# Chain of Responsibility～処理のたらい回し

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Chain of Responsibilityパターンについて説明します。

「Chain of Responsibility」とは長い名前ですね。直訳すると、「責任の鎖」となるでしょうか。それぞれのクラスは「責任」を持っています。その責任を明確にするよう設計をおこなうことが、オブジェクト指向設計では大きなポイントとなります。

Chain of Responsibilityパターンは、自分の責任で対処する必要かどうかを判断し、自分で対処できない場合は他に任せてしまうパターンです。まるで、責任のたらい回しみたいですね。

では、早速見ていきましょう。

## たとえば

条件によっておこなう処理を分岐させることはよくあることですね。たいていの場合、if文やswitch文を使って「この条件の場合はこう処理する」というコードを記述することになります。

ここで、入力された文字列を検証する場合を考えてみましょう。

入力値の検証には、文字列の長さのチェックやあるフォーマットに合っているかどうかのチェックがありますね。

一般的な検証処理の流れとしては、入力された文字列が所定のパターンにマッチするかどうかを判定し、マッチしない場合は適切なメッセージを表示する、というものになるかと思います。

この処理をif文を使って実装する場合、マッチングをおこなうパターンの数だけif文を繋げることになるでしょう。しかし、パターンの数が多い場合やマッチングの条件が複雑な場合、コードの見通しが非常に悪くなってしまいがちです。また、コードの再利用がしにくい状態になります。

if文やswitch文を使った条件分岐は、「どの場合にどう処理をすべきか」が全て1カ所にまとめられることになります。つまり、条件とそれに対応する処理の組を知っておく必要がある、ということです。このため、条件が複雑になったり分岐の数が多くなればなるほど、「知っておかなければならないこと」が増えてしまいます。

この場合、分岐の条件とそれに対応する処理の組ごとに分解できると、組ごとにその条件や処理内容に集中することができます。その結果、コードの見通しも良くなり、保守性や再利用性を高めることができそうです。

しかし、条件と処理の組に分解した場合、それらをどうやって組み立てて利用するかが問題になってしまいますね。分解してしまった分、扱いが大変になってしまうと分解した意味がありません。

こうした問題を解決するためのパターンとして、**Chain of Responsibilityパターン**があります。

## Chain of Responsibilityパターンとは？

Chain of Responsibility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。

> 1つ委譲のオブジェクトに要求を処理する機会を与えることにより、
> 要求を送信するオブジェクトと受信するオブジェクトの結合を避ける。
> 受信する複数のオブジェクトをチェーン状につなぎ、あるオブジェクトがその要求を
> 処理するまで、そのチェーンに沿って要求を渡していく。

Chain of Responsibilityパターンは、「処理を依頼する」側と「処理をおこなう」側を分離するパターンです。

まず、Chain of Responsibilityパターンの特徴である処理をおこなう「オブジェクトのチェーン」について説明しましょう。

このオブジェクトは、処理をおこなうための**共通のAPI**を持ち、それぞれ異なる処理を実装しています。この処理はif文などを使った場合に記述する「この条件の場合におこなう処理」になります。

また、内部に別の処理をおこなうオブジェクトを保持していて、自分が処理できないと判断した場合、そのオブジェクトに処理をお願いします。何だかバケツリレーに似ていますね。「要求」が入ったバケツを受け取り、自分自身が処理できない場合、次の人に渡して処理をお願いする、といった感じです。

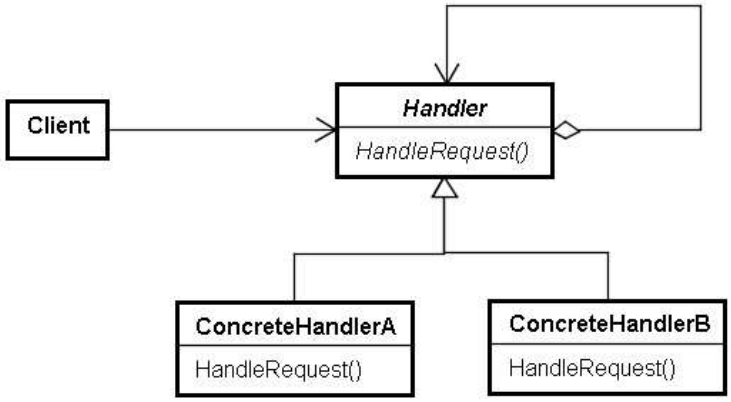
一方、クライアント側は、処理オブジェクトのチェーンに要求を送信するだけです。あとは、その要求が処理オブジェクトのチェーンを伝わっていき、適切に処理可能なオブジェクトが処理を行います。これが、「実際に処理を実行するオブジェクトを動的に決定する」ということです。

では、どの処理オブジェクトも処理できないことはないのでしょうか？

残念ながら、処理できない場合も当然あります。これはチェーンが正しく作成されていない場合も同様で、チェーンの終端で要求が消滅してしまう、といったことも起こり得ます。つまり、処理オブジェクトのチェーンで必ず処理されるわけではない事に、注意が必要です。

## Chain of Responsibilityパターンの構造

Chain of Responsibilityパターンのクラス図と構成要素は、次のとおりです。




**Handlerクラス**

処理オブジェクトの親クラスに相当し、要求を処理するためのAPIを定義します。これは、サブクラスで具体的な処理が実装されます。

また、内部にHandler型のオブジェクトを保持します。自分が処理ができなかった場合に処理をお願いする先になります。

**ConcreteHandlerクラス**

Handlerクラスのサブクラスです。処理オブジェクトの実クラスに相当します。このクラスは、Handlerクラスで定義されたAPIを実装します。なお、自分が担当する処理だけが実装されます。

**Clientクラス**

チェーンを構成しているConcreteHandlerクラスに処理を送信します。

## Chain of Responsibilityパターンのメリット

Chain of Responsibility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。

### 要求の送信側と受信側の結びつきを緩くする

Chain of Responsibilityパターンでは、要求の送信側(Clientクラス)で「どのオブジェクトに処理を行わせるか」ということを意識する必要がありません。「要求が適切に処理される」という事だけを知っていれば良いことになります。

結果として、Chain of Responsibilityパターンは、オブジェクトどうしの結びつきを緩めることができます。

### 新しい処理クラスを簡単に追加できる

要求の送信側と受信側の結びつきが緩くなるため、新しい処理クラス(**ConcreteHandlerクラス**)を追加するのが非常に簡単です。

### 動的に処理チェーンを変更できる

処理オブジェクトのチェーンは、オブジェクトを繋げたものです。つまり、継承関係のようにプログラミング時に関係が決まるような静的な関連はありません。また、すべての処理オブジェクトは、具体的にはHandler型のオブジェクトです。このため、実行時にオブジェクトを抜き差しすることで、チェーンを動的に変更できます。

たとえば、ユーザー操作によって処理を変更する場合でも、処理オブジェクトを組み替えたり、追加したりできます。

## Chain of Responsibilityパターンの適用例

Chain of Responsibilityパターンの適用例を見てみましょう。

ここでは、先に出てきた入力された文字列の検証にChain of Responsibilityパターンを適用した例になります。

まずは「条件とそれに対応する処理の組」に関連するクラスから見ていきます。

ValidationHandlerクラスはHandlerクラスに相当するクラスです。ここでは、抽象クラスとして定義しています。

また、実際の検証処理をおこなうexecValidationメソッドとエラーメッセージを取り出すgetErrorMessageメソッドは抽象メソッドになっています。この2つのメソッドが「条件判断」と「対応する処理」をおこなうメソッドになっており、このValidationHandlerクラスを継承したクラスで具体的な実装をおこなうことになります。

●ValidationHandlerクラス(ValidationHandler.class.php)

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

    private $next_handler;

    public function __construct() {
        $this->next_handler = null;
    }

    public function setHandler(ValidationHandler $handler) {
        $this->next_handler = $handler;
        return $this;
    }

    public function getNextHandler() {
        return $this->next_handler;
    }

    /**
     * チェーンの実行
     */
    public function validate($input) {
        $result = $this->execValidation($input);
        if (!$result) {
```

```
            return $this->getErrorMessage();
        } else if (!is_null($this->getNextHandler())) {
            return $this->getNextHandler()->validate($input);
        } else {
            return true;
        }
    }

    /**
     * 自クラスが担当する処理を実行
     */
    protected abstract function execValidation($input);

    /**
     * 処理失敗時のメッセージを取得する
     */
    protected abstract function getErrorMessage();
}
?>
```

このクラスで注目するのはvalidateメソッドです。このメソッドが、処理オブジェクトの鎖にクライアントからの要求を流す役割を果たします。

具体的には、execValidationメソッドを呼び出して実際に処理をおこない、処理できたかどうかを判断します。成功した場合は、内部に保持したValidationHandlerオブジェクトを取り出し、次の検証処理をおこないます。失敗した場合は、エラーメッセージをクライアントに返します。

最終的に全てのValidationHandlerオブジェクトで検証をおこない、全ての処理に成功した場合はtrueを返します。

次にValidationHandlerクラスを継承したクラスたちを見ていきましょう。ここでは検証のパターンとして4つほど用意しています。

まず、AlphabetValidationHandlerクラスとNumberValidationHandlerクラスです。このクラスたちは、名前の通り入力された文字列がアルファベット、もしくは数字だけで構成されているかどうかを検証します。指定された文字以外で構成されている場合、検証失敗となります。処理の詳細は、それぞれのクラスのexecValidationメソッドとgetErrorMessageメソッドを確認してください。

●AlphabetValidationHandlerクラス(AlphabetValidationHandler.class.php)

```
<?php
require_once 'ValidationHandler.class.php';
?>
<?php
/**
 * ConcreteHandlerクラスに相当する
 */
class AlphabetValidationHandler extends ValidationHandler {

    /**
     * 自クラスが担当する処理を実行
     */
    protected function execValidation($input) {
        return preg_match('/^[a-z]*$/i', $input);
    }

    /**
     * 処理失敗時のメッセージを取得する
     */
    protected function getErrorMessage() {
        return '半角英字で入力してください';
    }
}
?>
```

●NumberValidationHandlerクラス(NumberValidationHandler.class.php)

```
<?php
require_once 'ValidationHandler.class.php';
?>
<?php
/**
 * ConcreteHandlerクラスに相当する
 */
class NumberValidationHandler extends ValidationHandler {

    /**
     * 自クラスが担当する処理を実行
     */
    protected function execValidation($input) {
        return (preg_match('/^[0-9]*$/', $input) > 0);
    }

    /**
     * 処理失敗時のメッセージを取得する
     */
    protected function getErrorMessage() {
        return '半角数字で入力してください';
    }
}
?>
```

NotNullValidationHandlerクラスは、入力された文字列が空文字でないかどうかを検証します。空文字の場合、検証失敗となります。

●NotNullValidationHandlerクラス(NotNullValidationHandler.class.php)

```
<?php
```

```
require_once 'ValidationHandler.class.php';
?>
<?php
/**
 * ConcreteHandlerクラスに相当する
 */
class NotNullValidationHandler extends ValidationHandler {

    /**
     * 自クラスが担当する処理を実行
     */
    protected function execValidation($input) {
        return (is_string($input) && $input !== '');
    }

    /**
     * 処理失敗時のメッセージを取得する
     */
    protected function getErrorMessage() {
        return '入力されていません';
    }
}
?>
```

ValidationHandlerクラスのサブクラスの最後はMaxLengthValidationHandlerクラスです。このクラスは、入力された文字列の長さが指定された長さ以下かどうかを検証します。この長さの指定は、コンストラクタでおこなっています。

●MaxLengthValidationHandlerクラス(MaxLengthValidationHandler.class.php)

```
<?php
require_once 'ValidationHandler.class.php';
?>
<?php
/**
 * ConcreteHandlerクラスに相当する
 */
class MaxLengthValidationHandler extends ValidationHandler {

    private $max_length;

    public function __construct($max_length = 10) {
        parent::__construct();
        if (preg_match('/^[0-9]{,2}$/', $max_length)) {
            throw new RuntimeException('max length is invalid (0-99) !');
        }
        $this->max_length = (int)$max_length;
    }

    /**
     * 自クラスが担当する処理を実行
     */
    protected function execValidation($input) {
        return (strlen($input) <= $this->max_length);
    }

    /**
     * 処理失敗時のメッセージを取得する
     */
    protected function getErrorMessage() {
        return $this->max_length . 'バイト以内で入力してください';
    }
}
?>
```

そして、検証のクラス群を利用するクライアント側のコードです。動作を簡単に確認できるよう、入力用のHTMLフォームも表示します。

また、検証を実行するコードがValidationHandler型オブジェクトのvalidateメソッドを呼び出すだけになっていることを確認してください。if文で実装する場合と比べて、非常に簡単なコードになっていますね。

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

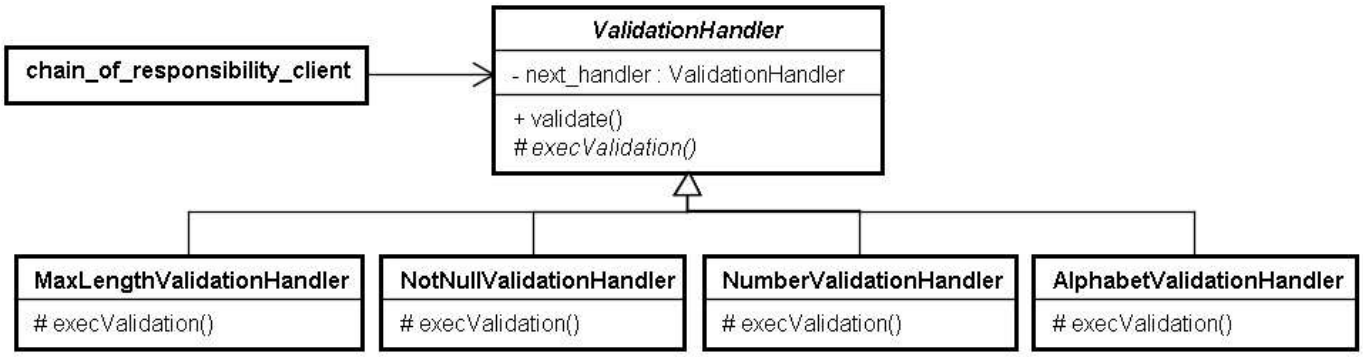
```
<?php
require_once 'MaxLengthValidationHandler.class.php';
require_once 'NotNullValidationHandler.class.php';
?>
<?php
    if (isset($_POST['validate_type']) && isset($_POST['input'])) {
        $validate_type = $_POST['validate_type'];
        $input = $_POST['input'];

        /**
         * チェーンの作成
         * validate_typeの値によってチェーンを動的に変更
         */
        $not_null_handler = new NotNullValidationHandler();
        $length_handler = new MaxLengthValidationHandler(8);

        $option_handler = null;
        switch ($validate_type) {
        case 1:
            include_once 'AlphabetValidationHandler.class.php';
            $option_handler = new AlphabetValidationHandler();
            break;
        case 2:
            include_once 'NumberValidationHandler.class.php';
            $option_handler = new NumberValidationHandler();
```

```
            break;
        }

        if (!is_null($option_handler)) {
            $length_handler->setHandler($option_handler);
        }
        $handler = $not_null_handler->setHandler($length_handler);

        /**
         * 処理実行と結果メッセージの表示
         */
        $result = $handler->validate($_POST['input']);
        if ($result === false) {
            echo '検証できませんでした';
        } else if (is_string($result) && $result !== '') {
            echo '<p style="color: #dd0000;">' . $result . '</p>';
        } else {
            echo '<p style="color: #008800;">OK</p>';
        }
    }
?>
<form action="" method="post">
  <div>
    値：<input type="text" name="input">
  </div>
  <div>
    検証内容：<select name="validate_type">
    <option value="0">任意</option>
    <option value="1">半角英字で入力されているか</option>
    <option value="2">半角数字で入力されているか</option>
    </select>
  </div>
  <div>
    <input type="submit">
  </div>
</form>
```

入力フォームのプルダウンで検証する内容を選択できるようになっていますが、これによってAlphabetValidationHandlerオブジェクトもしくはNumberValidationHandlerオブジェクトが生成され、検証オブジェクトの鎖に追加されます。

お分かりのように、検証のための処理を動的に追加しています。処理チェーンを動的に変更することができるのは、Chain of Responsibilityパターンの大きな特徴です。また、新しい検証クラスを作成し追加する場合も容易に対応できることが分かると思います。

最後に、Chain of Responsibilityパターンを適用したサンプルアプリケーションのクラス図を示します。




## Chain of Responsibilityパターンのオブジェクト指向的要素

Chain of Responsibilityパターンは「ポリモーフィズム」を活用したパターンです。

Chain of Responsibilityパターンの特徴は、処理オブジェクトのチェーンです。つまり、Handler型のオブジェクトのチェーンです。このチェーンは、Handlerクラスの内部に保持されたHandler型のオブジェクトです。実際には、Handlerクラスのサブクラスであるコンクリートハンドラクラスのインスタンスですが、Handlerクラス自身からはこのオブジェクトが具体的にどのクラスなのかは意識していません。ただ、Handler型のオブジェクトであるというだけです。

つまり、内部に保持されたオブジェクトは、具体的にどのようなクラスであれ、Handler型のオブジェクト、言い換えると、**Handlerクラスのサブクラスのインスタンスであれば、問題なく動作する**ということになります。

この結果、チェーンの組み替えや、新しいConcreteHandlerクラスを追加したりできるのです。

## 関連するパターン

**Compositeパターン**

CompositeパターンはChain of Responsibilityパターンと併用される場合があります。

## まとめ

ここでは、オブジェクトを鎖のように繋いで問題を対処するChain of Responsibilityパターンを見てきました。