


TemplateMethodパターン ～処理を穴埋めする

 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

振る舞い+クラス

はじめに

ここではTemplate Methodパターンについて説明します。

templateという単語には、「型板(かたいた)」や「鋳型」といった意味があります。いずれも「型」が定義されていて、できあがりの形は同じになります。しかし、何を使って作るかによって、具体的にどのようなものになるが変わります。

Template Methodパターンも「型」となるクラスが登場するパターンです。

では、早速見ていきましょう。

たとえば

プログラムを作っていると、似たような処理がたくさんできることがあります。オブジェクト指向プログラミングをおこなった場合も同様に、似たような処理をおこなうクラスやメソッドができることがあります。

完全に同じ処理内容であれば、親クラスにその処理を移したりクラスメソッドとして別クラスを作成したりするでしょう。

しかし、**処理フローはほとんど同じだがごく一部だけ処理内容が異なる**、という場合はどうでしょうか？

ここで安易にコピー＆ペーストして新しいクラスを作ってしまうと、その処理にバグがあった場合、コピー＆ペーストして作ったすべてのクラスを修正することになってしまいます。

その異なる処理の部分だけを分けておき、クラスごとにその処理を実装すれば良いようになれば理想的ですね。

これを実現するパターンが、**Template Methodパターン**です。

Template Methodパターンとは？

Template Method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。

1つのオペレーションにアルゴリズムのスケルトンを定義しておき、その中のいくつかのステップについては、サブクラスでの定義に任せることにする。
Template Methodパターンでは、アルゴリズムの構造を変えずに、アルゴリズム中のあるステップをサブクラスで定義する。

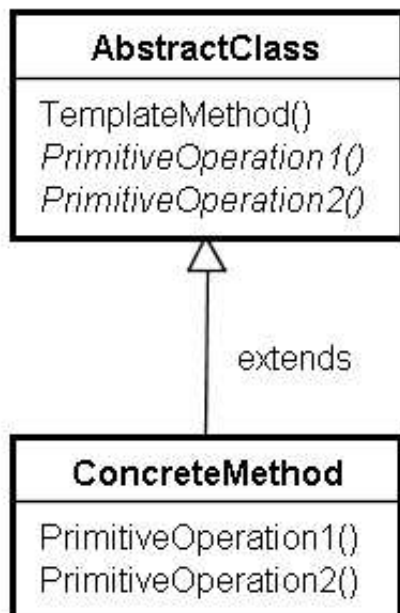
Template Methodパターンは、親クラスで処理の大きな枠組みを定義し、具体的な処理内容をサブクラスで決定させるためのパターンです。ひょっとすると、みなさんも知らず知らずのうちに使っているパターンかも知れません。

Template MethodパターンはGoFパターンの中でも最も基本的なパターンの1つで、他のパターンでも多く利用されています。

一見すると普通の継承と変わりませんが、「処理の一部をサブクラスで実装する」というのがポイントです。

Template Methodパターンの構造

Template Methodパターンのクラス図と構成要素は、次のとおりです。



Template Methodパターンの構成要素

AbstractClassクラス

処理の大きな枠組みを定義するクラスです。その枠組みを定義するメソッド(template methodと呼ばれます)と、枠組みから利用される抽象メソッドを含んでいます。抽象メソッドの具体的な処理内容は、サブクラスで実装します。なお、慣習上、抽象メソッドのようにサブクラスで実装することが期待されるメソッドの名前を、「do～」にすることが多いようです。

ConcreteClassクラス

AbstractClassクラスを継承したサブクラスです。AbstractClassクラスで定義された抽象メソッドを実装し、具体的な処理内容を決定します。このメソッドは、AbstractClassクラスで枠組みを定義したメソッドから呼び出されます。

Template Methodパターンのメリット

Template Methodパターンのメリットとしては、以下のものが挙げられます。

共通な処理をまとめることができる

それぞれのサブクラスに共通な処理を、親クラスにまとめることができます。これにより、サブクラス側に共通処理部分を記述する必要がなくなり、変更が発生した場合もAbstractClassクラス側を修正するだけで済みます。

サブクラスにより、具体的な処理内容を変えることができる

親クラスで定義された抽象メソッドは、サブクラス側で具体的に実装されます。つまり、処理の大きな枠組みを変更することなく、サブクラスによって具体的な処理内容を変えることができます。

Template Methodパターンの適用例

それではTemplate Methodパターンの適用例を見てみましょう。

ここでは渡されたデータを成形して表示する簡単なアプリケーションにTemplate Methodパターンを適用した例を示します。

まずは、処理の「型」を定義するAbstractDisplayクラスから見ていきます。AbstractClassクラスに相当するクラスです。

このクラスはコンストラクタの引数として表示するデータを受け取り、displayメソッドでそのデータを成形してヘッダ、ボディ、フッタの順に表示するようになっています。この処理の流れは定義されていますが、具体的な実装はそれぞれ抽象メソッドになっています。この抽象メソッドをサブクラスで実装することで、異なる成形処理をおこなうことができます。

●AbstractDisplayクラス (AbstractDisplay.class.php)

```
<?php
/**
 * AbstractClassクラスに相当する
 */
abstract class AbstractDisplay {

    /**
     * 表示するデータ
     */
    private $data;

    /**
     * コンストラクタ
     * @param mixed 表示するデータ
     */
    public function __construct($data) {
        if (!is_array($data)) {
            $data = array($data);
        }
        $this->data = $data;
    }

    /**
     * template methodに相当する
     */
    public function display() {
        $this->displayHeader();
        $this->displayBody();
        $this->displayFooter();
    }

    /**
     * データを取得する
     */
    public function getData() {
        return $this->data;
    }

    /**
     * ヘッダを表示する
     * サブクラスに実装を任せる抽象メソッド
     */
    protected abstract function displayHeader();

    /**
     * ボディ（クライアントから渡された内容）を表示する
     * サブクラスに実装を任せる抽象メソッド
     */
    protected abstract function displayBody();

    /**
     * フッタを表示する
     * サブクラスに実装を任せる抽象メソッド
     */
    protected abstract function displayFooter();
}
?>
```

次にAbstractDisplayクラスを継承するConcreteClassクラスに相当するクラスを見てみましょう。

ここでは2つのクラスを用意しています。ListDisplayクラスは一覧形式、TableDisplayクラスは表形式でそれぞれデータを表示するクラスです。両クラスともAbstractDisplayクラスで定義された抽象メソッドを実装していますが、実装内容を変えることでクラスごとの動作を変えています。

●ListDisplayクラス (ListDisplay.class.php)

```

<?php
require_once 'AbstractDisplay.class.php';
?>
<?php
/**
 * ConcreteClassクラスに相当する
 */
class ListDisplay extends AbstractDisplay {

    /**
     * ヘッダを表示する
     */
    protected function displayHeader() {
        echo '<dl>';
    }

    /**
     * ボディ（クライアントから渡された内容）を表示する
     */
    protected function displayBody() {
        foreach ($this->getData() as $key => $value) {
            echo '<dt>Item ' . $key . '</dt>';
            echo '<dd>' . $value . '</dd>';
        }
    }

    /**
     * フッタを表示する
     */
    protected function displayFooter() {
        echo '</dl>';
    }
}
?>

```

●TableDisplayクラス(TableDisplay.class.php)

```

<?php
require_once 'AbstractDisplay.class.php';
?>
<?php
/**
 * ConcreteClassクラスに相当する
 */
class TableDisplay extends AbstractDisplay {

    /**
     * ヘッダを表示する
     */
    protected function displayHeader() {
        echo '<table border="1" cellpadding="2" cellspacing="2">';
    }

    /**
     * ボディ（クライアントから渡された内容）を表示する
     */
    protected function displayBody() {
        foreach ($this->getData() as $key => $value) {
            echo '<tr>';
            echo '<th>' . $key . '</th>';
            echo '<td>' . $value . '</td>';
            echo '</tr>';
        }
    }
}

```

```
    }

    /**
     * フッタを表示する
     */
    protected function displayFooter() {
        echo '</table>';
    }
}
?>
```

最後にクライアント側のコードです。

まず、表示するデータを用意し、ListDisplayクラスとTableDisplayクラスに渡してそれぞれ表示しています。表示の手順は全く同じですが、データの表示形式がConcreteClassクラスによって変わっていることが分かります。

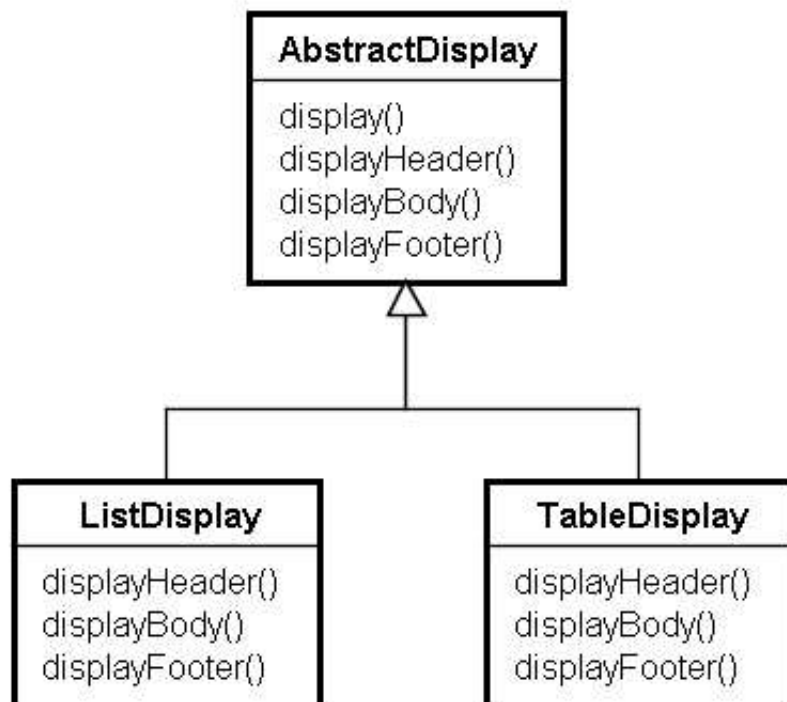
●クライアント側コード(template_method_client.php)

```
<?php
require_once 'ListDisplay.class.php';
require_once 'TableDisplay.class.php';
?>
<?php
    $data = array('Design Pattern',
                  'Gang of Four',
                  'Template Method Sample1',
                  'Template Method Sample2');

    $display1 = new ListDisplay($data);
    $display2 = new TableDisplay($data);

    $display1->display();
    echo '<hr>';
    $display2->display();
?>
```

このサンプルコードのクラス図は、次のようになります。



Template Methodパターンのオブジェクト指向的要素

Template Methodパターンは「**継承**」を利用しているパターンです。

親クラス側では、具体的な処理内容が変わる部分を「**わざと**」抽象メソッドとして定義します。この抽象メソッドは、サブクラスで必ず具体的な実装を行う必要があります。つまり、抽象メソッドを実装しなければ、親クラスが提供している機能を使うことはできません。このため、親クラスの視点から見ると、**抽象メソッドとして定義したメソッドの実装が「保証される」**ことになります。

一方、サブクラス側では、サブクラスによって実装を変えることができます。それぞれのサブクラスは同じ親クラスを継承していますので、すべてのサブクラスの型は、親クラスの型と見ることができます。しかし、PHP5ではJavaとは異なり、型の概念はタイプヒントング (Type Hinting) 以外は皆無ですが、親クラスで提供されているメソッドだけを使用することで、サブクラスを親クラスの型と見ることができます。このようにプログラミングすることで、サブクラスを差し替えても正しく動作するようになります。この「**サブクラスの型はその親クラスの型と置換可能**」ということは、継承に関する一般的な原則であり、リスコフの置換原則 (LSP: The Liskov Substitution Principle) と呼ばれます。これにより、クライアント側のコードを変更することなく、利用する処理を切り替える、といったことができるようになります。

また、Template Methodパターンは、親クラスがサブクラスのメソッドを呼び出すという「制御構造の反転」という特長を持つパターンでもあります。通常はその逆で、サブクラスが親クラスのメソッドを呼び出すことが多いと思います。このため、ハリウッドの原則 (The Hollywood Principle: 「Don't call us. We'll call you.」 「我々を呼び出すな。必要なときは、我々が君を呼び出す」) と呼ばれる原則の説明に利用されることがあります。ハリウッドの原則は、フレームワークの性質を端的に表すために良く用いられます。

関連するパターン

Factory Methodパターン

Template Methodパターンの典型的な応用例がFactory Methodパターンです。インスタンスの生成部分に使用されています。

Strategyパターン

Template Methodパターンでは、継承を使って処理内容の一部を切り替えます。一方のStrategyパターンでは、委譲を使って処理内容全体を切り替えます。

まとめ

ここではアルゴリズムの一部の動作をサブクラスによって決定するTemplate Methodパターンを説明しました。