted(Lime Green), Framed (19.0 x 11.0 in)

コンストラクタの引数のはじめに副次的な特徴（用紙サイズ、額装マットの色）を入れるのはおかしいと感じる方もいらっしゃるかもしれません。筆者と同じようにこういった点が気になるという場合、いつもはおもな要素

(つまりComponent) を最初の引数にしているのでしょう。ただし実際のところ、複数のデコレータを組み合わせることが多いコーディング・スタイルでは、副次的な項目を最初に置くことにより、コーディングが容易になります。メイン・プログラムに戻り、何段階もネストしたコンストラクタ呼出しがある場合に、括弧と終わりの引数がどのように並ぶかを考えてみてください。それでも気になるという方は、無理に合わせる必要はありません。これはスタイルの問題にすぎません。いずれかの方法を選び、階層の中でそれを統一すればよいでしょう。

一般的なクラス階層を図1に示します。この図で、`PhotoImage`は`Component`に、`Print`および`DigitalImage`は`Decorator`サブクラスに、`Mat`および`Frame`はその他のデコレータに当たります。先ほどの例にあった操作は`getDescription()`だけですが、本番用のコードには、他のメソッドも存在することでしょう。

PrintおよびDigitalImageは、図1のConcreteComponentのように直接PhotoImageをサブクラス化するのではなく、デコレータとしています。このようにする必要がある、特別な理由はありません。デコレータにしたのは、コード例でそうしているように、PhotoImageを1つ作成し、それを使用してPrintとDigitalImageの両方を作成できるようにするためです。実際のクラス階層で、必ずしもこれと同じことをする必要はないでしょう。この種の

Decoratorパターンをよく見かけるとい方がいるとすれば、それもそのはずです。Javaが始まって以来、一般的なjava.io StreamやReader、Writerはこのパターンを使って動作しています。




```
public class WidgetService {  
  
    @Transactional  
    public void saveWidget(Widget w) {  
        entityManager.persist(w);  
    }  
}
```

内部的には、いずれのフレームワークでもインターセプタを提供しています。このインターセプタの疑似コードは、おそらく次のようなものでしょう。実際には、トランザクションの種類（読取り専用、読取りと書込み）やニーズ（トランザクション必須、新規トランザクション必須など）、チェック例外と非チェック例外の扱いなどが異なるため、もう少し複雑なものになっているはずです。

```
public class TransactionInterceptor {

    UserTransaction transaction; // コンテナから注入

    // 疑似コード

    public void interceptTransaction(
        Method method, Object target, Object[] args) throws Exception {

        if (transaction.getStatus() != Status.NoTransaction) {
            transaction.begin();
        }
        try {
            method.invoke(target, args);
            transaction.commit();
        } catch (Exception ex) {
            transaction.rollback();
            throw ex;
        }
    }
}
```



は、従来のデコレータのように使われず、`JComponent`で定義されている次のメソッドを使っています。これは、いわば「プラグイン・デコレータ」とでも呼べるものです。

```
public void setBorder(Border);  
public Border getBorder();
```

このBorderクラスは、javax.swing.borderパッケージ内に存在します。Borderのインスタンスは、createEtchedBorder()やcreateTitledBorder()などのメソッドを呼び出すことで、javax.swing.BorderFactoryから取得します。

I/Oストリーム

Decorator パターンをよく見かけるとい方がいるとすれば、それもそのはずです。Javaが始まって以来、一般的な `java.io Stream` や `Reader`、`Writer` はこのパターンを使って動作しています。おそらく、次のようなコードは数え切れないほど見てきていることでしょう。

```
// IOStreamsDemo.javaより
BufferedReader is =
    new BufferedReader(new FileReader("some filename here"));
PrintWriter pout =
    new PrintWriter(new FileWriter("output filename here"));
LineNumberReader lnr =
    new LineNumberReader(new FileReader(foo.getFile()));
```

ここに個別のDecoratorクラスはありません。関係するクラスはすべて、ReaderやWriter（バイナリ・ファイルの場合はInputStreamやOutputStream）の直接のサブクラスであり、すべてのクラスに、引数としてトップレベルのクラス（もちろん、任意のサブクラスでも構いません）のインスタンスを受け取るコンストラクタが存在します。しかし、この使用法は、あるクラスが別のクラスをラップすることで機能を追加するという、Decoratorパターンの基本的な説明に一致しています。

