Composite ～木構造を表す

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Compositeパターンについて見ていきましょう。

「composite」とは「合成物」「混合物」という意味を持ちます。ということは、Compositeパターンは、何かを混ぜるためのパターンなのでしょうか？

Compositeパターンは、単一のオブジェクトとその集合のどちらも同じように扱えるようにするためのパターンです。つまり、「単一のオブジェクト」と「オブジェクトの集合」を混ぜて、アクセス方法を同じにしてしまうパターンです。

分かるような分からないような、不思議なパターンですね。では、早速見ていきましょう。

たとえば

ファイルシステムのディレクトリツリーを考えてみましょう。

Windowsであれば、エクスプローラでツリー状に連なったフォルダを表示できますね。このディレクトリツリーにはフォルダやファイルが含まれていますが、フォルダやファイルに対する新規作成や削除、コピーといった操作は共通です。エクスプローラを使っているときに、「これはフォルダだから、こうやって削除しよう」とか「これはファイルだからこうやってコピーしよう」というように意識しないで操作しているはずです。

また、フォルダはその下にフォルダを含む場合がありますね。場合によっては、フォルダの階層が何階層にもなることもあるでしょう。また、ファイルはフォルダに含まれている、とも言えるでしょう。しかし、これら場合も特に意識しないでコピーや削除といった操作をおこなうことができます。

これをオブジェクト指向的に考えてみると、「フォルダ」や「ファイル」はそれぞれクラスと考えることができます。具体的なフォルダやファイルは、それぞれのクラスのインスタンスになるでしょう。

そして、「フォルダ」や「ファイル」に対する操作は、それぞれのメソッドとして定義できそうです。たとえば、フォルダクラスの削除メソッドを呼び出すとフォルダが削除される、といった具合です。また、ファイルクラスの削除メソッドを呼び出した場合はファイルが削除されなければなりませんね。ついでに、フォルダかファイルかを意識しないで操作できれば、利用する側は非常に便利になりそうです。

あとは、どうやってツリー状に組み上げれば良いかという問題が残っています。何となく予想がつきましたか？そう、フォルダオブジェクトの内部に別のオブジェクトを持たせてやることで再帰的なツリーが表現できそうですね。

ここまでいろいろと考えてきましたが、何となくでもイメージできたでしょうか？実は、これが**Compositeパターン**なのです。

Compositeパターンとは？

Composite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。

> 部分-全体階層を表現するために、オブジェクトを木構造に組み立てる。
> Compositeパターンにより、クライアントは、個々のオブジェクトとオブジェクトを
> 合成したものを一様に扱うことができるようになる。

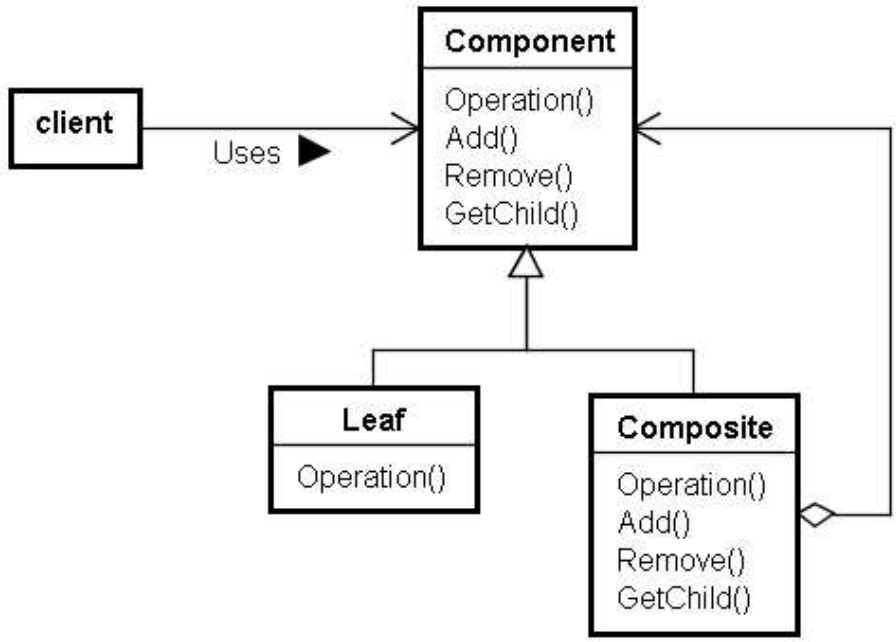
Compositeパターンはオブジェクトの構造に注目したパターンで、単体のオブジェクトとオブジェクトの集合を同一視するためのパターンです。その際、オブジェクトを木構造に組み立てるのが特徴です。

この木構造は、ファイルシステム上のディレクトリツリーをイメージするとよく分かります。逆さにすると、枝葉が延びるような形になっていますね。

Compositeパターンでは、親子関係を持つオブジェクトを再帰的に保持することで木構造を組み立てます。また、任意の枝の部分や末端の葉の部分に対して、共通の手順でアクセスできるような仕組みを提供しています。つまり、単一のオブジェクトにも、複数のオブジェクトから形成されたオブジェクトにも、同じ手順でアクセスできるAPIを提供します。

Compositeパターンの構成要素

Compositeパターンの構成要素は、次のとおりです。



▼Compositeパターンの構成要素

Componentクラス

Clientクラスに対して、共通にアクセスさせるためのAPIを提供します。このAPIには、子に相当するオブジェクトへのアクセスしたり、追加や削除するためのAPIも含まれます。

Leafクラス

Componentクラスのサブクラスの1つです。このクラスは、木構造の末端に位置する葉に相当するクラスです。このクラスは、子に相当するオブジェクトを持ちません。

Compositeクラス

Leafクラスと同様、Componentクラスのサブクラスの1つです。木構造の中で、任意枝に相当するクラスです。このクラスは、子に相当するオブジェクトを持ちます。また、Componentクラスで定義された子オブジェクトへのアクセスAPIや追加・削除APIなども実装します。

Clientクラス

ComponentクラスのAPIを通して、木構造にアクセスします。

Compositeパターンのメリット

Composite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。

クライアント側の操作が簡単になる

クライアントは、単一のオブジェクト、もしくは複数のオブジェクトから形成されたオブジェクトに同じ**API**でアクセスできます。また、木構造の枝の部分だろうが、葉の部分だろうが、同じAPIでアクセスできます。

通常ですと、今対象としているオブジェクトが枝に相当するのか葉に相当するのかを意識する必要があります。Compositeパターンを適用することで、アクセスする手順が統一されます。

新しい枝を簡単に追加できる

Compositeパターンでは、木構造の枝葉を同一視する事ができるため、新しい枝（**Composite**クラス）を追加するのが非常に簡単です。また、他のComponentクラスやLeafクラスを修正する必要はありません。

Compositeパターンの適用例

Compositeパターンの適用例を見てみましょう。

ここでは、組織とそれに所属する社員のデータを表示するサンプルです。Compositeパターンを使って組織と社員を同一視している事を意識しながら見てください。

まずは、Componentクラスに相当するOrganizationEntryクラスです。ここでは抽象クラスとして定義しています。OrganizationEntryクラスには組織と社員に共通なAPIであるgetCodeメソッドとgetNameメソッドが定義されています。

また、抽象メソッドとしてaddメソッドが定義されています。このメソッドは再帰的な構造を作るために必要です。引数は

OrganizationEntry型のオブジェクトで、この型がポイントになります。これについては、次のGroupクラスで説明していますので確認してくださいね。

もう1つ、データを出力するdumpメソッドが実装されています。これはデフォルトの実装として用意してあります。

●OrganizationEntryクラス(OrganizationEntry.class.php)

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

    private $code;
    private $name;

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

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

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

    /**
     * 子要素を追加する
     * ここでは抽象メソッドとして用意
     */
    public abstract function add(OrganizationEntry $entry);

    /**
     * 組織ツリーを表示する
     * サンプルでは、デフォルトの実装を用意
     */
    public function dump() {
        echo $this->code . ":" . $this->name . "<br>¥n";
    }
}
?>
```

続いて、Componentクラスを継承したクラスです。組織を表すGroupクラスはCompositeクラスとして、社員を表すEmployeeクラスはLeafクラスとなります。両クラスともOrganizationEntryクラスを継承し、OrganizationEntryクラスのaddメソッドの具体的な実装をおこなっています。

ここで、両者に実装の違いがあります。

組織を表すGroupクラスでは、それに属する組織や社員を追加できるよう、内部の配列にarray_push関数を使って保持するようになっています。この時、追加する内容は、OrganizationEntry型のオブジェクトです。つまり、OrganizationEntryクラスを継承しているGroupクラス、EmployeeクラスはいずれもOrganizationEntry型と言えますので、特に意識することなく、いずれのオブジェクトも引数として渡せるとことになります。

また、逆説的に内部に保持されるオブジェクトは、必ずOrganizationEntry型になることが保証されます。ここでdumpメソッドのforeach文を見てください。内部に保持した配列から1つずつオブジェクトを取り出して、そのdumpメソッドを呼び出していますね？配列の中身がどういった型のオブジェクトか分からない場合、実際にdumpメソッドがあるかどうかをチェックする必要がありますが、ここでは一切チェックをおこなっていません。これは先の説明のとおり、配列の中身が全てOrganizationEntry型のオブジェクトであることが保証されている、つまり必ずdumpメソッドが存在しているからこそ可能になっています。

●Groupクラス(Group.class.php)

```php
<?php
require_once 'OrganizationEntry.class.php';
?>
<?php
/**
 * Compositeクラスに相当する
 */
class Group extends OrganizationEntry {
```

```
    private $entries;

    public function __construct($code, $name) {
        parent::__construct($code, $name);
        $this->entries = array();
    }

    /**
     * 子要素を追加する
     */
    public function add(OrganizationEntry $entry) {
        array_push($this->entries, $entry);
    }

    /**
     * 組織ツリーを表示する
     * 自分自身と保持している子要素を表示
     */
    public function dump() {
        parent::dump();
        foreach ($this->entries as $entry) {
            $entry->dump();
        }
    }
}
?>
```

一方、社員を表すEmployeeクラスを見てみましょう。「社員」というものは「社員」自身に属する組織や社員を持つことはできません。ですので、ここではaddメソッドを呼び出した場合に例外を投げるよう実装しています。

●Employeeクラス(Employee.class.php)

```
<?php
require_once 'OrganizationEntry.class.php';
?>
<?php
/**
 * Leafクラスに相当する
 */
class Employee extends OrganizationEntry {

    public function __construct($code, $name) {
        parent::__construct($code, $name);
    }

    /**
     * 子要素を追加する
     * Leafクラスは子要素を持たないので、例外を発生させている
     */
    public function add(OrganizationEntry $entry) {
        throw new Exception('method not allowed');
    }
}
?>
```

最後に、説明してきたクラス群を利用するクライアント側のコードです。

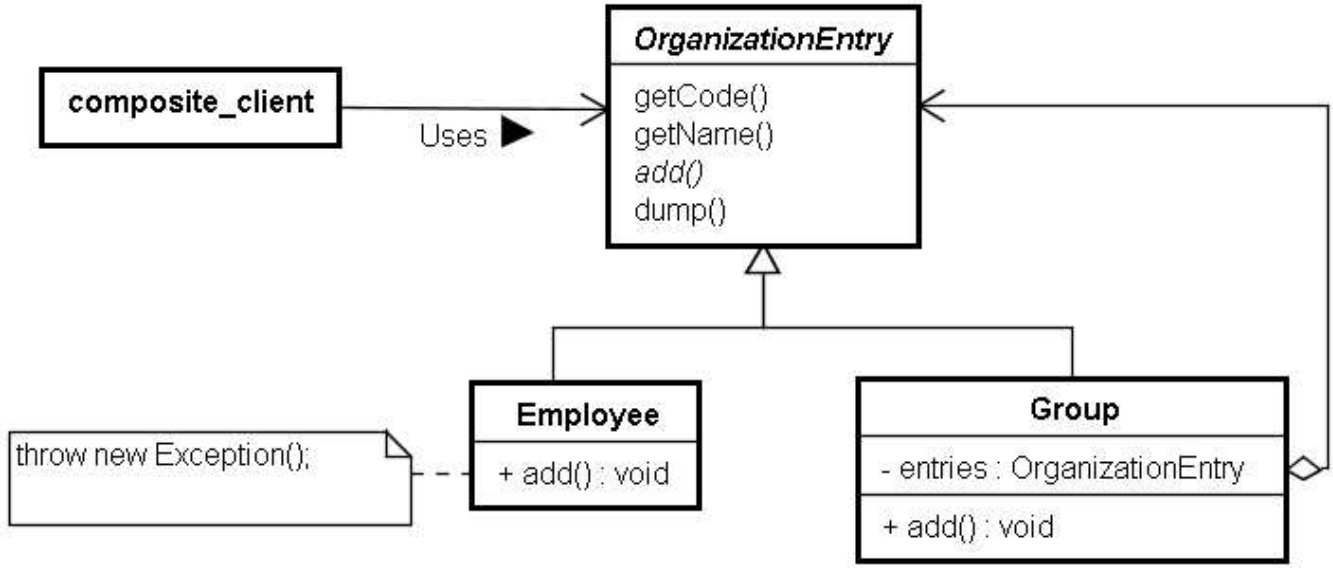
まず、組織の木構造を作っていますが、addメソッドの引数に注目してください。Groupオブジェクトだろうがemployeeオブジェクトだろうが、構わず引数に指定されているのが分かると思います。

●クライアント側コード(composite_client.php)

```
<?php
require_once 'Group.class.php';
require_once 'Employee.class.php';
?>
<?php
    /**
     * 木構造を作成
```

```
 */
$root_entry = new Group("001", "本社");
$root_entry->add(new Employee("00101", "CEO"));
$root_entry->add(new Employee("00102", "CTO"));

$group1 = new Group("010", "〇〇支店");
$group1->add(new Employee("01001", "支店長"));
$group1->add(new Employee("01002", "佐々木"));
$group1->add(new Employee("01003", "鈴木"));
$group1->add(new Employee("01003", "吉田"));

$group2 = new Group("110", "△△営業所");
$group2->add(new Employee("11001", "川村"));
$group1->add($group2);
$root_entry->add($group1);

$group3 = new Group("020", "××支店");
$group3->add(new Employee("02001", "萩原"));
$group3->add(new Employee("02002", "田島"));
$group3->add(new Employee("02002", "白井"));
$root_entry->add($group3);

/**
 * 木構造をダンプ
 */
$root_entry->dump();
?>
```

最後にサンプルコードのクラス図を示しておきます。



Compositeパターンのオブジェクト指向的要素

Composite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。

Componentクラスでは、Clientクラスに対して共通にアクセスさせるためのAPIを提供しています。また、Compositeクラスはごにょごにょと処理を実行し、CompositeクラスやLeafクラスのインスタンスを、Clientクラスに適切に返します。これらのインスタンスは、いずれもComposite型であるところが第1のポイントです。Clientクラスは、返されたインスタンスが、**Composite**クラスのインスタンスなのか、**Leaf**クラスのインスタンスなのか分かりません。しかし、Component型であることは分かっています。Clientクラスでは、**Component**クラスで提供された**API**だけを使ってプログラミングすることで、Componentクラスの向こうにある**Composite**クラスや**Leaf**クラスを意識することなく、同一視することができます。

また、Componentクラスのサブクラスであるクラスでは、内部にComponent型のオブジェクトを保持しています。このオブジェクトが、自分自身の子に相当するオブジェクトになります。ここでも、このオブジェクトは「Component型である」というだけで、具体的にCompositeクラスのインスタンスなのか、Leafクラスのインスタンスなのかは分かりません。

何となく気づきましたか？そうです。Compositeクラスで実装する「子に対するアクセスメソッド」は、Componentクラスが提供しているAPIだけを使うことで、ここでも枝葉の同一視をしています。ClientクラスとComponentクラスの関係と同じですね。

関連するパターン

Chain of Responsibilityパターン

Chain of Responsibilityパターンもオブジェクトどうしのつながりを持っているパターンです。

Commandパターン

複数のコマンドを組み合わせ、大きなコマンドを作る場合にCompositeパターンが使われます。

Decoratorパターン

Compositeパターンと良く併用されるパターンです。

まとめ

ここでは「単一のオブジェクト」と「オブジェクトの集合」を同一視するCompositeパターンについてみてきました。