


## State ～状態を表す

 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Stateパターンについて見ていきましょう。

stateという単語は「状態」の意味がありますが、Stateパターンは物ではなく「状態」をクラスとして表現し、「状態」ごとに振る舞いを切り替えられるようにするパターンです。

### たとえば

たとえば、部屋の照明を考えてみましょう。照明には、点灯している状態(オン)と消灯している状態(オフ)の2つの状態があることになります。照明の状態がオンの場合、当然ですが照明が灯っている、つまり「明かりが灯る」という動作をしていると言えます。逆にオフの場合は「明かりが消える」という動作をしているということになります。このように「状態」によって振る舞いが変わるものはよくあります。

アプリケーションでも何らかの状態によって振る舞いが変わるものがあります。

たとえば、認証機能を持つアプリケーションの場合、認証済みの状態と認証していない状態が存在します。また、その状態によってメニュー表示を変えたり特定の機能の動作を変えるなどをおこなうことがあります。

このような場合、状態に依存する処理をどのように記述すれば良いのでしょうか？おそらく、if文やswitch文を使って実装する場面が多いと思います。先の認証機能の例ですと状態は2種類しかありませんのでさほど問題にはならないかも知れません。しかし、状態の種類が多くなるとコードの可読性が落ち、保守性・拡張性を落とす原因になってしまいます。

また、状態に関するコードがあちこちに散らばってしまうことになります。このため、新しい状態を追加することが困難になります。

**Stateパターン**は、「状態」と「状態による振る舞い」を1つのクラスにまとめることで、先のような問題を解決します。

### Stateパターンとは？

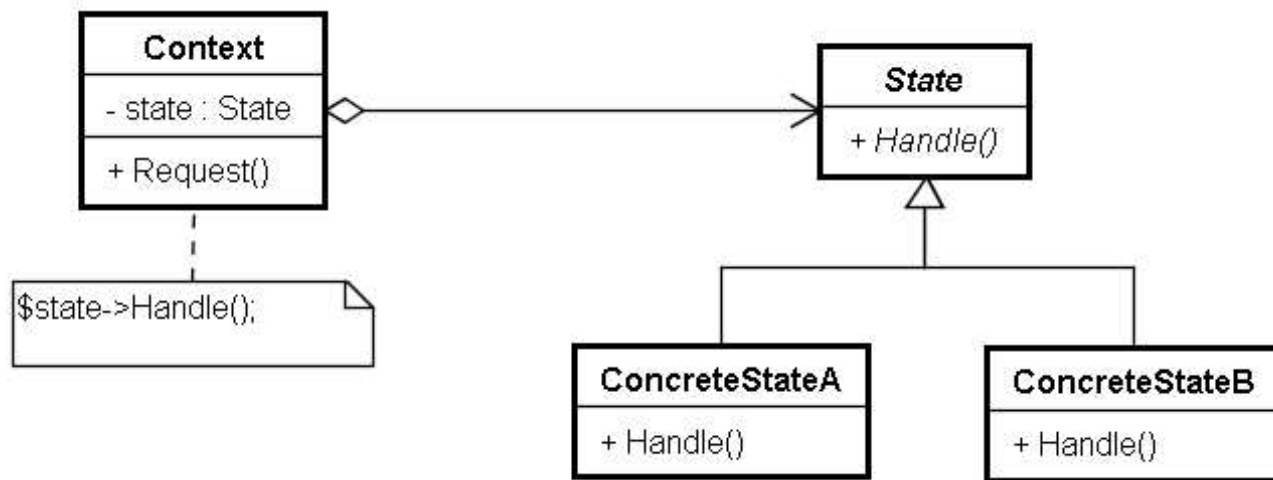
Stateパターンは、オブジェクトの振る舞いに注目したパターンです。

Stateパターンでは、それぞれの具体的な状態をクラスとして定義します。これらの状態クラスは「どの状態か」を意識することなくアクセスできるよう共通のAPIを持っています。また、クライアントからアクセスさせるための処理クラスを用意し、その内部に状態クラスのインスタンスを保持します。この処理クラスは、クライアントからの要求を受け取った後、内部に保持した状態オブジェクトに実際の処理を任せています。

こうすることで、状態による振る舞いの変化を実現しています。

### Stateパターンの構造

Stateパターンのクラス図と構成要素は、次のとおりです。



## Stateクラス

それぞれの状態に共通のAPIを定義します。このAPIは、状態固有の振る舞いをおこなうメソッドになります。Contextクラスからは、Stateクラスで定義されたAPIを通じて、ConcreteStateクラスで実装された具体的な処理を呼び出します。

## ConcreteStateクラス

Stateクラスのサブクラスで、State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、**状態ごとの具体的な処理内容を記述**します。

## Contextクラス

State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。これにより、ConcreteStateクラスに依存することがなくなり、ConcreteStateクラスを簡単に切り替えることができます。

## Stateパターンのメリット

State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。

状態に固有の処理をまとめることができる

Stateパターンを適用すると、それぞれの状態に固有な処理がクラスにまとめて実装されますので、コードを記述する際、その状態に固有な処理に専念できます。これにより、保守性が高まります。

また、新しい状態が追加された場合も、新しい状態クラスを作成するだけで済みます。

状態に固有の処理を選択するための条件文がなくなる

状態によって振る舞いを変える場合、1つのクラスやメソッドに処理を記述したくなってしまう。しかし、クラスやメソッドごとにif文やswitch文を使って処理を分岐することになってしまい、コードの可読性を落としてしまいます。また、保守性・拡張性が下がることにもつながります。Stateパターンを適用すると、状態ごとの処理がクラス単位にまとめて実装されますので、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。

## Stateパターンの適用例

Stateパターンの適用例を見てみましょう。

ここでは簡単な認証機能にStateパターンを適用した例に取り上げます。

このアプリケーションには、認証機能のほか、簡単なカウンタ機能があります。このカウンタ機能はログインした状態の場合のみ利用できます。また、状態としては認証済み（ログイン）、未認証（ログアウト）の2つがあります。

では、早速コードを見ていきましょう。

まずは、Contextクラスに相当するUserクラスです。クライアントはこのUserクラスを利用しますが、その動作は内部に保持している状態オブジェクトによって変化します。

●Userクラス (User.class.php)

```
<?php
require_once 'UnauthorizedState.class.php';
?>
```

```

<?php
/**
 * Contextクラスに相当する
 */
class User {

    private $name;
    private $state;
    private $count = 0;

    public function __construct($name) {
        $this->name = $name;

        // 初期値
        $this->state = UnauthorizedState::getInstance();
        $this->resetCount();
    }

    /**
     * 状態を切り替える
     */
    public function switchState() {
        echo "状態遷移:" . get_class($this->state) . "→";
        $this->state = $this->state->nextState();
        echo get_class($this->state) . "<br>";
        $this->resetCount();
    }

    public function isAuthenticated() {
        return $this->state->isAuthenticated();
    }

    public function getMenu() {
        return $this->state->getMenu();
    }

    public function getUsername() {
        return $this->name;
    }

    public function getCount() {
        return $this->count;
    }

    public function incrementCount() {
        $this->count++;
    }

    public function resetCount() {
        $this->count = 0;
    }
}
?>

```

また、状態を切り替えるためのメソッドswitchStateが定義されています。このメソッドの中に状態を切り替えるためのコードが記述されていますが、

```
$this->state = $this->state->nextState();
```

といった具合に内部に保持している状態オブジェクトのnextStateメソッドを呼び出しているだけです。状態オブジェクトには2種類あると説明しましたが、どの状態かを意識することなくnextStateメソッドを呼び出しています。この詳細の説明は、状態クラスの説明のときにおこないますので、ちょっと待っていてください。

次に状態を表すクラスたちを見ていきましょう。

まずは状態クラスに共通のAPIを定義しているUserStateインターフェースです。Stateクラスに相当します。

それぞれの具体的な状態クラスは、このインターフェースを実装することになります。

- UserStateインターフェース (UserState.class.php)

```

<?php
/**
 * Stateクラスに相当する
 * 状態毎の動作・振る舞いを定義する
 */
interface UserState {
    public function isAuthenticated();
    public function nextState();
    public function getMenu();
}
?>

```

このインターフェースには3つのメソッドが定義されています。認証されているかどうかを返すisAuthenticatedメソッド、状態を切り替えるnextStateメソッド、その状態でどのようなメニュー一覧を返すgetMenuメソッドです。

次は具体的な状態を表すクラスです。ConcreteStateクラスに相当します。

認証済みの状態を表すAuthorizedStateクラスと未認証の状態を表すUnauthorizedStateクラスの2クラスです。

●AuthorizedStateクラス(AuthorizedState.class.php)

```

<?php
require_once 'UserState.class.php';
require_once 'UnauthorizedState.class.php';
?>
<?php
/**
 * ConcreteStateクラスに相当する
 * 認証後の状態を表すクラス
 */
class AuthorizedState implements UserState {

    private static $singleton = null;

    private function __construct() {

    }

    public static function getInstance() {
        if (self::$singleton == null) {
            self::$singleton = new AuthorizedState();
        }
        return self::$singleton;
    }

    public function isAuthenticated() {
        return true;
    }

    public function nextState() {
        // 次の状態（未認証）を返す
        return UnauthorizedState::getInstance();
    }

    public function getMenu() {
        $menu = '<a href="?mode=inc">カウントアップ</a> | '
            . '<a href="?mode=reset">リセット</a> | '
            . '<a href="?mode=state">ログアウト</a>';
        return $menu;
    }
}
?>

```

●UnauthorizedStateクラス(UnauthorizedState.class.php)

```

<?php
require_once 'UserState.class.php';
require_once 'AuthorizedState.class.php';
?>
<?php
/**
 * ConcreteStateクラスに相当する
 * 未認証の状態を表すクラス
 */
class UnauthorizedState implements UserState
{
    private static $singleton = null;

    private function __construct() {

    }

    public static function getInstance() {
        if (self::$singleton === null) {
            self::$singleton = new UnauthorizedState();
        }
        return self::$singleton;
    }

    public function isAuthenticated() {
        return false;
    }

    public function nextState() {
        // 次の状態（認証）を返す
        return AuthorizedState::getInstance();
    }

    public function getMenu() {
        $menu = '<a href="?mode=state">ログイン</a>';
        return $menu;
    }
}
?>

```

これらのクラスには、Singletonパターンも併せて適用されています。

注目していただきたいメソッドはnextStateメソッドです。このメソッドは次の状態に切り替えるためのメソッドですが、UnauthorizedStateクラスではAuthorizedStateクラスのインスタンス、AuthorizedStateクラスではUnauthorizedStateクラスのインスタンスを返すようになっています。つまり、「未認証」の次の状態は「認証済み」、「認証済み」の次の状態は「未認証」という状態の遷移を表しています。

Userクラスの内部では、このnextStateメソッドを使って状態の切り替えをおこなっていましたが、その際「**現在どの状態なのか**」を意識しないでnextStateメソッドを呼び出していたと思います。これは、状態クラスであるAuthorizedStateクラスやUnauthorizedStateクラス自身が次に遷移すべき状態を知っているからこそ可能になっています。

なお、新しい状態を増やす場合、AuthorizedStateクラスやUnauthorizedStateクラスと同様にUserStateインターフェースを実装したクラスを用意するだけで済みます。

最後にクライアントのコードです。このコードには、**状態に関連するコードが一切出てきていない**ことを確認してください。

#### ●クライアント側コード(state\_client.php)

```

<?php
require_once 'User.class.php';
?>
<?php
session_start();

$context = isset($_SESSION['context']) ? $_SESSION['context'] : null;
if (is_null($context)) {
    $context = new User('ほげ');
}

```

```

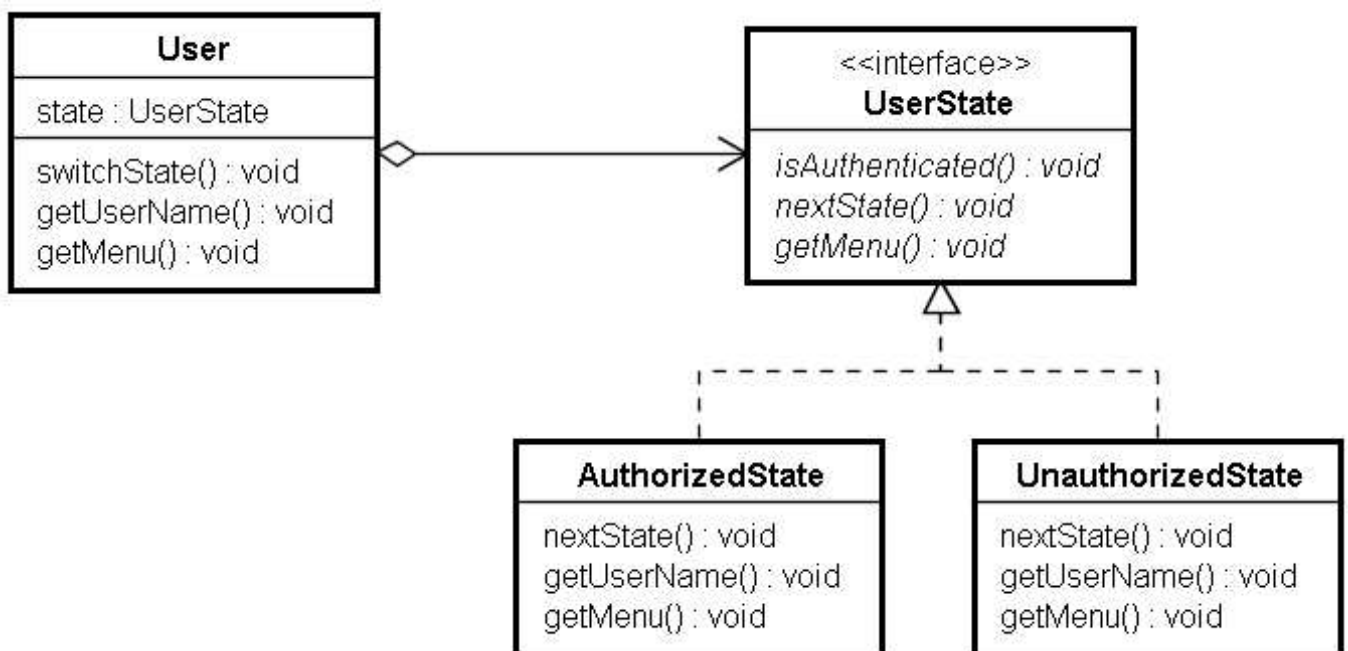
$mode = (isset($_GET['mode']) ? $_GET['mode'] : '');
switch ($mode) {
case 'state':
    echo '<p style="color: #aa0000">状態を遷移します</p>';
    $context->switchState();
    break;
case 'inc':
    echo '<p style="color: #008800">カウントアップします</p>';
    $context->incrementCount();
    break;
case 'reset':
    echo '<p style="color: #008800">カウントをリセットします</p>';
    $context->resetCount();
    break;
}

$_SESSION['context'] = $context;

echo 'ようこそ、' . $context->getUserName() . 'さん<br>';
echo '現在、ログインして' . ($context->isAuthenticated() ? 'います' : 'いません') . '<br>';
echo '現在のカウント:' . $context->getCount() . '<br>';
echo $context->getMenu() . '<br>';
?>

```

このサンプルアプリケーションのクラス図は、次のようになります。



## Stateパターンのオブジェクト指向的要素

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

Contextクラスは、State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Stateクラスを継承もしくは実装したConcreteStateクラスのインスタンスです。一方、Contextクラスは、クライアントから受け取った処理要求を、保持しているStateインスタンスに処理を委譲します。**Contextクラス自身は状態に関する処理を一切行いません**。この結果、保持したインスタンスが具体的にどのConcreteStateクラスのインスタンスなのかを意識することなく、振る舞いを変更することができます。併せて、ConcreteStateクラスを簡単に差し替えたり、追加したりできるのです。Stateパターンは、委譲を使って状態固有の処理を切り替えるパターンと言えます。

また、Stateパターンと**Strategyパターン**は非常によく似たパターンですが、用いる場面は全く異なります。Stateパターンは**状態**によって振る舞いを変えますが、Strategyパターンは**処理する対象の違い**によって振る舞いを変えます。

## 関連するパターン

### Singletonパターン

Singletonパターンは、ConcreteStateクラスに適用できることが多いパターンです。

## Flyweightパターン

Singletonパターンと同様、ConcreteStateクラスにFlyweightパターンを適用できる場合があります。

## まとめ

---

ここでは「状態」と「状態による振る舞い」を1つのクラスにまとめるStateパターンについて説明しました。