Command～要求をクラスで表す

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

振る舞い＋オブジェクト

はじめに

ここではCommandパターンについて説明します。

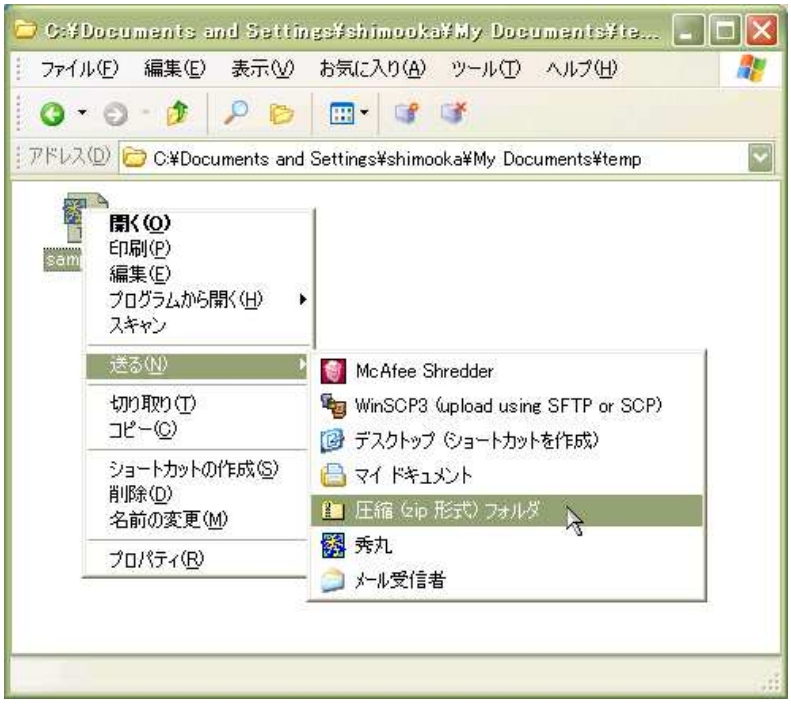
commandという単語は「命令」という意味ですね。「コマンド」と書くと、DOSプロンプトやUNIX、Linuxのコンソールで入力するコマンドを連想される方も多いかと思います。

Commandパターンはその名の通り「命令」そのものをクラスとして表すパターンです。しかし、「命令」をクラスにするとどの様なことになるのでしょうか？

たとえば

ほとんどのアプリケーションでは、利用者はアプリケーションに何らかの要求を出し、アプリケーションは要求を受け取って処理を実行しています。先に出てきたDOSやLinuxなどのコマンドもその1つですし、WindowsやX WindowといったGUIでの操作も含まれます。

たとえば、ファイルを圧縮する場合を考えてみましょう。Windowsの場合、圧縮ソフトにファイルをドラッグアンドドロップするといった操作になりますし、Linuxなどのコンソールから実行する場合、圧縮コマンドにファイル名を指定して実行するでしょう。



この時、利用者はソフトウェアやコマンドを通じて「圧縮する」という要求を出し、対象のファイルがその要求を受け取っている、と考えることができます。また、ファイルに別の操作をおこなう、つまり別な要求を出す場合は、「処理をおこなうソフトやコマンド」を差し替えたものと考えられるでしょう。

ここで、「要求を送る」ということは、オブジェクト指向プログラミングの場合、オブジェクトのメソッドを呼び出すということになりますが、要求が複雑になったり要求の種類が多くなると、その実装にも限界が出てきますし保守性も悪くなってしまいます。

そこで、「要求を送る」「要求を受け取る」という考えに基づいて、「要求」自身をクラスとしてまとめてしまうとどうでしょうか？

そうすると、具体的な要求を1つのオブジェクトとして扱うことができ、複雑な要求の場合でも容易に扱えるようになります。また、送る要求オブジェクトを変えることで、異なる要求を実現こともできそうです。

Commandパターンは、このような特徴を持っているパターンです。

Commandパターンとは？

Commandパターンは、GoF本では以下のように定義されています。

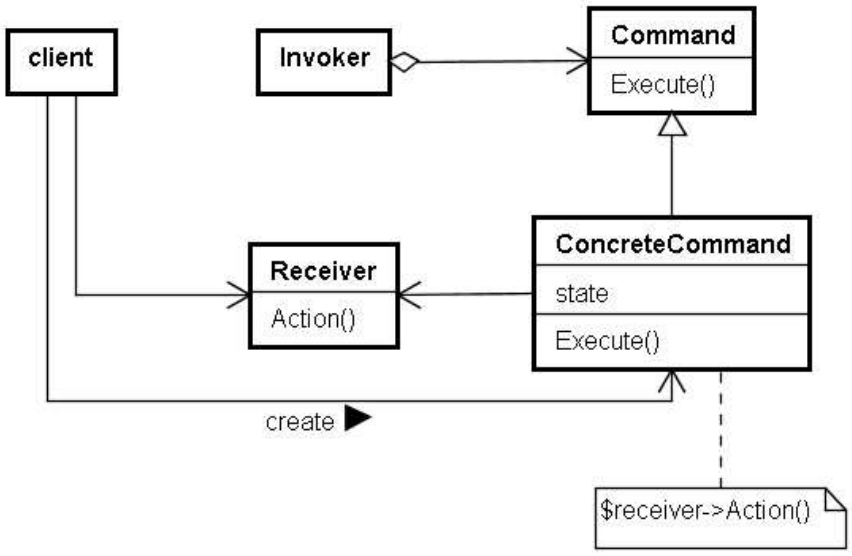
> 要求をオブジェクトとしてカプセル化することによって、
> 様々な要求または要求からなるキューやログによりクライアントを
> パラメータ化する。そして、取り消し可能な操作をサポートする。

Command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。

Commandパターンでは、異なる種類の要求に対する処理を同じ**API**を持つクラスとして実装します。その結果、処理クラスのインスタンスを切り替えるだけで、様々な要求に対する処理を実行できるようなります。また、Commandパターンを適用すると、新しい要求に対する処理クラスを実装するだけで、既存のクラスを修正することなく対応可能になります。

Commandパターンの構造

Commandパターンのクラス図と構成要素は、次のようになります。



Commandクラス

命令を実行するためのAPIを定義します

ConcreteCommandクラス

Commandクラスのサブクラスで、Commandクラスで定義されたAPIを実装します。

Invokerクラス

命令実行の要求を出すクラスです。

Receiverクラス

命令をどの様に実行するかを知っている唯一のクラスです。任意のクラスがReceiverクラスになることができます。

Commandパターンのメリット

Commandパターンのメリットとしては、次のものが挙げられます。

既存のコードを修正することなく機能拡張できる

Commandパターンを適用すると、要求の受付と要求に対応する処理を切り離して実装できます。その結果、新しい要求が追加された場合でも既存のクラスを修正する必要がなく、追加された要求を処理するためのクラスを実装するだけで済みます。

クラスの再利用性を向上させる

命令そのものが独立したクラスとして実装されますので、他のアプリケーションでの再利用がしやすくなります。

処理のキューイング

要求と実際の実行を別のタイミングで実施することができるようになります。

UndoやRedoのサポート

Commandクラスに実行したコマンド結果を保持しておくことで、Undo機能やRedo機能を実現することができます。

Commandパターンの適用例

早速、Commandパターンを適用してみましょう。

ここでは、ファイルの作成・圧縮・コピーをおこなうアプリケーションを用意しました。Commandパターンを適用し、ファイルに対する操作をコマンドとして定義しています。

まずは、Fileクラスから見ていきます。FileクラスはReceiverクラスに相当するクラスです。作成・圧縮・コピーそれぞれの処理に対するメソッドがありますが、今回はメッセージを表示するだけの実装としています。

●Fileクラス（File.class.php）

```
<?php
/**
 * Receiverクラスに相当する
 */
class File {
    private $name;
    public function __construct($name) {
        $this->name = $name;
    }
    public function getName() {
```

```
        return $this->name;
    }
    public function decompress() {
        echo $this->name . 'を展開しました<br>';
    }
    public function compress() {
        echo $this->name . 'を圧縮しました<br>';
    }
    public function create() {
        echo $this->name . 'を作成しました<br>';
    }
}
?>
```

続いて、要求を処理する側のクラス群を見ていきましょう。

Commandインターフェースは、すべてのコマンドに共通のAPIであるexecuteメソッドを宣言しています。

●Commandインターフェース(Command.class.php)

```
<?php
/**
 * Commandクラスに相当する
 */
interface Command {
    public function execute();
}
?>
```

このCommandインターフェースを実装したクラスが、TouchCommandクラス、CompressCommandクラス、CopyCommandクラスです。ConcreteCommandクラスに相当し、Commandインターフェースで宣言されたexecuteメソッドを実装しています。

これらのクラスは、コンストラクタでFileオブジェクトを受け取り、executeメソッドでそれぞれの要求に対する処理をおこないますが、具体的な処理は受け取ったFileオブジェクトに任せています。

●TouchCommandクラス(TouchCommand.class.php)

```
<?php
require_once 'Command.class.php';
require_once 'File.class.php';
?>
<?php
/**
 * ConcreteCommandクラスに相当する
 */
class TouchCommand implements Command {
    private $file;
    public function __construct(File $file) {
        $this->file = $file;
    }
    public function execute() {
        $this->file->create();
    }
}
?>
```

●CompressCommandクラス(CompressCommand.class.php)

```
<?php
require_once 'Command.class.php';
require_once 'File.class.php';
?>
<?php
/**
 * ConcreteCommandクラスに相当する
 */
class CompressCommand implements Command {
    private $file;
    public function __construct(File $file) {
        $this->file = $file;
    }
    public function execute() {
        $this->file->compress();
    }
}
?>
```

●CopyCommandクラス(CopyCommand.class.php)

```php
<?php
require_once 'Command.class.php';
require_once 'File.class.php';
?>
<?php
/**
 * ConcreteCommandクラスに相当する
 */
class CopyCommand implements Command {
    private $file;
    public function __construct(File $file) {
        $this->file = $file;
    }
    public function execute() {
        $file = new File('copy_of_' . $this->file->getName());
        $file->create();
    }
}
?>
```

QueueクラスはCommandオブジェクトを保持するInvokerクラスに相当するクラスで、実際にCommandオブジェクトを実行します。runメソッドを呼び出すと、内部に保持したCommandオブジェクトを順に実行します。

●Queueクラス(Queue.class.php)

```php
<?php
require_once 'Command.class.php';
?>
<?php
/**
 * Invokerクラスに相当する
 */
class Queue {
    private $commands;
    private $current_index;
    public function __construct() {
        $this->commands = array();
        $this->current_index = 0;
    }
    public function addCommand(Command $command) {
        $this->commands[] = $command;
    }

    public function run() {
        while (!is_null($command = $this->next())) {
            $command->execute();
        }
    }

    private function next() {
        if (count($this->commands) === 0 ||
            count($this->commands) <= $this->current_index) {
            return null;
        } else {
            return $this->commands[$this->current_index++];
        }
    }
}
?>
```

次は、これまで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。

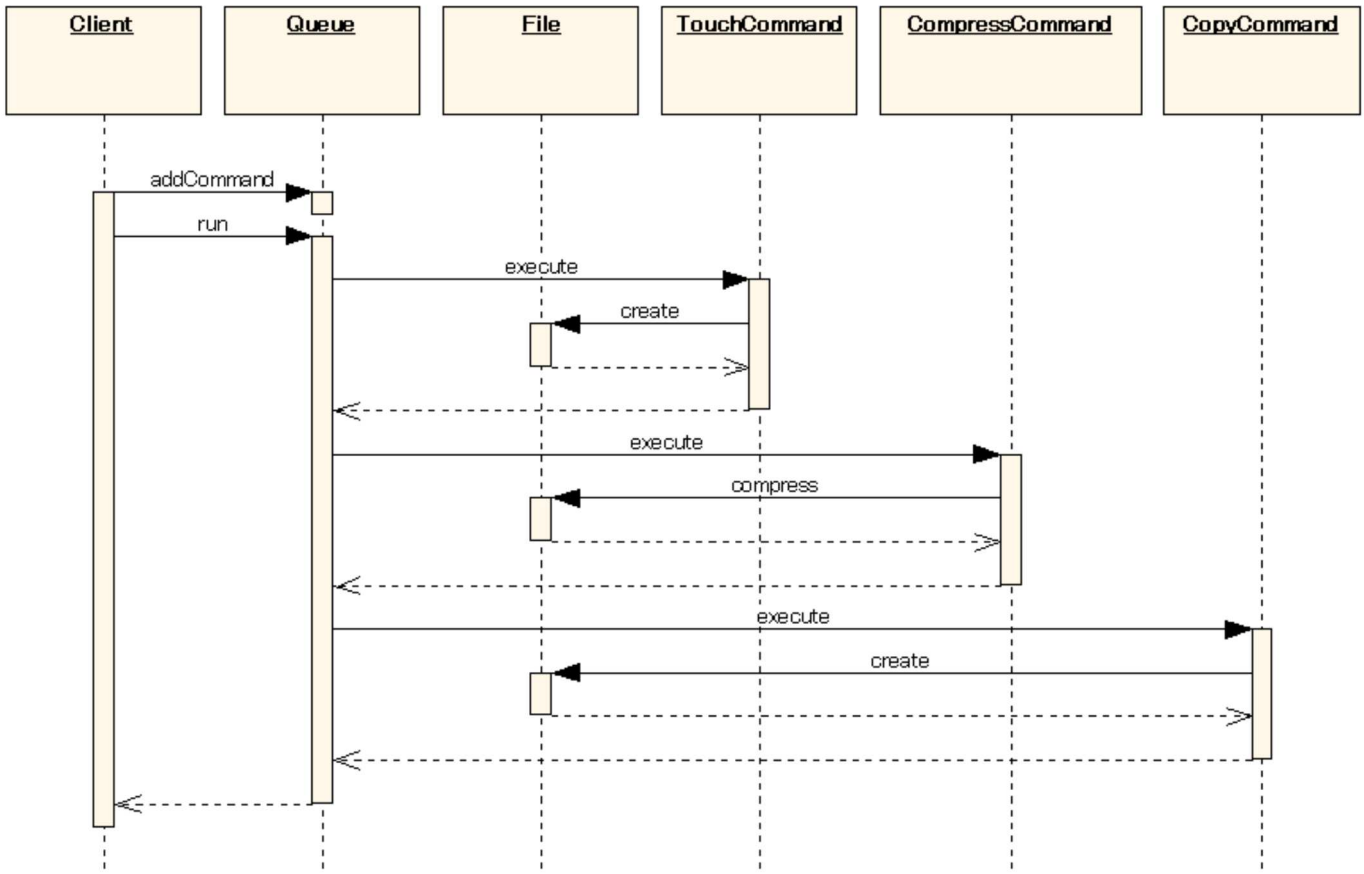
まず、Queueオブジェクトを生成した後、addCommandメソッドを使ってファイルに対する要求を表すCommandオブジェクトを追加しています。そして、最後に追加したCommandオブジェクトを実行しています。

●クライアント側コード(command_client.php)

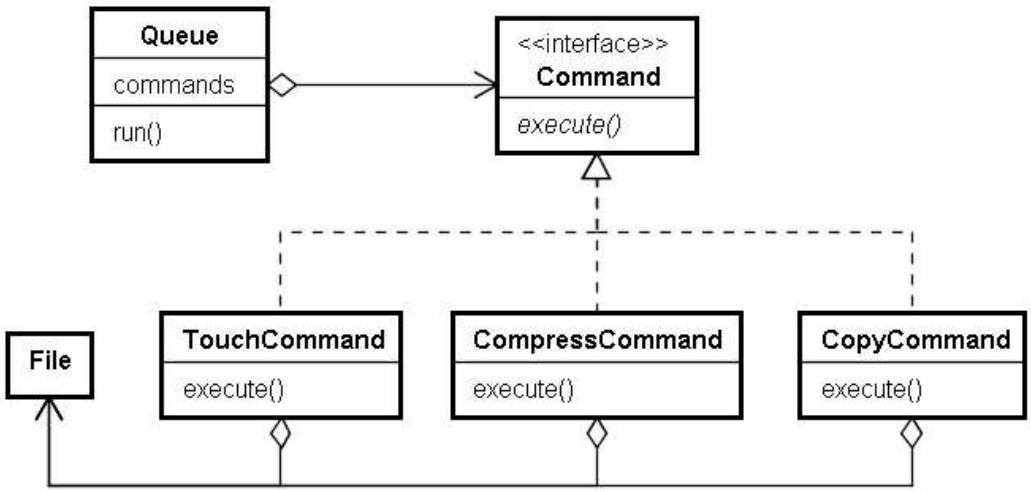
```php
<?php
require_once 'Queue.class.php';
require_once 'TouchCommand.class.php';
require_once 'CompressCommand.class.php';
require_once 'CopyCommand.class.php';
require_once 'File.class.php';
?>
<?php
    $queue = new Queue();
    $file = new File("sample.txt");
    $queue->addCommand(new TouchCommand($file));
    $queue->addCommand(new CompressCommand($file));
    $queue->addCommand(new CopyCommand($file));

    $queue->run();
?>
```

ここで、それぞれの要求が実行される様子を確認しておきましょう。

Invokerクラスに相当するQueueオブジェクトが各ConcreteCommandオブジェクトを実行し、要求の受け取り側のFileオブジェクトにアクセスしている様子が分かりますね。



最後に、このサンプルアプリケーションのクラス図となります。



Commandパターンのオブジェクト指向的要素

Commandパターンは、「ポリモーフィズム」を利用したパターンになります。

具体的な「要求」は、Commandクラスのサブクラスであるコンクリートコマンドクラスで表されます。また、ConcreteCommandクラスではCommandクラスで定義されたAPIを具体的に実装しています。

また、Invokerクラスは内部にCommand型のオブジェクトを保持し、それらを実行します。この時、これらのオブジェクトはあくまでCommand型として扱われています。つまり、具体的な**ConcreteCommand**クラスに依存していないのです。

多くのデザインパターンでもポリモーフィズムを使って具体的なクラスを差し替え可能になっています(「交換可能性」といいます)。Commandパターンでは、変化する可能性がある「要求」にポリモーフィズムを使うことで、異なる要求を容易に差し替えられるようにしています。

関連するパターン

Compositeパターン

複数のコマンドをまとめたマクロコマンドを実現するために利用されます。

Mementoパターン

コマンドの実行履歴を管理するために、Mementoパターンが利用されることがあります。

Prototypeパターン

Commandクラスを複製したい場合にPrototypeパターンが利用されることがあります。

まとめ

ここでは「要求」そのものをクラスとして表し、「要求を送る側」と「要求を受け取る側」を分離するCommandパターンについて見てきました。