

IAN DARWIN

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

・ ンテリア・デコレーターとは、家にやってきて部屋の見た目や雰囲気をどのように変えるべきかを教えてくれる人を指します。たとえば、どんな家具や壁、床の敷物を追加すべきかについて、アドバイスしてくれます。ソフトウェアにおけるデコレータとは、ターゲット・クラスを「ラップ」することによって、そのクラスを変更せずに機能をコードに追加するものを指します。デコレータは、クラスの拡張に対してはオープンでありつつも変更に対してはクローズドという、オープン/クローズド原則を体現するものです。

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。しかし、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らかです。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たんに、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、Photolmage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 Photolmage image = new Photolmage(
    "Sunset at Tres Ríos", "2020/ifd12345.jpg");

// 米国のレターサイズの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 Photolmage("Goodbye at the Station",
          "1968/ifd.00042.jpg"))));
 addToPrintOrder(im2);
 // 電子的に画像を販売する
 Photolmage dig = new Digitallmage(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。
   PhotoImage (Componentに相当するもの)のコードを次に示します。
 /** PhotoImageは、ある時点で撮影した写真を表す。
 public class PhotoImage {
   /** 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 PhotoImage() {
     // 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 \( \) 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 Photolmage 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

Decoratorパターンをよく見かけるという方がいるとすれば、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。

(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、PhotolmageはComponentに、Printおよび Digitallmage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







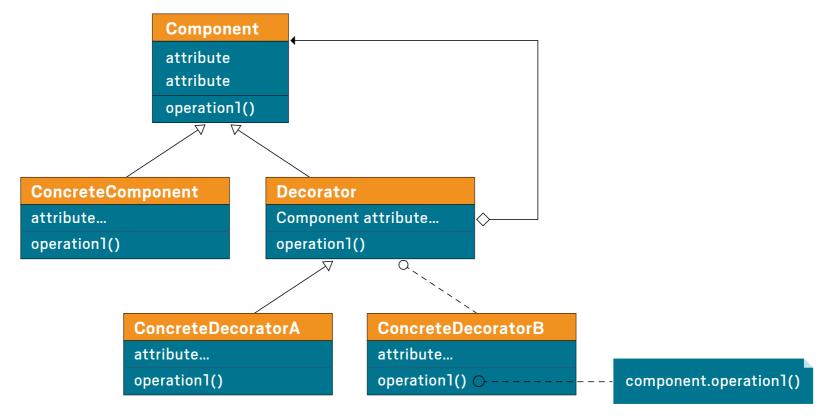


図1:デコレータの階層

クラスをConcreteComponentにするか、またはデコレータにするかは、ユースケース次第です。なお、デコレータ・クラスが1つしかなく、増える見込みもないような非常に単純なケースでは、Decoratorを別の抽象クラスとして定義せずに、実際のデコレータ内で直接委譲することもできる点に注意してください。クラスを分けるのは、複数のデコレータを想定している場合に、階層を作る場所として便利だというだけのことです。

すべてのコードは、GitHubのサブディレクトリsrc/main/java/structure/decoratorで公開しています。

フレームワークの使用

マネージドBeanの呼出しをインターセプトするための仕組みは、広く使用されているさまざまなフレームワークに搭載されています。たとえば、Java EEのインターセプタやSpringのアスペクト指向プログラミング (AOP) では、事前定義済みのインターセプタとともに、独自のインターセプタを定義できる機能も提供しています。これらはデコレータです。たとえば、いずれのフレームワークでもアノテーション・ベースのトランザクション管理を提供しており、次のような記述が可能です。







```
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;
```







```
}
```

EJB3インターセプタAPI (@Interceptorを参照) またはSpringのAOP (@Aroundを参照) を使うことで、独自のインターセプタを記述できます。

境界線デコレータ

かつて、Decoratorパターンは、グラフィック用のアドオンで多用されていました。その一例が、レイアウトの一部 として、テキスト領域やダイアログ・ボックスにきれいな境界線を追加することでした。デコレータは、Smalltalkや InterViewsなどの、初期における多くの有力なウィンドウ・システムでそのように使われていました。具体的には、次 の疑似コードのようにして使っていました(TextFieldコンストラクタは、引数として行および列を受け取ります)。

```
Component niceTextBox = new BorderDecorator(new TextField(1, 20));
Component niceTextArea = new BorderDecorator(new ScrollingDecorator(new TextField(5, 20)));
```

すべての委譲対象メソッドを転送する必要があるかどうかは、アプリケーションによって異なります。Decoratorの問題は、デコレーションされるコンポーネントに大量のメソッドがある場合(わかりやすくすることを意識しているのであれば、そうすべきではありません)、すべてのメソッドを転送しなければならなくなり、必要になる委譲メソッドの数が増えて、使いにくいパターンになってしまう可能性があります。しかし、継承した特定のメソッドを転送しなかった場合、継承の性質上、そのメソッドは(呼び出された場合)ターゲットではなくDecoratorのコンテキストだけで実行され、おかしなことが起きる場合があります。IDEを使ってすべての委譲を生成することもできますが(たとえばEclipseでは、Source → Generate Delegate Methodsオプションを使用)、メンテナンスしやすいコードにはなりません。このような場合は、StrategyパターンやProxyパターンを使って機能を追加するとよいでしょう。

JavaのSwing UIパッケージの作成者は、この問題に気づいていました。JComponent (Abstract Window ToolkitのComponentのサブクラス)をデコレーションしたくても、これにはおよそ120のパブリック・メソッドがあります。このためだけにサブクラスを作ることを避け、本来的な意味でデコレーションを行うことができるようにするために、別のアプローチが採用され、SwingのBorderオブジェクトが提供されました。このBorderオブジェクト







は、従来のデコレータのようには使われず、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。

1/0ストリーム

Decoratorパターンをよく見かけるという方がいるとすれば、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 Irdr =
  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、この使用方法は、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。







このシリーズのこれまでの記事:

Commandパターン
Stateパターン
Visitorパターン

Decorator & Proxy

他のクラスを拡張する別のパターンとして、Proxyパターンがあります。多くの場合、代理の対象となるオブジェクトと同じインタフェースが使用されます。その結果、ProxyパターンとDecoratorパターンが混同されることもあります。ただし、Decoratorはおもに、ターゲットに機能を追加するものです。Gang of Four (四人組) による定義では、Proxy はターゲットのアクセス制御を行うものとされています。この制御では、高価なオブジェクトの遅延作成、パーミッションやセキュリティ・チェックの観点の強制(セキュリティ・プロキシ)、ターゲット位置の隠蔽(RMI、リモートEJB 呼出し、JAX-RSクライアント、JAX-WSクライアントなどのRPCベースのネットワーク通信APIで使われるリモート・アクセス・プロキシ)を実現できます。

遅延作成を行う状況では、高価な(重量)オブジェクトと同じインタフェースで軽量プロキシが作成されますが、実際に設定されているフィールドはほとんどないか、あるいはまったくありません(IDとタイトルのみといったものでしょう)。このプロキシは、完全なオブジェクトを使用するメソッドが呼び出された場合に限り、重量オブジェクトの作成を制御します。

ネットワーク通信が行われる状況では、クライアントのコードはローカル・オブジェクトを呼び出しているよう に見えますが、実際にはすべて、ある程度透過的に、ネットワーク通信と、オブジェクトと転送可能なフォーマットと の相互変換を管理するプロキシ・オブジェクトを呼び出しています。 Decoratorパターンでデコレータの作成を担当 するのは、通常、クライアントです。多くの場合、デコレータはデコレーションの対象となるオブジェクトと同時に作成されます。

まとめ

デコレータは、ターゲット・クラスを変更せずに機能を追加する便利な方法です。通常、デコレータはターゲットと同じAPIを提供し、引数をターゲット・オブジェクトに転送する前または後に、追加で作業を行います。今度小さなクラス群に機能を追加する必要が生じた際は、ぜひDecorator/パターンを使ってみてください。</article>

Ian Darwin (@lan_Darwin):メインフレーム・アプリケーションから、UNIXおよびWindows向けのデスクトップ・パブリッシング・アプリケーション、Javaによるデスクトップ・データベース・アプリケーションやAndroid向けのヘルスケア・アプリまで、あらゆる開発を手がける。『Java Cookbook』、『Android Cookbook』(いずれもO'Reilly)の著者。また、Learning Tree Internationalでいくつかのコースを作成し、多くのコースで教えている。





