


## Decorator ～かぶせて機能UP

 絶版に伴い、校正前の原稿テキストを公開したものです。基本的に原稿そのままをHTML形式に変換したものですので、誤字/脱字、説明不足の箇所もあるかも知れませんがご了承ください。初出:「PHPによるデザインパターン入門」(下岡秀幸/畑勝也/道端良 著, 秀和システム, ISBN4-7980-1516-4, 2006年11月23日発売)

### GoF本における分類

構造+オブジェクト

#### はじめに

ここではDecoratorパターンについて説明します。

Decoratorとは「装飾者」という意味があります。そもそも装飾するというのは、基本となる物がありそこに様々な効果をもたらす要素を加えていく行為です。クリスマスツリーに飾りをつけるのもそうですし、アイスクリームにトッピングをすることもそうです。

Decoratorパターンも同様なイメージとして捉えることができます。つまり、基本となるものに対して様々な機能を一つひとつ加えていき、あたかも装飾するかのようなイメージです。

#### たとえば

世の中には様々なアプリケーションが存在しています。そのほとんどには画面があり、カスタマイズできるものが多いです。

たとえば、Webブラウザを考えてみましょう。Webページを表示した時、一画面に収まりきらない場合には横スクロールバーや縦スクロールバーが表示されます。また、縦スクロールバーもしくは横スクロールバーだけ表示される時もありますし、両方表示される場合もあります。

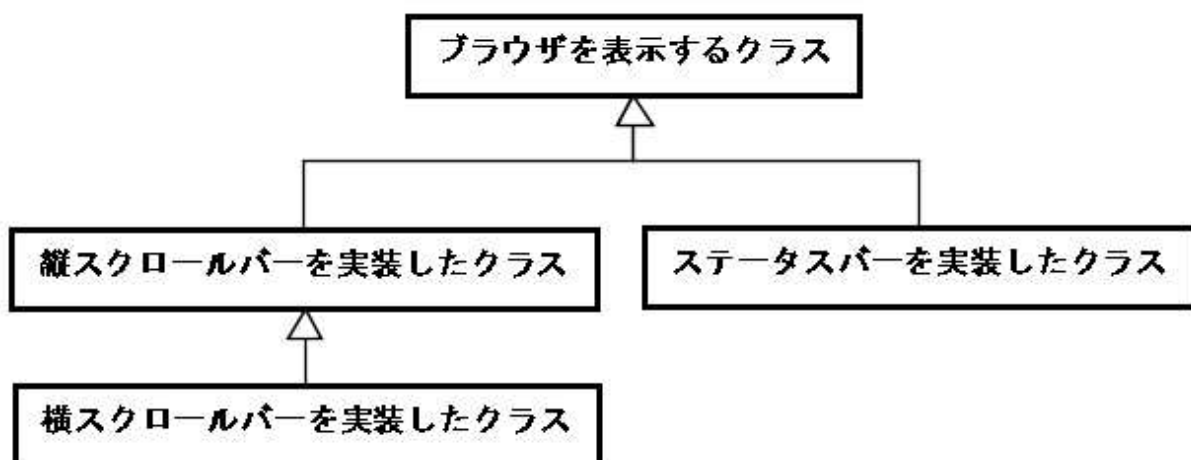
このスクロールバーは、「Webページを表示する」というWebブラウザの基本的な機能に「画面をスクロールさせる」という機能を追加したものと捉えることができます。その他にも、様々な効果をもたらす機能がブラウザにありますが、いずれもWebブラウザの機能を拡張したものと言えるでしょう。

さて、それぞれのクラスには様々な機能があります。つまり、「責任」を持っているということです。この責任を追加したい場合、つまり機能を拡張したい場合、どの様にすれば良いでしょうか？

とっさに思いつくのは、継承を使ってサブクラス化することです。継承を使うことで、親クラスの責任を受け継ぎつつ、新しい責任を追加することができます。しかし、この「継承」が時として不便になってくる場合があります。

先ほどのスクロールバーの例で考えてみましょう。たとえば、「横スクロールバーは実装しないが、縦スクロールバーは実装したい」や「さらにその逆の実装パターンも存在する」といった要求が出てきた場合、それぞれの機能を実装したクラスが必要になります。

これは**継承による機能拡張は「静的」な拡張**だからです。言い換えると、クラスのコードを作成した時点でその責任が決定されるということです。コードを書いた時点で機能が決まってしまうため、当然ですが**機能を組み合わせることが非常に難しく**なります。



つまり、すべてのパターンを網羅しようとする、作成すべきクラスの数が非常に多くなってしまいます。また、新しい機能を追加する場合も、すべてのパターンを網羅するために…これは非常に大変そうです。

また、「ユーザの操作によってスクロールバーを追加したり取り外したりしたい」といった要求もあるでしょう。継承を利用して機能を拡張した場合、このような要求に応えることも容易ではありません。

このような場面で、**Decoratorパターン**が有用になってきます。

## Decoratorパターンとは？

Decoratorパターンの目的は、GoF本では次のように定義されています。

オブジェクトに責任を動的に追加する。Decorator パターンは、サブクラス化よりも柔軟な拡張方法を提供する。

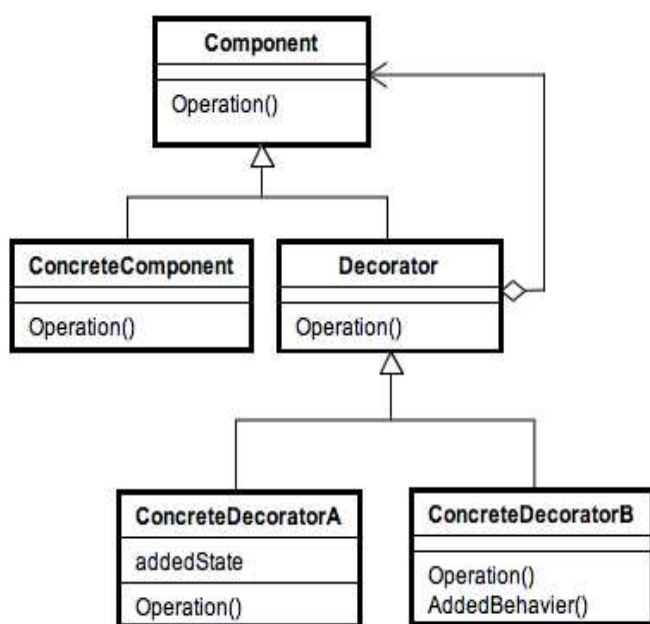
Decoratorパターンはオブジェクトの構造に注目したパターンで、オブジェクトに対して機能を柔軟に追加できるパターンです。

クラスの責任を拡張するにはサブクラス化、つまり継承によって実現しますが、継承による拡張は、責任を「静的」に追加することを意味します。

一方、Decoratorパターンでは責任を「動的」に追加することができます。つまり、実行中に責任を追加したり外したりできるということです。これにより、オブジェクトの柔軟な機能拡張が可能です。

## Decoratorパターンの構造

Decoratorパターンのクラス図と構成要素は次のようになります。



### Componentクラス

拡張される機能を定義した抽象クラスです。

### ConcreteComponentクラス

Componentクラスで定義した機能を基本実装する、飾り付けされる具象クラスです。

### Decoratorクラス

Componentを継承し、さらにメンバー変数としてComponentを保持する抽象クラスです。

自身のOperationメソッドではComponentのOperationを呼び出すようにします。つまりOperationメソッドの具体的な実装をComponentへ委譲します。

### ConcreteDecoratorAクラス、ConcreteDecoratorBクラス

Componentに機能を追加するためにDecoratorクラスを継承し、自身のOperationメソッドで親クラスのOperationメソッドを利用しながら、機能の拡張(飾り付け)を行います。

## Decoratorパターンのメリット

### 柔軟な拡張が可能

既存クラスのメソッドに機能を追加する場合、サブクラスを作成し、既存クラスの機能追加したいメソッドをオーバーライドする事が一般的かもしれませんが。つまり継承です。しかし、追加したい「機能」のパターンが複数ある場合、また、追加したい「機能」は他のパターンを踏まえた上で実装されることがある場合、さらには追加するパターンに「順序」が存在する場合、そのパターンの組

み合わせを考慮したクラス設計が必要になります。機能が静的に追加されている状態であり、必要なパターンの組み合わせの数分、サブクラスの作成が必要となります。

こういった場合に、Decoratorパターンを適応することで、機能を動的に追加できるようになり、オブジェクトを利用する側でパターンの選択ができるようになります。Decoratorパターンは、継承ではなく委譲を利用することにより、柔軟な機能の拡張が可能となっています。

機能の実装を階層構造の上位で定義しなくて済む

Decoratorパターンは機能の枠を上層で定義(インターフェースなど)し、それに対しての実装をDecoratorで定義できるため、クラス階層の上層で機能を定義する必要がなくなります。こうすることで、再度編集する際に上層のクラスに手を入れる必要がなくなり、メンテナンス性の向上に繋がります。

## Decoratorパターンの適用例

Decoratorパターンの適用例を見てみましょう。

ここでは、入力された文字列を加工し表示するアプリケーションにDecoratorパターンを適用してみます。文字列の加工には、以下のような種類を用意しました。

対応するクラス    機能の概要

UpperCaseText    入力された文字の半角小文字を半角大文字に変換する

DoubleByteText    入力された文字の半角文字を全角文字に変換する

まずは、Componentクラスに相当するTextインターフェースから見てみましょう。

このインターフェースは文字列の入力、出力のためのメソッドを定義したインターフェースです。

●Textインターフェース(Text.class.php)

```
<?php
/**
 * テキストを扱うインターフェースクラスです
 */
interface Text {
    public function getText();
    public function setText($str);
}
?>
```

続けて、Textインターフェースを実装したPlainTextクラスです。ConcreteComponentクラスに相当します。

このクラスは、加工する前の文字列を管理するためのクラスです。

●PlainTextクラス(PlainText.class.php)

```
<?php
require_once('Text.class.php');
?>
<?php
/**
 * 編集前のテキストを表すクラスです
 */
class PlainText implements Text {

    /**
     * インスタンスが保持する文字列です
     */
    private $textString = null;

    /**
     * インスタンスが保持する文字列を返します
     */
    public function getText() {
        return $this->textString;
    }

    /**
```

```

    * インスタンスに文字列をセットします
    */
    public function setText($str) {
        $this->textString = $str;
    }
}
?>

```

次はDecoratorクラスに相当するTextDecoratorクラスです。

TextDecoratorクラスは、Textインターフェースを継承しさらに、Text型のインスタンスを変数として保持しています。そして、getTextメソッドやsetTextメソッドでは保持しているText型のインスタンスを実行しています。

●TextDecoratorクラス(TextDecorator.class.php)

```

<?php
require_once('Text.class.php');
?>
<?php
/**
 * Textクラスを修飾するDecoratorです
 */
abstract class TextDecorator implements Text {

    /**
     * Text型の変数です
     */
    private $text;

    /**
     * インスタンスを生成します
     */
    public function __construct(Text $target) {
        $this->text = $target;
    }

    /**
     * インスタンスが保持する文字列を返します
     */
    public function getText() {
        return $this->text->getText();
    }

    /**
     * インスタンスに文字列をセットします
     */
    public function setText($str) {
        $this->text->setText($str);
    }
}
?>

```

そして、次は実際の「decorate」をおこなうクラス群です。

ConcreteDecoratorクラスに相当するUpperCaseTextクラスとDoubleByteTextクラスは、TextDecoratorを継承するクラスです。

これらのクラスのgetTextメソッドでは、内部に保持したText型オブジェクトのgetTextメソッドを実行します。そして、その戻り値に自分自身が担当する処理を施し結果を返します。

UpperCaseTextクラスでは半角小文字を半角大文字に変換、DoubleByteTextクラスでは半角文字を全角文字に変換する処理を、それぞれおこないます。

●UpperCaseTextクラス(UpperCaseText.class.php)

```

<?php
require_once('TextDecorator.class.php');
?>
<?php
/**

```

```

* TextDecoratorクラスの実装クラスです
*/
class UpperCaseText extends TextDecorator {

    /**
     * インスタンスを生成します
     */
    public function __construct(Text $target) {
        parent::__construct($target);
    }

    /**
     * 半角小文字を半角大文字に変換して返します
     */
    public function getText() {
        $str = parent::getText();
        $str = strtoupper($str);
        return $str;
    }
}
?>

```

#### ●DoubleByteTextクラス(DoubleByteText.class.php)

```

<?php
require_once('TextDecorator.class.php');
?>
<?php
/**
 * TextDecoratorクラスの実装クラスです
 */
class DoubleByteText extends TextDecorator {

    /**
     * インスタンスを生成します
     */
    public function __construct(Text $target) {
        parent::__construct($target);
    }

    /**
     * テキストを全角文字に変換して返します
     * 半角英字、数字、スペース、カタカナを全角に、
     * 濁点付きの文字を一文字に変換します
     */
    public function getText() {
        $str = parent::getText();
        $str = mb_convert_kana($str, "RANSKV");
        return $str;
    }
}
?>

```

最後にクライアント側のコードです。簡単に動作を確認できるよう、入力用のHTMLフォームも表示します。

入力用のHTMLフォームにはテキストボックスと2つのチェックボックスがあります。テキストボックスには任意の文字列を入力します。チェックボックスは、入力した文字列に対する装飾が選べるようになっています。

#### ●クライアント側コード(decorator\_client.php)

```

<?php
require_once('UpperCaseText.class.php');
require_once('DoubleByteText.class.php');
require_once('PlainText.class.php');
?>
<?php
$text = (isset($_POST['text'])? $_POST['text'] : '');
$decorate = (isset($_POST['decorate'])? $_POST['decorate'] : array());
if ($text != '') {

```

```

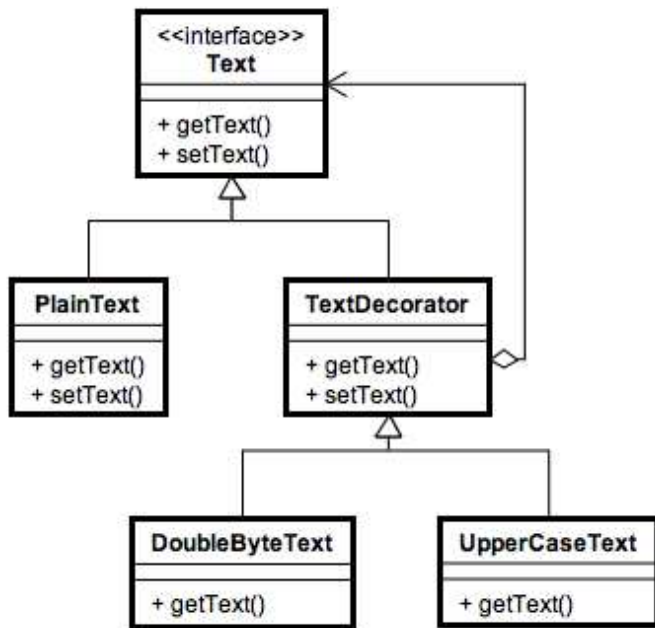
$text_object = new PlainText();
$text_object->setText($text);

foreach ($decorate as $val) {
    switch ($val) {
        case 'double':
            $text_object = new DoubleByteText($text_object);
            break;
        case 'upper':
            $text_object = new UpperCaseText($text_object);
            break;
    }
}
echo $text_object->getText() . "<br>";
}

?>
<hr>
<form action="" method="post">
テキスト: <input type="text" name="text"><br>
装飾: <input type="checkbox" name="decorate[]" value="upper">大文字に変換
      <input type="checkbox" name="decorate[]" value="double">2バイト文字に変換
      <input type="submit">
</form>

```

それでは、このサンプルのクラス図を確認しておきましょう。



## Decoratorパターンのオブジェクト指向的要素

Decoratorパターンは「**ポリモーフィズム**」を非常に活用したパターンです。

Decoratorパターンは、責任を「追加する側」のDecoratorクラスが「追加される側」のComponentクラスを包み込んで機能を拡張します。この時、「追加する側」は「追加される側」と同じAPIを持っています。なぜなら、両者ともこの共通APIを定義したComponentクラスを実装しているためです。つまり、Componentクラスの利用側から見ると、責任を「追加する側」と「追加される側」を同一視することができます。言い換えると、責任を「追加する側」のクラスがあろうとなかろうと、また「追加する側」のクラスが具体的にどの**ConcreteDecorator**クラスかを意識することなく、「追加される側」のクラスを利用できる、ということです。

また、Decoratorクラスは、Component型のオブジェクトを内部に保持しています。このオブジェクトは「Component型である」というだけで、具体的にどのクラスのインスタンスなのかは分かりません。逆に言うと、Component型のオブジェクトであれば、分け隔てなく扱うことができることを意味しています。ここでも**具体的なクラスに依存するのではなく、そのインターフェースに依存する**構造が利用されています。

## 関連するパターン

### Adapter パターン

AdapterパターンもDecoratorパターンと同様、オブジェクトを包み込むパターンです。

Decoratorパターンはオブジェクトを包み込むことで、オブジェクトの責任を変化させるパターンです。一方のAdapterパターンは、オブジェクトを包み込んでそのAPIを変化させるパターンです。

## Composite パターン

CompositeパターンはDecoratorパターンの構造と非常に良く似ており、オブジェクトの集約を目的としたパターンです。

## Strategy パターン

Strategyパターンもオブジェクトの責任を変えるためのパターンです。

Decoratorパターンはオブジェクトの責任を変えるために「外側」からアプローチしますが、Strategyパターンは「内側」をごっそり変えるアプローチを採ります。

Componentクラスが大きく、Decoratorパターンを適用した場合にコストがかかりすぎる場合に使用される場合があります。

## まとめ

---

ここではオブジェクトの責任を外側から変えるDecoratorパターンについて説明しました。