Strategy ～戦略を切り替える

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Strategyパターンについて見ていきましょう。

strategyという単語は「戦略」「作戦」「方針」「方策」などの意味があります。

Strategyパターンは、この「作戦」や「戦略」を1つのクラスにまとめ、「作戦」や「戦略」の単位で切り替えられるようにするパターンです。

たとえば

CSVファイルもしくはXMLファイルを読み込んでデータ処理をおこなう場合を考えてみましょう。いずれの場合も、処理フローは次のようになります。

データファイルを読み込む

読み込んだデータを処理する

しかし、「データファイルを読み込む」部分は、データファイルがCSVファイルなのかXMLファイルなのかによって読み込み時の処理が異なることになります。

そこで、これらの読み込み処理を同じクラスやメソッドとして作成することを考えてみましょう。どうでしょうか？あまり良い方法ではなさそうですね。

同じクラスやメソッドとして作成してしまうと、当然ですが質の異なる処理が混ざることになります。その結果、保守性や再利用性が損なわれることになります。たとえば、新しいデータフォーマットに対応するためには、そのクラスやメソッドそのものを修正する必要が発生してしまう、といった具合です。

「データファイルを読み込む」という処理は非常に一般的なものです。データフォーマットごとの処理をまとめられるとコードをシンプルに保つことができますし、他のアプリケーションでも再利用できそうです。

あとは、どうやってデータフォーマットごとの処理クラスを切り替えられるようにするか？

ここで、Strategyパターンの登場です。

Strategyパターンとは？

Strategy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。

> アルゴリズムの集合を定義し、各アルゴリズムをカプセル化して、それらを交換可能にする。
> Strategyパターンを利用することで、アルゴリズムを、それを利用するクライアントからは
> 独立に変更することができるようになる。

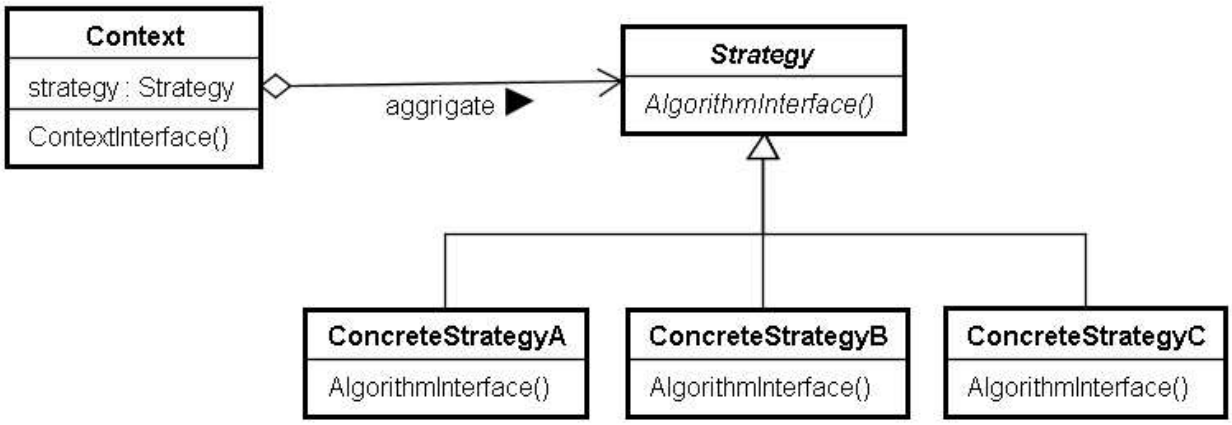
Strategy パターンは、オブジェクトの振る舞いに注目したパターンです。

Strategyパターンでは、それぞれの処理をクラスとして定義します。その際、クライアントにアクセスさせるための共通APIを用意しておくのがポイントです。これにより、処理クラスを利用する側は具体的な実装を意識することなく、共通のAPIで処理を実行できます。

また、処理の実行を処理クラスのオブジェクトに委譲することで、処理の切り替えができるようにしています。

Strategyパターンの構造

Strategyパターンのクラス図と構成要素は、次のとおりです。



Strategyクラス

それぞれの処理に共通のAPIを定義します。Contextクラスからは、Strategyクラスで定義されたAPIを通じて、ConcreteStrategyクラスで提供される具体的な処理を呼び出します。

ConcreteStrategyクラス

Strategyクラスのサブクラスで、Strategyクラスで定義されたAPIを実装したクラスです。このクラスに具体的な処理内容を記述します。

Contextクラス

Strategy型のオブジェクトを内部に保持し、具体的な処理をそのオブジェクトに委譲します。こうすることで、ConcreteStrategyクラスに依存することがなくなりますので、ConcreteStrategyクラスを切り替えることができます

Strategyパターンのメリット

Strategy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。

処理毎にまとめることができる

それぞれの処理がクラスにまとめられて実装されており、コードは処理内容に専念することができます。これにより、保守性が高まります。

また、新しい処理が追加された場合も、既存のコードに手を入れることなく、新しいクラスを作成するだけで済みます。

異なる処理を選択するための条件文がなくなる

1つのクラスやメソッドに異なる処理を記述した場合、if文やswitch文を使って処理を分岐することになります。これは、コードの可読性を落とすため、保守性・拡張性が下がります。Strategyパターンを適用すると、処理がクラス単位にまとめて実装されます。この結果、if文やswitch文を使うことがなくなり、非常にすっきりしたコードになります。

異なる処理を動的に切り替えることができる

クラス単位に処理がまとめて実装されているので、クライアントは使いたいConcreteStrategyクラスのインスタンスをContextオブジェクトに渡すだけで、処理を動的に切り替えることができます。

Strategyパターンの適用例

ここでStrategyパターンの適用例を見てみることにしましょう。

フォーマットの異なる商品データを読み込み、それを一覧表示するサンプルアプリケーションを用意しました。データファイルは固定長とタブ区切りの2種類で、データレイアウトと用意したデータファイルは、それぞれ次の通りです。

●固定長データ(fixed_length_data.txt)

```
商品名              商品番号  価格    発売日
限定Ｔシャツ        ABC0001   3800    20060311
ぬいぐるみ          ABC0002   1500    20051201
クッキーセット      ABC0003   800     20060520
```

項目	開始位置	終了位置	備考
商品名	1	20	
商品番号	21	30	
価格	31	38	
発売日	39		- YYYYMMDD形式

●タブ区切りデータ(tab_separated_data.txt)

```
商品番号 商品名　価格　発売日
ABC0001 限定Ｔシャツ　3800　2006/3/11
ABC0002 ぬいぐるみ　1500　2005/12/1
ABC0003 クッキーセット　800 2006/5/20
```

項目	備考
商品名	
商品番号	
価格	
発売日	YYYYMMDD形式

最初はReadItemDataStrategyクラスです。Strategyクラスに相当し、抽象クラスとして定義しています。

2つのメソッドgetDataとreadDataが定義されていますが、getDataメソッドの方はcontextクラスに相当するItemDataContextクラスに公開されるメソッドです。もう一方のgetItemDataメソッドは抽象メソッドとなっており、サブクラスで具体的な読み込み処理を実装します。

●ReadItemDataStrategyクラス(ReadItemDataStrategy.class.php)

```
<?php
/**
 * Strategyに相当する
 */
abstract class ReadItemDataStrategy {
```

```
    private $filename;

    /**
     * コンストラクタ
     */
    public function __construct($filename) {
        $this->filename = $filename;
    }

    /**
     * データファイルを読み込み、オブジェクトの配列で返す
     * Contextに提供するメソッド
     * @param string データファイル名
     * @return データオブジェクトの配列
     */
    public function getData() {
        if (!is_readable($this->getFilename())) {
            throw new Exception('file [' . $this->getFilename() . '] is not readable !');
        }

        return $this->readData($this->getFilename());
    }

    /**
     * ファイル名を返す
     * @return ファイル名
     */
    public function getFilename() {
        return $this->filename;
    }

    /**
     * ConcreteStrategyクラスに実装させるメソッド
     * @param string データファイル名
     * @return データオブジェクトの配列
     */
    protected abstract function readData($filename);
}
?>
```

次は、ReadItemDataStrategyクラスのサブクラスたちです。

先ほどの説明の通り、データ形式ごとにクラスを作成しています。ReadFixedLengthDataStrategyクラスは固定長データ、ReadTabSeparatedDataStrategyクラスはタブ区切りデータをそれぞれ読み込みます。なお、読み込むファイル名は、コンストラクタで指定します。

●ReadFixedLengthDataStrategyクラス（ReadFixedLengthDataStrategy.class.php）

```
<?php
require_once 'ReadItemDataStrategy.class.php';
?>
<?php
/**
 * 固定長データを読み込む
 * ConcreteStrategyに相当する
 */
class ReadFixedLengthDataStrategy extends ReadItemDataStrategy {

    /**
     * データファイルを読み込み、オブジェクトの配列で返す
     * @param string データファイル名
     * @return データオブジェクトの配列
     */
    protected function readData($filename) {
        $fp = fopen($filename, 'r');

        /**
         * ヘッダ行を抜く
         */
        $dummy = fgets($fp, 4096);

        /**
         * データの読み込み
         */
        $return_value = array();
        while ($buffer = fgets($fp, 4096)) {
            $item_name = trim(substr($buffer, 0, 20));
            $item_code = trim(substr($buffer, 20, 10));
            $price = (int)substr($buffer, 30, 8);
            $release_date = substr($buffer, 38);

            /**
             * 戻り値のオブジェクトの作成
             */
            $obj = new stdClass();
            $obj->item_name = $item_name;
            $obj->item_code = $item_code;
```

```
            $obj->price = $price;

            $release_date_arr = strptime($release_date, '%Y%m%d');
            $obj->release_date = mktime(0, 0, 0,
                                        $release_date_arr['tm_mon'],
                                        $release_date_arr['tm_mday'],
                                        $release_date_arr['tm_year']);

            $return_value[] = $obj;
        }

        fclose($fp);

        return $return_value;
    }
}
?>
```

●ReadTabSeparatedDataStrategyクラス(ReadTabSeparatedDataStrategy.class.php)

```
<?php
require_once 'ReadItemDataStrategy.class.php';
?>
<?php
/**
 * タブ区切りデータを読み込む
 * ConcreteStrategyに相当する
 */
class ReadTabSeparatedDataStrategy extends ReadItemDataStrategy {

    /**
     * データファイルを読み込み、オブジェクトの配列で返す
     * @param string データファイル名
     * @return データオブジェクトの配列
     */
    protected function readData($filename) {
        $fp = fopen($filename, 'r');

        /**
         * ヘッダ行を抜く
         */
        $dummy = fgets($fp, 4096);

        /**
         * データの読み込み
         */
        $return_value = array();
        while ($buffer = fgets($fp, 4096)) {
            list($item_code, $item_name, $price, $release_date) = split("¥t", $buffer);

            /**
             * 戻り値のオブジェクトの作成
             */
            $obj = new stdClass();
            $obj->item_name = $item_name;
            $obj->item_code = $item_code;
            $obj->price = $price;

            $release_date_arr = strptime($release_date, '%Y/%m/%d');
            $obj->release_date = mktime(0, 0, 0,
                                        $release_date_arr['tm_mon'],
                                        $release_date_arr['tm_mday'],
                                        $release_date_arr['tm_year']);

            $return_value[] = $obj;
        }

        fclose($fp);

        return $return_value;
    }
}
?>
```

続いて、Contextクラスに相当するクラス、ItemDataContextクラスを見てみましょう。

ItemDataContextクラスの特徴は、コンストラクタにReadItemDataStrategy型のオブジェクトを受け取り、getItemDataメソッドで具体的な処理を委譲している部分です。つまり、コンストラクタに引き渡すReadItemDataStrategy型のオブジェクトを変更するだけで、getItemDataメソッドの動作を変更できます。

●ItemDataContextクラス(ItemDataContext.class.php)

```php
<?php
/**
 * Contextに相当する
 */
class ItemDataContext {

    private $strategy;

    /**
     * コンストラクタ
     * @param ReadItemDataStrategy ReadItemDataStrategyオブジェクト
     */
    public function __construct(ReadItemDataStrategy $strategy) {
        $this->strategy = $strategy;
    }

    /**
     * 商品情報をオブジェクトの配列で返す
     * @return データオブジェクトの配列
     */
    public function getItemData() {
        return $this->strategy->getData();
    }

}
?>
```

最後にクライアントのコードを確認しておきましょう。

単純にそれぞれのデータファイルを読み込んで一覧表示する処理をおこなっているだけですが、いかがでしょうか？実際にインスタンス化しているReadItemDataStrategyオブジェクトの指定以外は全く同じですね。

●クライアント側コード(strategy_client.php)

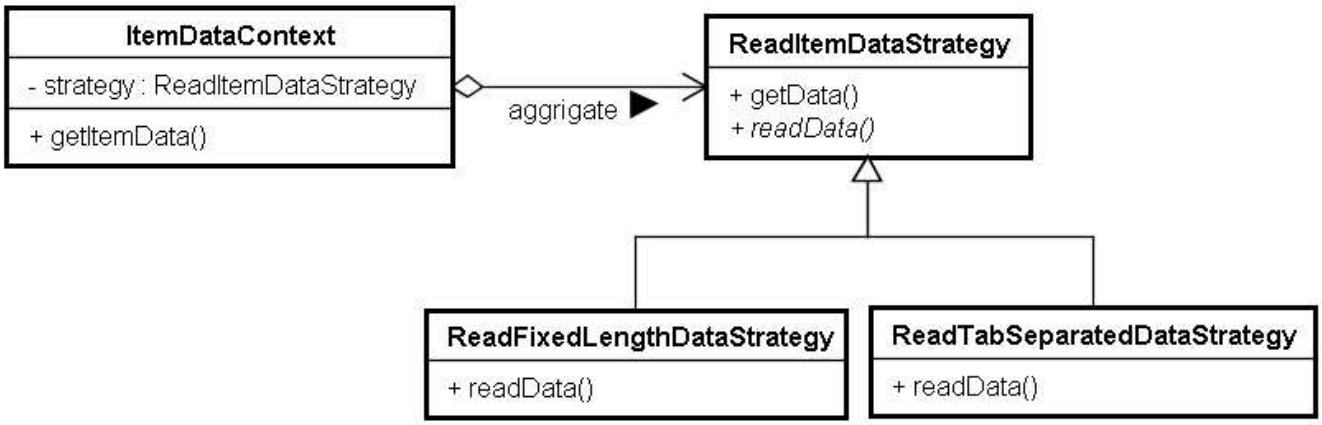
```php
<?php
require_once 'ItemDataContext.class.php';
require_once 'ItemDataContextByName.class.php';
require_once 'ReadFixedLengthDataStrategy.class.php';
require_once 'ReadTabSeparatedDataStrategy.class.php';
?>
<?php
function dumpData($data) {
    echo '<dl>';
    foreach ($data as $object) {
        echo '<dt>' . $object->item_name . '</dt>';
        echo '<dd>商品番号：' . $object->item_code . '</dd>';
        echo '<dd>¥¥' . number_format($object->price) . '-</dd>';
        echo '<dd>' . date('Y/m/d', $object->release_date) . '発売</dd>';
    }
    echo '</dl>';
}
?>
<?php
    /**
     * 固定長データを読み込む
     */
    $strategy1 = new ReadFixedLengthDataStrategy('fixed_length_data.txt');
    $context1 = new ItemDataContext($strategy1);
    dumpData($context1->getItemData());

    echo '<hr>';

    /**
     * タブ区切りデータを読み込む
     */
    $strategy2 = new ReadTabSeparatedDataStrategy('tab_separated_data.txt');
    $context2 = new ItemDataContext($strategy2);
    dumpData($context2->getItemData());

?>
```

また、サンプルコードのクラス図は次のようになります。



Strategyパターンのオブジェクト指向的要素

Strategyパターンは「継承」と「ポリモーフィズム」を活用しているパターンです。

StrategyクラスとConcreteStrategyクラスは、継承の関係にあります。親クラスであるStrategyクラスで処理内容が変わる部分を抽象メソッドとして定義します。一方、サブクラスであるConcreteStrategyクラスでは、抽象メソッドを実装し、具体的な処理を記述します。こうすることで、同じAPIを持ち、かつ具体的な処理が異なるクラス群を用意できます。

また、Contextクラスは、Strategy型のインスタンスを内部に保持します。このインスタンスは、具体的にはStrategyクラスを継承したサブクラスのインスタンスです。Contextクラスは、クライアントからの処理要求を受け取ると、保持したインスタンスに具体的な処理を委譲します。この時、処理を委譲する部分を、処理側の親クラスであるStrategyクラスのAPIだけを使ってプログラミングを行っておくことがポイントです。こうすることで、Strategy型のインスタンスがどの様な処理を行うものであれ、正しく動作することになります。

この結果、ConcreteStrategyクラスを簡単に差し替えたり、追加したりできるのです。Strategyパターンは、委譲を使って処理内容全体を切り替えるパターンと言えます。

なお、このような処理を切り替えるパターンとしては、Strategyパターン以外にTemplate Methodパターンがあります。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ています。

関連するパターン

Flyweightパターン

ConcreteStrategyクラスのインスタンスは、Flyweightパターンを使って共有できる場合があります。

まとめ

ここではアルゴリズムをクラスにまとめ、そのアルゴリズムごとに切り替えできるようにするStrategyパターンを説明しました。