Prototype ～コピーして作る

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

生成＋オブジェクト

はじめに

ここではPrototypeパターンについて説明します。

prototypeとはカタカナ言葉の「プロトタイプ」で、「試作品」「原型」といった意味があります。

それでは、Prototypeパターンはそのプロトタイプを使って何をするパターンなのでしょうか？早速見ていきましょう。

たとえば

ユーザーの操作を保存したオブジェクトを考えてみましょう。たとえば、アプリケーション上でのユーザの行動履歴を保持する監査クラスです。

このオブジェクトには、ユーザがいつどの画面にアクセスし、どういう操作をおこなったか、という情報が保存されています。通常、ユーザーの操作は決まった手順に沿ったものにはなりません。つまり、対象のユーザによって、オブジェクトが保持する情報はバラバラになるということです。

ここで、何らかの理由でこのオブジェクトと同じ内容のオブジェクトを生成する必要が出てきたとします。この場合、どのようにオブジェクトを生成すればよいでしょうか？

通常、オブジェクトは**new**演算子を使って生成します。しかし、この監査オブジェクトのように、内部状態が決まっていないオブジェクト**1**から作り上げるのは難しそうです。内部状態をパラメータを渡して設定するとしても、非常に面倒な作業になりそうです。

この場合、対象のオブジェクトそのものをコピーして、もう1つのインスタンスを生成した方が簡単そうです。

ここで説明する**Prototype**パターンは、原型となるインスタンスをコピーして新しいインスタンスを生成するためのパターンです。

Prototypeパターンとは？

Prototype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。

> 生成すべきオブジェクトの種類を原型となるインスタンスを使って明確にし、
> それをコピーすることで新たなオブジェクトの生成を行う。

Prototypeパターンは、オブジェクトの生成方法に注目したパターンです。

Prototypeパターンでは、親クラスでインスタンスをコピーするためのメソッドを定義します。このメソッドの戻り値は、自分自身のクラス型となります。そして、そのサブクラスで自分自身のコピーを返すよう実装します。

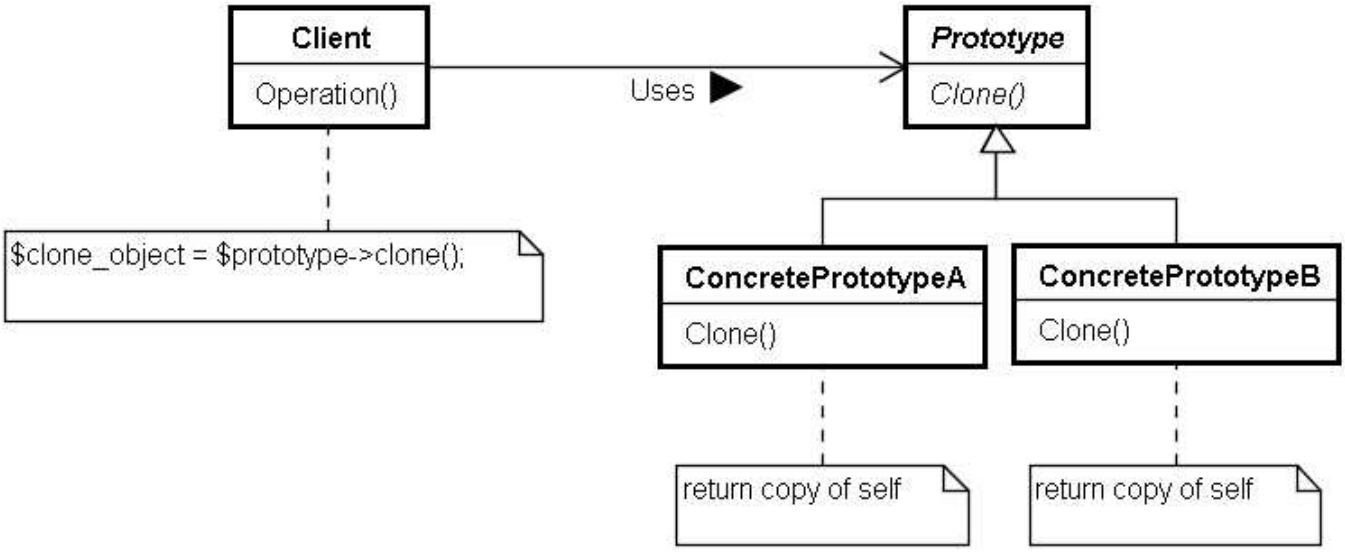
なお、Prototypeパターンを適用する場合、「浅いコピー」と「深いコピー」を意識して実装する必要がありますので、注意が必要です。

PHPでは、バージョン5.0.0から**clone**キーワードが追加されました。このcloneキーワードを使うことで、オブジェクトのコピーを作成できるようになりました。このcloneキーワードを使ったコピーは、「浅いコピー」(shallow copy)と呼ばれます。「浅いコピー」では、変数に格納された値がそのままコピーされます。具体的には、数値や文字列などは値として格納されているので、値そのものがコピーされます。これは期待する動作です。一方、オブジェクトの参照を格納した変数では、参照がコピーされることになります。この結果、コピー元とコピー先で、共通のオブジェクトを参照することになります。つまり、「外側の器はコピーされたが、中身はコピーされていない」状態になります。これは期待する動作ではありません。つまり、オブジェクトをコピーして、別のオブジェクトとなったはずが、コピー元のオブジェクトを変更するとコピー先のオブジェクトまで変更されてしまう、といった問題が発生します。

これを回避するには、「深いコピー」(deep copy)と呼ばれる方法でコピーする必要があります。「深いコピー」とは、内部の参照もコピーする方法です。内部の参照をコピーするような実装は、Prototypeパターンの親クラスで定義されたメソッド内、もしくは、**__clone**メソッドで行います。__cloneメソッドは、cloneキーワードと共にPHP5.0.0で追加されたメソッドです。

Prototypeパターンの構造

Prototypeパターンのクラス図と構成要素は、次のとおりです。



Prototypeクラス

コピーするためのメソッドを定義する親クラスです。

ConcretePrototypeクラス

Prototypeクラスのサブクラスで、Productクラスで定義されたコピー用のメソッドを実装するクラスです。

Clientクラス

Prototype型のオブジェクトを利用して、新しいインスタンスを生成します。

Prototypeパターンのメリット

Prototype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。

オブジェクトの生成処理を隠蔽できる

Prototypeパターンでは、利用側に対して実際にインスタンス化を行う具体的なクラスを隠蔽します。つまり、クライアントが意識しておく必要があるクラスの数を抑えることができます。

意識しなければならないクラスの数が増えるということは、クラスどうしの関係が強くなり、再利用することが難しくなります。Prototypeパターンでは、クラスどうしの関係を緩くする効果があると言えます。

クラスからインスタンスの生成が難しい場合に適用できる

たとえば、ユーザーの操作を保存したオブジェクトがある場合を考えてみましょう。

通常、ユーザーの操作は、決まった手順に沿ったものにはなりません。このため、同じ内容を持つインスタンスがもう一つ必要になった場合、クラスをインスタンス化するのは非常に困難です。この場合、インスタンスをコピーして、もう一つのインスタンスを生成した方が簡単です。

Prototypeパターンを適用することで、このような複雑なインスタンスを生成できます。

サブクラス化を減らすことができる

インスタンスを生成するパターンの代表例として、Factory Methodパターンがあります。Factory Methodパターンでは、生成する側（Creator）と生成される側（Product）それぞれのクラス階層で継承関係を持ちます。また、インスタンスを生成する専用のメソッド（factory method）を用意します。

Prototypeパターンでは、生成する側のクラス階層は必要ありません。インスタンスを生成する専用メソッドを呼び出す代わりに、生成される側のコピー用メソッドを呼び出すだけです。

Prototypeパターンの適用例

Prototypeパターンの適用例を見ていきます。

ここでは、商品クラスをインスタンス化しコピーした結果を比較する簡単なアプリケーションを用意しました。このサンプルは、深いコピーと浅いコピーの例を確認できるようになっていますので、そこに注目しながらコードを見ていきましょう。

まずは、コピーするためのメソッドを定義しているItemPrototypeクラスです。Prototypeクラスに相当します。

●ItemPrototypeクラス（ItemPrototype.class.php）

```
<?php
/**
```

```
 * Prototypeクラスに相当する
 */
abstract class ItemPrototype {
    private $item_code;
    private $item_name;
    private $price;
    private $detail;

    public function __construct($code, $name, $price) {
        $this->item_code = $code;
        $this->item_name = $name;
        $this->price = $price;
    }

    public function getCode() {
        return $this->item_code;
    }

    public function getName() {
        return $this->item_name;
    }

    public function getPrice() {
        return $this->price;
    }

    public function setDetail(stdClass $detail) {
        $this->detail = $detail;
    }

    public function getDetail() {
        return $this->detail;
    }

    public function dumpData() {
        echo '<dl>';
        echo '<dt>' . $this->getName() . '</dt>';
        echo '<dd>商品番号：' . $this->getCode() . '</dd>';
        echo '<dd>¥¥' . number_format($this->getPrice()) . '-</dd>';
        echo '<dd>' . $this->detail->comment . '</dd>';
        echo '</dl>';
    }

    /**
     * cloneキーワードを使って新しいインスタンスを作成する
     */
    public function newInstance() {
        $new_instance = clone $this;
        return $new_instance;
    }

    /**
     * protectedメソッドにする事で、外部から直接cloneされない
     * ようにしている
     */
    protected abstract function __clone();
}
?>
```

このクラスで注目するのは、newInstanceメソッドと__cloneメソッドです。newInstanceメソッドでは、cloneキーワードを使って自分自身のコピーを作っています。__cloneメソッドは抽象メソッドとなっており、サブクラスで異なる実装をおこなっています。

また、内部にstdClass型の詳細情報オブジェクトを保持するようになっていますが、これについては、次のDeepCopyItemクラスとShallowCopyItemクラスで説明します。

続いて、ConcretePrototypeクラスに相当するDeepCopyItemクラスとShallowCopyItemクラスです。名前の通り、DeepCopyItemクラスは深いコピー、ShallowCopyItemクラスは浅いコピーをそれぞれおこなう商品クラスです。

●DeepCopyItemクラス(DeepCopyItem.class.php)

```
<?php
require_once 'ItemPrototype.class.php';
?>
<?php
/**
 * ConcretePrototypeクラスに相当する
 */
class DeepCopyItem extends ItemPrototype {
    /**
```

```
     * 深いコピーを行うための実装
     * 内部で保持しているオブジェクトもコピー
     */
    protected function __clone() {
        $this->setDetail(clone $this->getDetail());
    }

}
?>
```

●ShallowCopyItemクラス(ShallowCopyItem.class.php)

```
<?php
require_once 'ItemPrototype.class.php';
?>
<?php
/**
 * ConcretePrototypeクラスに相当する
 */
class ShallowCopyItem extends ItemPrototype {

    /**
     * 浅いコピーを行うので、空の実装を行う
     */
    protected function __clone() {
    }
}
?>
```

先ほど出てきた__cloneメソッドを実装しているクラスですが、両クラスで実装内容が異なります。DeepCopyItemクラスでは、内部に保持したstdClass型の詳細情報オブジェクトをコピーしています。つまり、DeepCopyItemクラスをコピーしたときに、内部の詳細情報オブジェクトも併せてコピーされるということです。一方のShallowCopyItemクラスでは、__cloneメソッドは空の実装がなされています。これは、コピーしたときに詳細情報オブジェクトはコピーされないということですが、この違いがどう現れるのでしょうか。コードを一通り説明したあとに動作結果を見てみることにして、次に行きたいと思います。

次は、商品オブジェクトを管理するItemManagerクラスです。Clientクラスに相当します。

このクラスはコピーするオブジェクトを管理しつつ、新しいインスタンスを要求されたときにオブジェクトをコピーする役割を担っています。createメソッドを確認してください。

●ItemManagerクラス(ItemManager.class.php)

```
<?php
require_once 'ItemPrototype.class.php';
?>
<?php
/**
 * Clientクラスに相当する
 * このクラスからはConcretePrototypeクラスは見えていない
 */
class ItemManager {
    private $items;

    public function __construct() {
        $this->items = array();
    }

    public function registItem(ItemPrototype $item) {
        $this->items[$item->getCode()] = $item;
    }

    /**
     * Prototypeクラスのメソッドを使って、新しいインスタンスを作成
     */
    public function create($item_code) {
        if (!array_key_exists($item_code, $this->items)) {
            throw new Exception('item_code [' . $item_code . '] not exists !');
        }
        $cloned_item = $this->items[$item_code]->newInstance();

        return $cloned_item;
    }
}
?>
```

ここまで見てきたクラス群を利用するクライアント側のコードも見ておきましょう。

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

```php
<?php
require_once 'ItemManager.class.php';
require_once 'DeepCopyItem.class.php';
require_once 'ShallowCopyItem.class.php';

function testCopy(ItemManager $manager, $item_code) {
    /**
     * 商品のインスタンスを2つ作成
     */
    $item1 = $manager->create($item_code);
    $item2 = $manager->create($item_code);

    /**
     * 1つだけコメントを削除
     */
    $item2->getDetail()->comment = 'コメントを書き換えました';

    /**
     * 商品情報を表示
     * 深いコピーをした場合、$item2への変更は$item1に影響しない
     */
    echo '■オリジナル';
    $item1->dumpData();
    echo '■コピー';
    $item2->dumpData();
    echo '<hr>';
}
?>
<?php
    $manager = new ItemManager();

    /**
     * 商品データを登録
     */
    $item = new DeepCopyItem('ABC0001', '限定Tシャツ', 3800);
    $detail = new stdClass();
    $detail->comment = '商品Aのコメントです';
    $item->setDetail($detail);
    $manager->registItem($item);

    $item = new ShallowCopyItem('ABC0002', 'ぬいぐるみ', 1500);
    $detail = new stdClass();
    $detail->comment = '商品Bのコメントです';
    $item->setDetail($detail);
    $manager->registItem($item);

    testCopy($manager, 'ABC0001');
    testCopy($manager, 'ABC0002');
?>
```

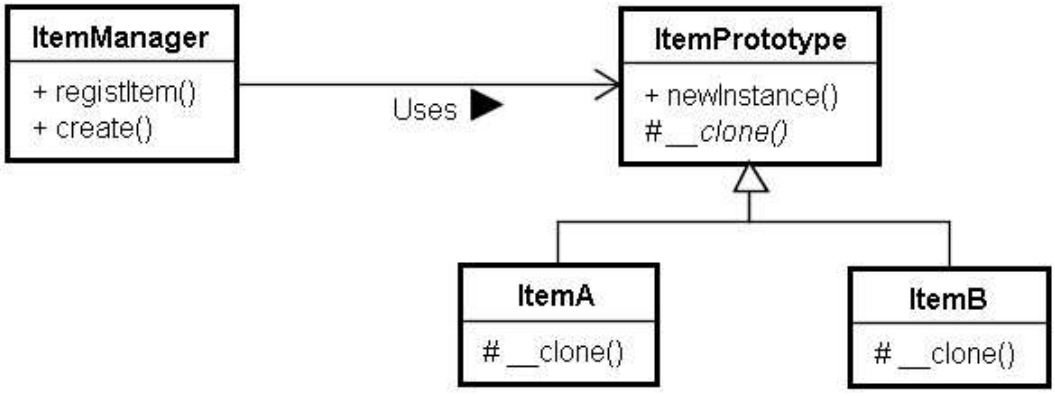
ここでは、それぞれの商品クラスをインスタンス化してItemManagerオブジェクトに登録し、そこから商品オブジェクトのコピーを2つ作成しています。そして、その片方に変更を加えたあと、オブジェクトの内容を表示しています。

早速、このコードの実行結果を見てみましょう。

深いコピーをおこなうDeepCopyItemクラスの場合、どちらかのオブジェクトを変更しても他方のオブジェクトに影響を与えていませんが、浅いコピーをおこなうShallowCopyItemクラスの場合、一方のオブジェクトが他方の影響を与えてしまっています。違いがお分かりでしょうか？

このように、Prototypeパターンを適用する場合は、深いコピーなのか浅いコピーなのかを必ず意識する必要があります。

最後に、Prototypeパターンを適用したサンプルのクラス図を確認しておきましょう。



Prototypeパターンのオブジェクト指向的要素

Prototypeパターンは「継承」と「ポリモーフィズム」を利用しているパターンです。

PrototypeクラスとConcretePrototypeクラスの間には継承関係があります。親クラスであるPrototypeクラスでは、インスタンスを生成するためのメソッドを用意します。Prototypeは抽象クラス、もしくはインターフェースで実装されます。親クラスを継承、もしくは実装したクラスがConcretePrototypeクラスです。ConcretePrototypeクラスでは、インスタンスを生成する処理を具体的に実装します。

Clientクラスでは、Prototypeクラスで提供されているAPIのみを使って、プログラミングを行います。具体的なConcretePrototypeクラスに関することを一切書かないわけです。こうすることで、ConcretePrototypeクラスに依存しないコードになり、Clientクラス側とPrototypeクラス側は独立して修正を行うことができます。

また、ClientクラスとConcretePrototypeクラスに依存関係がないお陰で、Clientクラスを修正することなく、ConcretePrototypeクラスを差し替えたり、新しく追加したりできます。

関連するパターン

Abstract Factoryパターン

Abstract FactoryパターンとPrototypeパターンが併用される場合があります。つまり、Abstract Factoryから返すインスタンスを、インスタンス化するのではなく、コピーすることで生成するのです。

Compositeパターン、Decoratorパターン

CompositeパターンやDecoratorパターンを使って、複雑な構造を持つオブジェクトを動的に生成する場面があります。こういった場合、Prototypeパターンが有効に使える場合があります。

まとめ

ここではインスタンスをコピーして新しいインスタンスを生成するPrototypeパターンについて説明しました。