Visitor ～要素と要素に対する操作を分離する

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＋オブジェクト

はじめに

ここではVisitorパターンについて説明します。

「visitor」とは「visit」する人、つまり「訪問者」という意味ですね。Visitorパターンは、訪問者に相当するクラスが登場します。

では、この訪問者は誰を訪問し、何をするのでしょうか？

それをこれから見ていきましょう。

たとえば

ファイルシステムのディレクトリツリーのようなあるデータ構造の各要素に対して、何らかの処理をおこなう場合を考えてみましょう。この場合、要素はディレクトリ(フォルダ)やファイルになります。

これら要素に対する処理として、ファイルサイズの合計を取得するとします。この場合、情報のカプセル化やコードを書く効率を考えると、自身のサイズを返すAPIを用意し、このAPIを再帰的に呼び出すことで比較的容易に実装できそうです。つまり、処理コードをそれぞれの要素の中に書いてしまう、ということです。

しかし、新しい処理を追加する必要に迫られた場合はどうでしょうか？

そうですね。それぞれの要素の中に処理コードを書いていますので、新しい処理のコードを追記する必要があります。要素の数が多い場合や追加するコードが複雑な場合は非常にコストのかかる作業になってしまいます。

これは、データ構造とそれに対する処理が同居してしまっているため、どちらかを修正した場合に他方にも影響が出てしまっている状態だと言えます。

ここで、データ構造と処理を分けておけば、新しい操作を追加する場合もデータ構造を変更する必要がなくなることが期待できます。また、分けることでそれぞれのコードに集中することができ、コードもシンプルになりそうです。

これから詳しく見ていくVisitorパターンは、データ構造から分離された「操作」がデータ構造を渡り歩き、順に処理をおこなっていくという一風変わった動作をします。

そして、この「操作」を表すオブジェクトが「訪問者」(visitor)として振る舞っているように見えることから、visitorという名が付いています。

Visitorパターンとは？

Visitor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。

> あるオブジェクト構造上の要素で実行されるオペレーションを表現する。
> Visitor パターンにより、オペレーションを加えるオブジェクトのクラスに変更を加えずに、
> 新しいオペレーションを定義することができるようになる。

Visitorパターンは、オブジェクトの振る舞いに注目したパターンで、データ構造上の要素とそれに対する操作を分離するパターンです。

Visitorパターンは、データ構造を表すクラス階層と操作を表すクラス階層から構成されます。

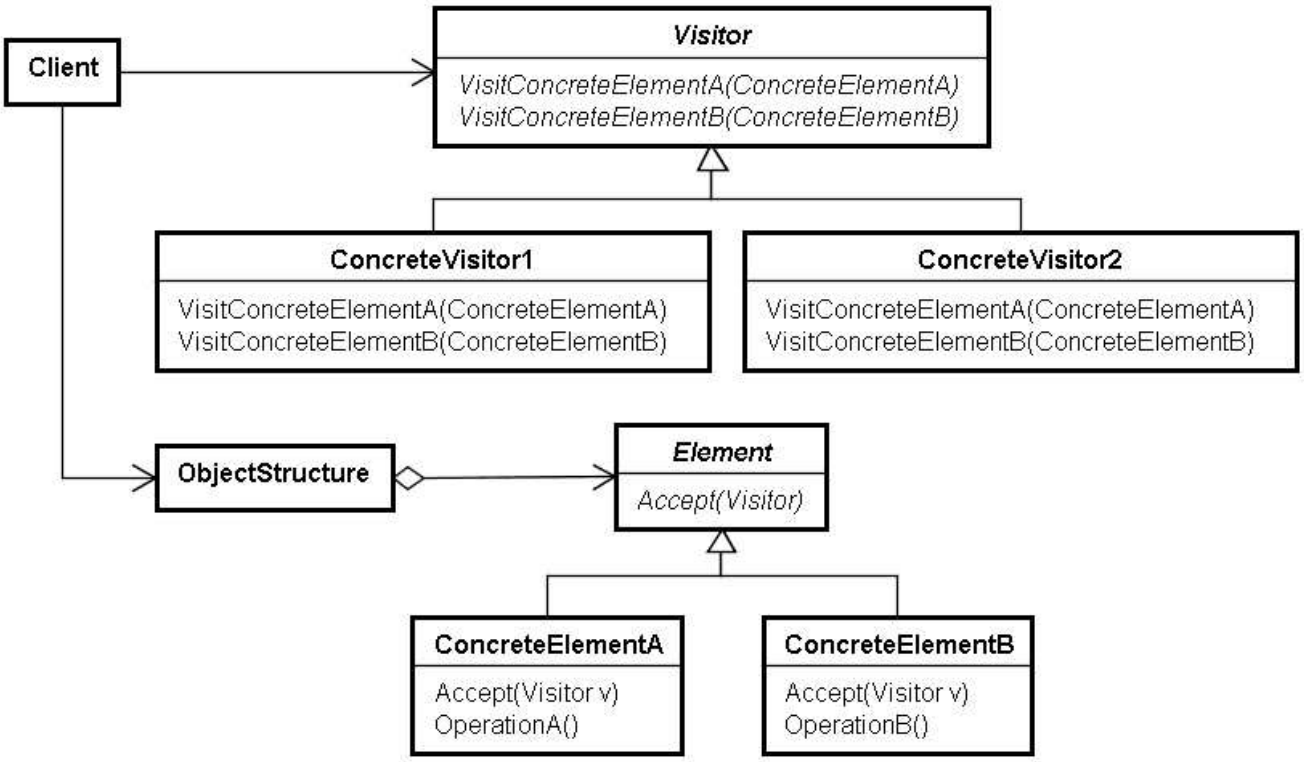
データ構造のクラス階層には当然、データ構造を表すための関連やメソッドが用意されますが、訪問者を受け入れるためのメソッドが別に用意されます。一方の訪問者である操作のクラス階層には、データ構造上の各要素に対する処理が実装されます。そして、操作クラスがデータ構造クラスに受け入れられたとき、データ構造の要素は「訪問者」である操作クラスに具体的な操作を依頼します。こうすることで、それぞれの要素を処理可能になります。

では、「データ構造を訪問者が渡り歩く」ためのコードはどこに書けばよいでしょうか？一概には言えませんが、次の3つのうちいずれかに実装することになります。それぞれのメリットとデメリットを考慮して、どこに実装するかを決めていきます。

実装箇所	概要
データ構造のクラス階層	このクラス階層がデータ構造を管理しているので、クラスの責任範囲が明確になりやすい
iterator	操作クラスが走査をおこなわずにデータ要素クラスを呼び出すだけの場合に有効
操作クラス	データ構造のクラス階層から渡り歩くための情報を取得できる場合に利用できる。ただし、不要な情報まで公開しないことに注意

Visitorパターンの構造

Visitorパターンの構成要素は、次のとおりです。



Visitorクラス

操作のクラス階層で最上位に位置するクラスです。データ構造の具体的な要素であるConcreteElementクラスを処理するためのメソッドを用意します。ConcreteElementクラスはこのメソッドを呼び出すことで処理を依頼します。

このメソッドは通常引数の型が異なるだけの同名メソッド（オーバーロード）として実装されますが、PHP5では文法上許されていません。このため、メソッド内で引数の型を判断し、それぞれのConcreteElementクラス用の処理をおこなうことになります。

ConcreteVisitorクラス

Visitorクラスのサブクラスです。Visitorクラスで宣言された処理用メソッドを実装します。

Elementクラス

データ構造のクラス階層で最上位に位置するクラスです。Visitor型のオブジェクトを受け入れるためのメソッドを宣言します。このメソッドは引数としてVisitor型のオブジェクトを受け取ります。

ConcreteElementクラス

Elementクラスのサブクラスです。Elementクラスで宣言された受け入れ用メソッドを実装します。

ObjectStructureクラス

Elementクラスの集合を表すクラスです。

Visitorパターンのメリット

Visitor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。

共通な操作を局所化する

データ構造と操作を一緒にコーディングしてしまうと、コードが複雑化し拡張性も乏しくなってしまいます。Visitorパターンを適用することで、データ構造と操作が分離され、それぞれのコードが局所化されるため、よりシンプルなコードになりやすくなります。また、それぞれのクラスの部品化を促進することになります。

新しい操作を簡単に追加できる

Visitorパターンは、データ構造とデータ要素に対する操作を分離するパターンです。データ構造と操作は、具体的にはそれぞれElementクラスの階層とVisitorクラスの階層になります。Visitorパターンは、これらの階層どうしの結びつきをゆるくする働きがあります。このためデータ構造を変更することなく、新しいConcreteVisitorクラスを追加したり拡張できます。

一方、新しいデータ要素、つまりConcreteElementクラスを追加することは容易ではありません。例えば、サンプルコードに新しいデータ要素クラスであるDivisionクラスを追加する場合を考えてみましょう。これまで何度も出てきましたが、Visitorパターンはデータ構造と「データ要素に対する操作」を分離します。つまり、**Visitorクラスに新しいデータ要素に対する操作を追加しなければならなくなります**。

このことから、Visitorパターンは「データ構造は変わらないが操作を色々変化させたい」という場合に向いていると言えるでしょう。

Visitorパターンの適用例

それではVisitorパターンの適用例を見てみましょう。

ここでは組織と属する社員を表示するサンプルで、Compositeパターンの適用例にあるサンプルにVisitorパターンを適用したものになります。

まずは、組織構造を形成するためのクラス群です。

OrganizationEntryクラスは、オブジェクト構造に共通のAPIを定義し、組織や社員のコードと名前を格納しておくためのクラスです。ここでは抽象クラスとしており、後ほど出てくる組織クラスと社員クラスがサブクラスとなります。

注目は訪問者を迎え入れるためのacceptメソッドです。受け取ったVisitor型のオブジェクトのvisitメソッドに自分自身を渡して、「自分自身に対する処理」をお願いしています。具体的にどの様な処理がおこなわれるかは、訪問者によって異なります。

●OrganizationEntryクラス(OrganizationEntry.class.php)

```php
<?php
require_once 'Visitor.class.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);

    /**
     * 子要素を取得する
     * ここでは抽象メソッドとして用意
     */
    public abstract function getChildren();

    /**
     * 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 accept(Visitor $visitor) {
        $visitor->visit($this);
    }
}
?>
```

次にOrganizationEntryクラスを継承した組織クラスと社員クラスです。この2クラスには、特に難しいところはないと思います。

●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);
    }

    /**
     * 子要素を取得する
     */
    public function getChildren() {
        return $this->entries;
```

```
    }

}
?>
```

●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');
    }

    public function getChildren() {
        return array();
    }
}
?>
```

続けて訪問者側のクラス群を見ていきましょう。

まずは、訪問者の共通APIを定義するVisitorインターフェースです。唯一のメソッドvisitが用意されており、引数として組織構造であるOrganizationEntry型のオブジェクトを受け取るようになっています。

●Visitorクラス(Visitor.class.php)

```
<?php
require_once 'OrganizationEntry.class.php';
?>
<?php
interface Visitor {
    public function visit(OrganizationEntry $entry);
}
?>
```

次はConcreteVisitorクラスに相当する具体的な訪問者クラスです。ここでは2種類の訪問者を用意しました。また、データ構造を走査するコードは訪問者に記述しています。

1つ目は組織や社員のコードと名前を表示するDumpVisitorクラスです。

visitメソッドを見てください。まず、渡されたOrganizationEntry型のオブジェクト自身の情報を表示します。続いて、ぶら下がっている組織や社員を再帰的に訪問(visit)していることが分かると思います。

●DumpVisitorクラス(DumpVisitor.class.php)

```
<?php
require_once 'Visitor.class.php';
?>
<?php
class DumpVisitor implements Visitor {
    public function visit(OrganizationEntry $entry) {
        if (get_class($entry) === 'Group') {
            echo '■';
        } else {
            echo '  ';
        }
        echo $entry->getCode() . ":" . $entry->getName() . "<br>¥n";

        foreach ($entry->getChildren() as $ent) {
            $ent->accept($this);
        }
    }
}
?>
```

2つ目は組織や社員の数をカウントするCountVisitorクラスです。

visitメソッドを見みると、先のDumpVisitorクラスのvisitメソッドと似ていることが分かると思います。こちらも渡された組織構造を再帰的に訪問し、組織と社員の数を別々にカウントしています。カウントした結果は、getGroupCountメソッドとgetEmployeeCountメソッドでそれぞれ取得できます。

●CountVisitorクラス(CountVisitor.class.php)

```
<?php
require_once 'Visitor.class.php';
?>
<?php
class CountVisitor implements Visitor {
    private $group_count = 0;
    private $employee_count = 0;

    public function visit(OrganizationEntry $entry) {
        if (get_class($entry) === 'Group') {
            $this->group_count++;
        } else {
            $this->employee_count++;
        }
        foreach ($entry->getChildren() as $ent) {
            $this->visit($ent);
        }
    }

    public function getGroupCount() {
        return $this->group_count;
    }

    public function getEmployeeCount() {
        return $this->employee_count;
    }
}
?>
```

ここまで説明してきたクラスを利用する側のコードも見ておきましょう。

最初に組織構造を作成している部分は、Compositeパターンにあるものと同じです。しかし、組織構造を処理するときに処理をおこなうオブジェクト(Visitor型のオブジェクト)を渡しているところが特徴的です。

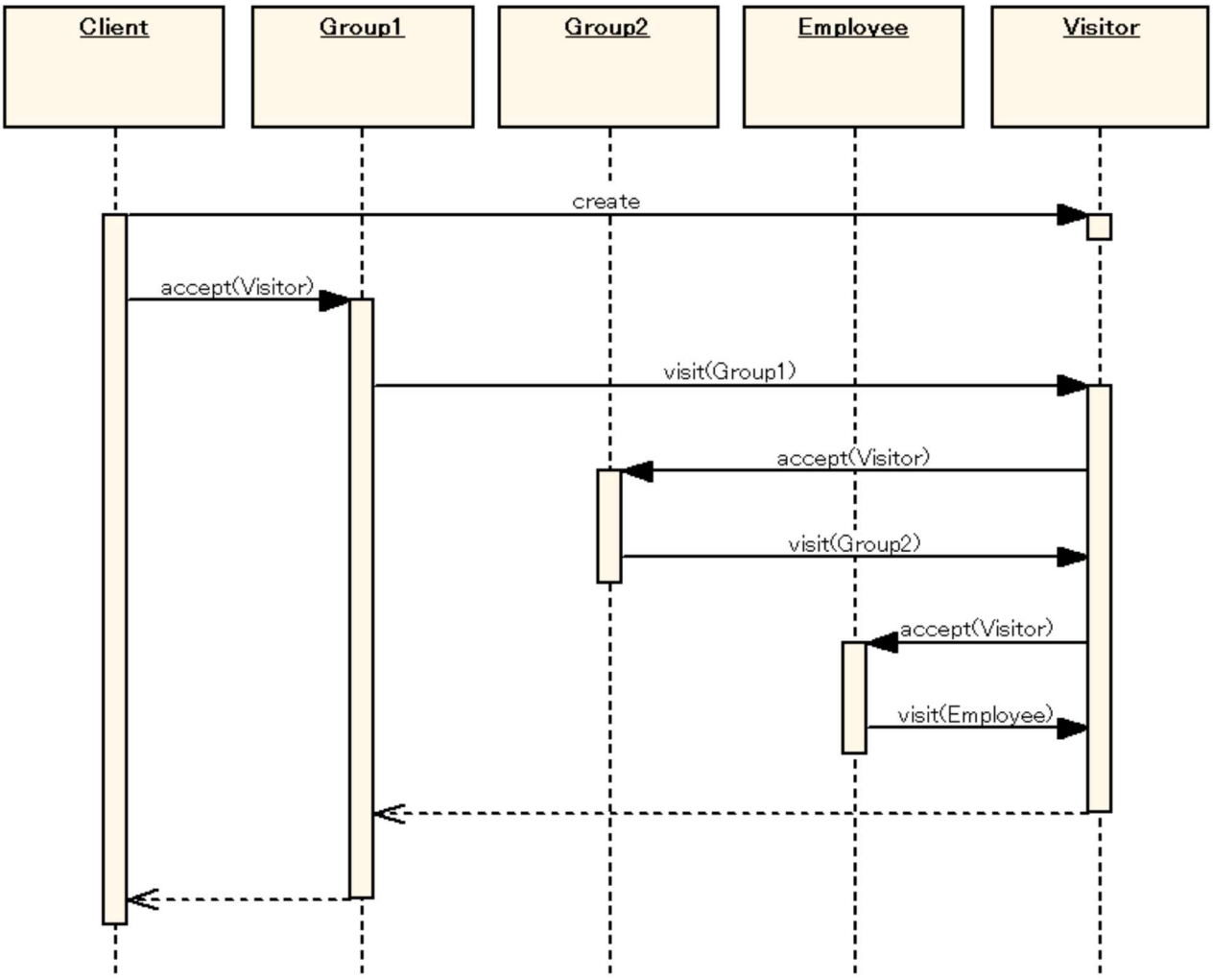
このように、オブジェクト構造を修正することなく、Visitor型のオブジェクトを変えるだけでさまざまな処理をおこなうことができます。

●クライアント側コード(visitor_client.php)

```
<?php
require_once 'Group.class.php';
require_once 'Employee.class.php';
require_once 'DumpVisitor.class.php';
require_once 'CountVisitor.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->accept(new DumpVisitor());

    /**
     * 同じ木構造に対して、別のVisitorを使用する
     */
```

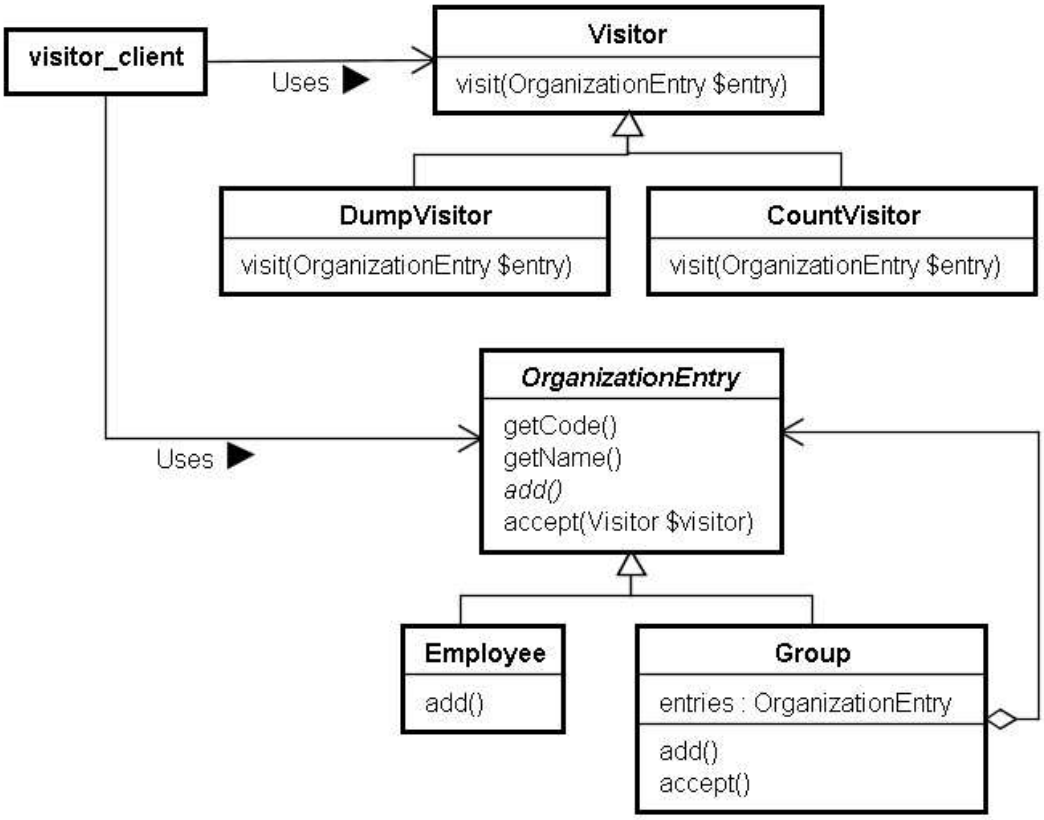
```
    $visitor = new CountVisitor();
    $root_entry->accept($visitor);
    echo '組織数：' . $visitor->getGroupCount() . '<br>';
    echo '社員数：' . $visitor->getEmployeeCount() . '<br>';
?>
```

ここで、データ構造を渡り歩く訪問者の様子を示しておきます。話を単純にするため、組織は2階層（Group1とGroup2）で2階層目の組織に社員（Employee）が属している場合の例になっています。



こう見ると、GroupオブジェクトもEmployeeオブジェクトも、Visitorオブジェクトをacceptメソッドで受け入れ、VisitorオブジェクトのvisitメソッドにVisitor自分自身を渡して処理してもらっている様子が分かりますね。

最後に、サンプルコードのクラス図を示します。



Visitorパターンのオブジェクト指向的要素

Visitorパターンの特徴的な要素に「ダブルディスパッチ（**double dispatch**）」が挙げられます。ここでは、このダブルディスパッチについて見ていきましょう。

まず、ダブルディスパッチとは何でしょうか？簡単には「実行される処理が**2**つの型に依存すること」と言えます。

Visitorパターンの場合、「2つの型」とはConcreteVisitorとConcreteElementを指します。Visitorパターンでは、Elementのacceptメソッドと、与えられたVisitorのvisitメソッドによって実行される処理が確定します。このacceptメソッドとvisitメソッドの2つに対して要求が送信される（dispatchされる）ことから、ダブルディスパッチと呼ばれます。

関連するパターン

Compositeパターン

データ構造にCompositeパターンが適用されることがあります。サンプルコードでは、データ構造にCompositeパターンが適用されています。

Iteratorパターン

Visitorパターンと同様、あるデータ構造に対して操作をおこなうパターンです。

Visitorパターンはデータ構造の要素に対して操作をおこなうパターンですが、Iteratorパターンはデータ構造の要素を1つずつ取得するために利用されるパターンです。

まとめ

ここではデータ構造から分離された「操作」がデータ構造を渡り歩いて処理をおこなうVisitorパターンを説明しました。