Bridge ～実装と機能の架け橋

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Bridgeパターンについて説明します。

「bridge」とは「橋」の意味ですね。橋は川の両岸や島と島など、ある点とある点を結ぶ役割を持っています。

では、Bridgeパターンは、何と何を結ぶ橋なのでしょうか？早速、見てみましょう。

たとえば

何らかの処理を実装する場面を考えてみましょう。当然ですが、「何をするのか」は決まっていますね？しかし、「どうやって実現するのか」が色々考えられる場合があります。

たとえば、データのソート処理はその代表例と言えるでしょう。「ソートをする」という「何をするのか」は分かっていても、「どうやって実装するのか」には、バブルソートやヒープソート、クイックソートなど様々なロジックが存在します。

このような場合、それぞれの実装ごとにクラスを用意してやることが最も単純になるでしょう。しかし、クラスの数があっという間に増えてしまいます。

また、機能を拡張しようとした場合、すべてのクラスを変更する必要がでてきてしまいます。クラスの数が多ければ多いほど、修正に要する作業は大きくなってしまいます。

もう一つ、何らかのデータソースからデータを読み込んで一覧表示するアプリケーションを考えてみましょう。このアプリケーションの機能、つまり「何をするのか」は、以下のようにまとめられると思います。

- データソースを開く
- データを読み込む
- データを一覧表示する

この程度のアプリケーションであれば、みなさんも簡単に作成してしまうことでしょう。

ここで、データソースへのアクセス方法やデータの取得方法、データの表示形式が色々考えられる場合はどうでしょうか？if文やswitch文を使って処理を分岐させることになるでしょう。しかし、新しい機能の追加や実装の変更があるたび、コードの修正をおこなっていてはテストをやり直す必要がありますし、if文やswitch文も複雑になり、メンテナンスビリティの悪いコードになってしまいます。

これは、「何をするのか」と「どうやって実現するのか」を一緒に考えてしまっているから起こっている問題です。ここで「何をするのか」と「どうやって実現するのか」を分けて考えてみると、機能を拡張する場合は「何をするのか」側を変えることになりますし、実装の方法を変える場合は「どうやって実現するのか」側を変えれば良いことになります。

「何をするのか」と「どうやって実現するのか」分けて考え、これらを結びつけるための橋を用意するパターン、それが**Bridgeパターン**です。

Bridgeパターンとは？

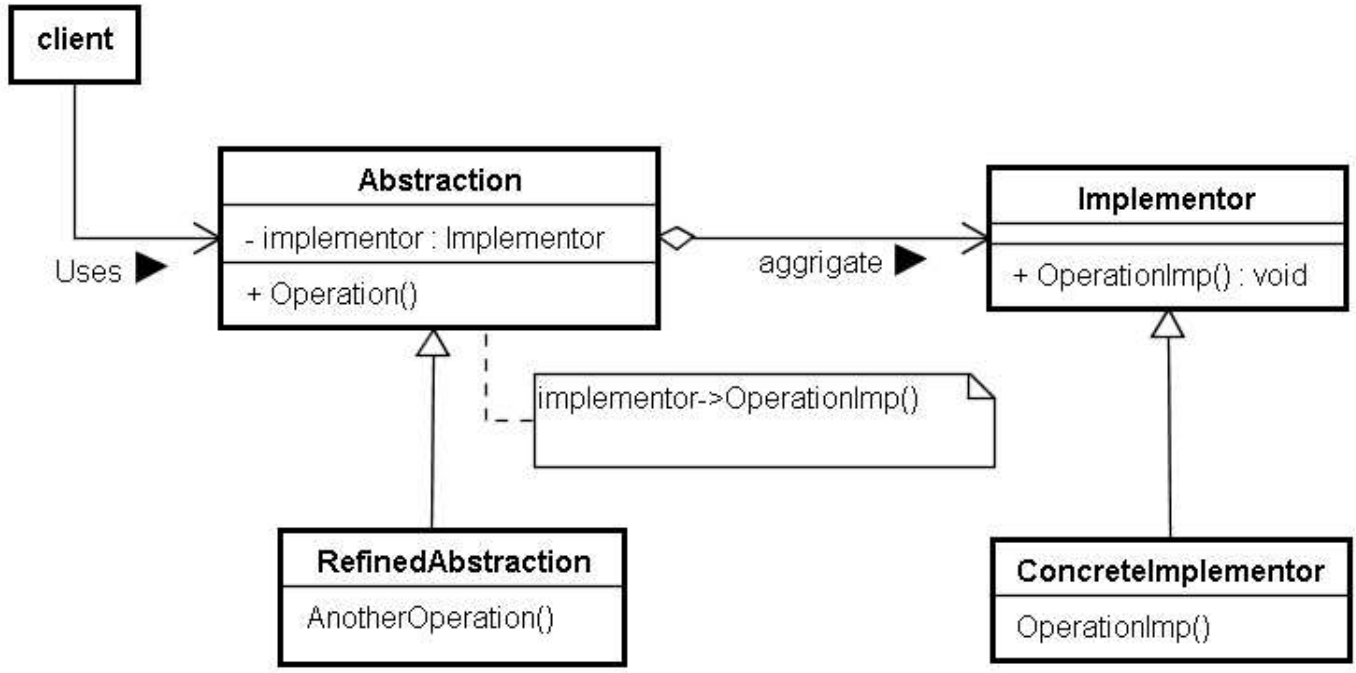
Bridgeパターンの目的は、GoF本では次のように定義されています。

> 抽出されたクラスと実装を分離して、それらを独立に変更できるようにする。

Bridgeパターンはオブジェクトの構造に注目したパターンで、「機能を提供するクラス群」と「実装を提供するクラス群」を分けるパターンです。「分ける」と言っても「インターフェース」と「実装クラス」を分けることを言っているのではなく、「何をするのか」と「どうやって実現するのか」を分けるということです。また、「Bridge」とは「橋」の意味ですが、委譲を使うことで「機能を提供するクラス群」と「実装を提供するクラス群」を橋渡ししているように見えることから名付けられています。

Bridgeパターンの構造

Bridgeパターンのクラス図と構成要素は、次のようになります。



Abstractionクラス

「何をするのか」を実現するクラス群で最上位に位置するクラスです。内部には、**Implementorオブジェクトを保持**していて、Implementorクラスが提供する機能をclientに提供します。

RefinedAbstractionクラス

Abstractionクラスで提供される機能を拡張するサブクラスです。

Implementorクラス

「どうやってするのか」を実現するクラス群で最上位に位置するクラスです。クラスではなく、インターフェースとして実装される場合もあります。また、Abstractionクラスで提供しているAPIに一致する必要はありません。

ConcreteImplementorAクラス、ConcreteImplementorBクラス

Implementorクラスを継承したサブクラスです。このクラスに具体的な実装をおこないます。

Bridgeパターンのメリット

Bridge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。

クラス階層の見通しが良くなる

「機能」と「実装」を提供するクラス群が分けられているので、クラス階層を理解しやすく、見通しが良くなります。つまり、保守性が高くなると言えます。機能と実装が一つのクラスで実装されていると、概してクラス階層が複雑になり、どの部分が機能なのか実装なのかが分かりづらくなるため、保守性が低くなりがちです。

最終的に作成すべきクラス数を抑えることができる

継承を使った単純な多態性を使った場合と比べ最終的に作成すべきクラス数を抑えることができます。例として、機能の種類が4つ、実装の種類が3つある場合を考えてみます。継承を使った単純な多態性のみの場合ですと、

```
親クラス：1 ＋ サブクラス：12（4×3） ＝ 13
```

のクラスを作成する必要があります。一方、Bridgeパターンを使うと、

```
親クラス：2 ＋ サブクラス：7（4+3） ＝ 9
```

のクラスを作成することになります。当然、機能機能や実装が増えるたびに修正しなければならないクラスの数に開きが出てくることは明かです。

機能の拡張と実装の切り替えが容易

「機能」と「実装」を分けることで、お互いに影響するなく拡張や切り替えが可能になります。機能を拡張したい場合、機能を提供するクラス側のみ変更することになります。また、実装を追加したい場合も同様のことが言えます。

Bridgeパターンの適用例

Bridgeパターンの適用例を見てみましょう。

ここでは、データを取得して表示するアプリケーションにBridgeパターンを適用してみます。このアプリケーションはかなり単純ですのでBridgeパターンを適用するほどではありませんが、Bridgeパターンを適用すると設計がスマートになります。

まずは「どうやって実現するのか」側から見ていきましょう。

DataSourceインターフェースは「どうやって実現するのか」側の最上位に位置するクラスで、Implementorクラスに相当します。今回はインターフェースとして実装しています。また、3つのメソッドopen、read、closeが定義されていますが、これは後ほど出てくるListingクラスが利用するAPIになります。

●DataSourceインターフェース(DataSource.class.php)

```
<?php
/**
 * Implementorに相当する
 * このサンプルでは、インターフェースとして実装
 */
interface DataSource {
    public function open();
    public function read();
    public function close();
}
?>
```

続けてConcreteImplementorクラスに相当するFileDataSourceクラスです。DataSourceインターフェースを実装し、定義された3メソッドを具体的に実装しています。今回は名前の通り、データソースとしてファイルを使用しています。このファイル名はコンストラクタの引数として指定するようになっています。

●FileDataSourceクラス(FileDataSource.class.php)

```
<?php
require_once 'DataSource.class.php';
?>
<?php
/**
 * Implementorクラスで定義されている機能を実装する
 * ConcreteImplementorに相当する
 */
class FileDataSource implements DataSource {

    /**
     * ソース名
     */
    private $source_name;

    /**
     * ファイルハンドラ
     */
    private $handler;

    /**
     * コンストラクタ
     * @param $source_name ファイル名
     */
    function __construct($source_name) {
        $this->source_name = $source_name;
    }

    /**
     * データソースを開く
     * @throws Exception
     */
    function open() {
        if (!is_readable($this->source_name)) {
            throw new Exception('データソースが見つかりません');
        }
        $this->handler = fopen($this->source_name, 'r');
        if (!$this->handler) {
            throw new Exception('データソースのオープンに失敗しました');
        }
```

```
    }

    /**
     * データソースからデータを取得する
     * @return string データ文字列
     */
    function read() {
        $buffer = array();
        while (!feof($this->handler)) {
            $buffer[] = fgets($this->handler);
        }
        return join($buffer);
    }

    /**
     * データソースを閉じる
     */
    function close() {
        if (!is_null($this->handler)) {
            fclose($this->handler);
        }
    }
}
?>
```

さて、もう一方の「何をするのか」側も見てみましょう。

Listingクラスは「何をするのか」側の最上位に位置するクラスで、利用者に提供するAPIを定義しています。このAPIは、Implementorクラスであ るDataSourceインターフェースと同じAPIとしています。また、内部にDataSource型のオブジェクトを保持するようになっており、open、read、closeの各メソッドは具体的な処理をこのオブジェクトに委譲しています。

この部分が「何をするのか」と「どうやって実現するのか」を結ぶ「橋」となっています。お分かりでしょうか？

また、このクラスに実装側の具体的なクラス名が出てきていないことを確認してください。つまり、具体的な実装を意識することなく、利用者側に機能のAPIを提供することが可能になっていることが分かります。

●Listingクラス（Listing.class.php）

```
<?php
require_once 'DataSource.class.php';

class Listing {
    private $data_source;

    /**
     * コンストラクタ
     * @param $source_name ファイル名
     */
    function __construct($data_source) {
        $this->data_source = $data_source;
    }

    /**
     * データソースを開く
     */
    function open() {
        $this->data_source->open();
    }

    /**
     * データソースからデータを取得する
     * @return array データの配列
     */
    function read() {
        return $this->data_source->read();
    }

    /**
     * データソースを閉じる
     */
    function close() {
        $this->data_source->close();
    }
}
?>
```

次は、Listingクラスの機能を拡張したクラスです。ExtendedListingクラスはListingクラスを継承し、さらに新しい機能のためのメソッドreadWithEncodeが追加されています。

このクラスにも実装側の具体的なクラス名は出てきていませんね。具体的な実装と関係なく、機能が拡張できています。

●ExtendedListingクラス(ExtendedListing.class.php)

```
<?php
require_once 'Listing.class.php';
?>
<?php
/**
 * Listingクラスで提供されている機能を拡張する
 * RefinedAbstractionに相当する
 */
class ExtendedListing extends Listing {

    /**
     * コンストラクタ
     * @param $source_name ファイル名
     */
    function __construct($data_source) {
        parent::__construct($data_source);
    }

    /**
     * データを読み込む際、データ中の特殊文字を変換する
     * @return 変換されたデータ
     */
    function readWithEncode() {
        return htmlspecialchars($this->read(), ENT_QUOTES);
    }

}
?>
```

クライアント側のコードも見てみましょう。ここでは、ListingクラスとExtendedListingクラスの両方をインスタンス化しています。コンストラクタの引数に、データの読み込み処理を具体的に実装したクラスFileDataSourceを指定しています。

●クライアント側コード(bridge_client.php)

```
<?php
require_once 'Listing.class.php';
require_once 'ExtendedListing.class.php';
require_once 'FileDataSource.class.php';
?>
<?php
    /**
     * Listingクラス、ExtendedListingクラスをインスタンス化する。
     * 具体的な処理クラスとして、FileDataSourceクラスを使う。
     * データファイルは、data.txt
     */
    $list1 = new Listing(new FileDataSource('data.txt'));
    $list2 = new ExtendedListing(new FileDataSource('data.txt'));

    try {
        $list1->open();
        $list2->open();
    }
    catch (Exception $e) {
        die($e->getMessage());
    }

    /**
     * 取得したデータの表示 (readメソッド)
     */
    $data = $list1->read();
    echo $data;

    /**
     * 取得したデータの表示 (readWithEncodeメソッド)
```

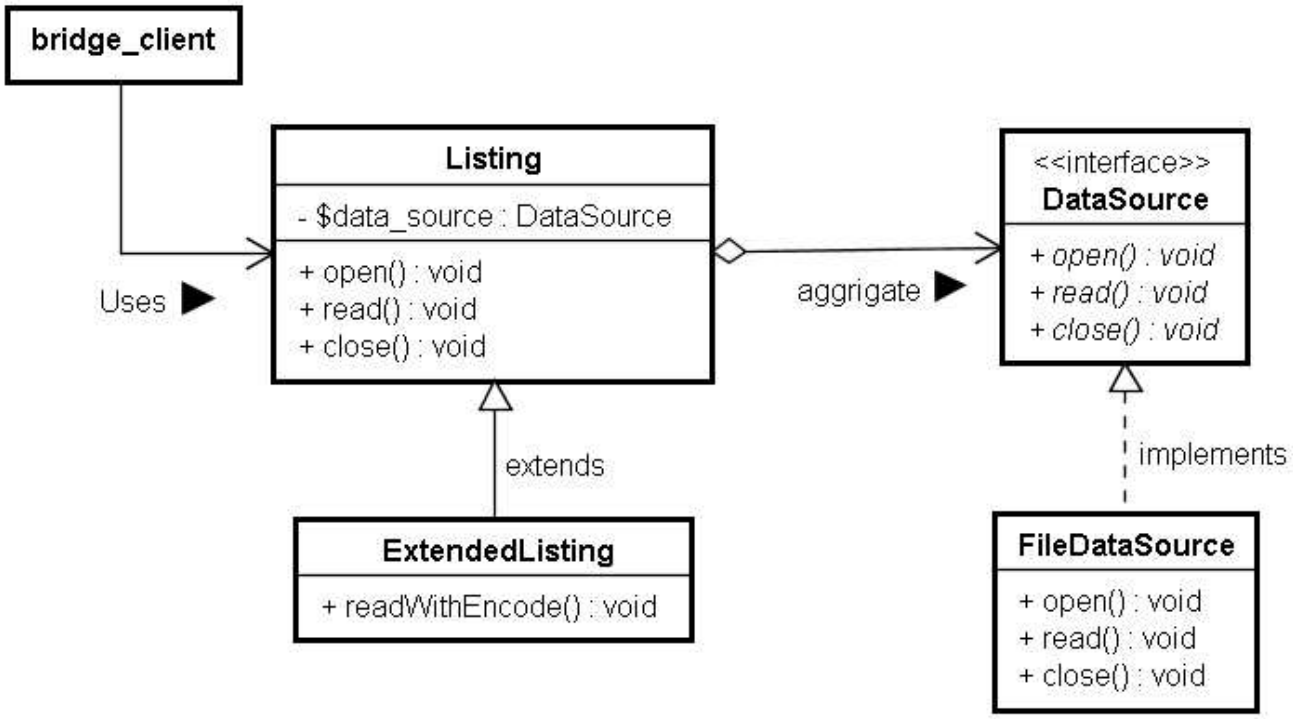
```
 */
$data = $list2->readWithEncode();
echo $data;


$list1->close();
$list2->close();
?>
```

今回サンプルとして用意したデータファイルは次のようなものです。

●データファイル(data.txt)

```
<h1>This is a sample of the Bridge Pattern !</h1>
```

最後に、Bridge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。



Bridgeパターンのオブジェクト指向的要素

Bridgeパターンは「多態性」と「委譲」を非常に活用しているパターンです。

機能を提供するクラス側では、まずクライアント側に提供する「機能のAPI」をクラスで定義し、そのクラスを継承することで機能の拡張を実現します。一方、実装を提供するクラス側でも同様に、機能側のクラスに提供する「実装のAPI」をインターフェースもしくはクラスで定義し、それらを実装もしくは継承することで異なる実装を実現しています。ここまでの内容を見てみると、機能を提供するクラス群と実装を提供するクラス群はそれぞれ**Template Method**パターンになる場合があります。お分かりでしょうか？

では、ここからが本番です。クライアントに提供する具体的な処理の実装を、実装を提供するクラス群に委譲している、つまり、実装を提供するクラス群の最上層で定義されている「実装のAPI」に基づいたクラスに任せています。ここがBridgeパターンの神髄ともいうべきところで、「Bridge」と名付けられた所以です。この委譲を使うことにより、実装側のクラスにある具体的な処理内容を意識することがなくなります。この「意識することがなくなる」ため、処理を切り替える場合も機能側に属するクラスを一切変更する必要がなくなります。また、あとで新しい実装を追加したりする事が可能になります。

また、Bridgeパターンでは、クライアント・機能側のクラスとの間にある「**機能のAPI**」と、機能側のクラス・実装側のクラスとの間にある「**実装のAPI**」で、多態性を二度使っている、という見方もできますね。

関連するパターン

Abstract Factoryパターン

ConcreteImplementorを適切に構築するために使われる場合があります。

Adapterパターン

Adapterパターンの構造と比較するとよく分かりますが、BridgeパターンはAdapterパターンと非常によく似ています。また、本質的にも変

わりありません。

ただし、Adapterパターンのように「既存クラスを再利用するために繋ぎ合わせる」といった後天的な理由ではなく、「設計の段階で実装と機能を分離し、それぞれを繋ぎ合わせる」といった先天的な理由で導入されます。その場合、実装の変更が考えられる処理については、実装のクラス階層に処理を委譲するようにします。

まとめ

ここでは「機能」のクラス階層と「実装」のクラス階層を橋のように結ぶBridgeパターンについて見てきました。