Observer ～状態変化を通知する

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Observerパターンについて説明します。

observerという単語は「observe」する人、つまり「観察者」「観測者」の意味です。ということは、「観察」する対象はやはりオブジェクトということになりそうですね。

実際そのとおりで、観察対象のオブジェクトが変化したときに通知してもらい、その変化を他のオブジェクトにも伝えるためのパターンです。

では、Observerパターンを見ていきます。

たとえば

たとえば、複数のオブジェクトが連携して動作する場面を考えてみましょう。

こういった場面では、あるオブジェクトの状態に変化があった場合、矛盾を起こさないよう他のオブジェクトもその変化にあわせて振る舞いを変える必要が出てきます。

では、この連係動作をどうやって管理すれば良いでしょうか？

最も簡単な方法は、「このオブジェクトが変化したらこうする」といったコードを記述することでしょう。しかし、こういったある特定のクラスに依存したコードを記述するとクラスどうしの関係が緊密になってしまい、クラスの再利用性を下げることになります。

また、連携し合うオブジェクトの数が少なければオブジェクトどうしの関連はそれほど複雑にはなりませんので、あまり大きな問題になることは少ないでしょう。しかし、連携するオブジェクトの数が多い場合はどうでしょうか？最後には誰も管理できないぐらい複雑になってしまいますね？

ここで、**Observerパターン**の登場です。Observerパターンはクラスどうしの結合をゆるく保ったまま、協調動作を実現するパターンです。

Observerパターンとは？

Observer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。

> あるオブジェクトが状態を変えたときに、それに依存するすべてのオブジェクトに
> 自動的にそのことが知らされ、また、それらが更新されるように、オブジェクト間に
> 一対多の依存関係を定義する。

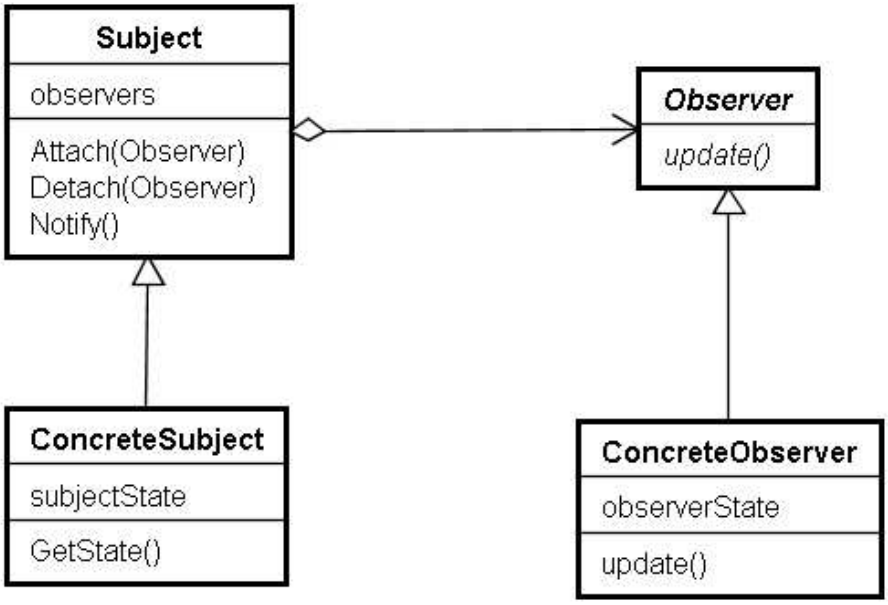
Observerパターンはオブジェクトの振る舞いに注目したパターンで、観測対象のオブジェクトに変化があったとき、それを観測しているすべてのオブジェクトに通知を行うためのパターンです。つまり、状態変化に応じた処理を行う場合に有効なパターンと言えます。

Observerパターンでは、観測対象のクラスとそれを観測するクラスをそれぞれ用意します。観測対象クラスは、内部に観測クラスのインスタンスを複数保持できる構造になっています。一方の観測クラスは、通知を受け取るための共通の**API**を持っています。そして、観測対象クラスの状態が変化したとき、保持した観測インスタンスの通知用メソッドを呼び出し、「状態が変化した」との通知をおこないます。

このとおり、観測対象クラスと観測クラスの間に強い結びつきはありませんが、協調動作するための仕組みはちゃんと用意されています。お気づきになりましたか？

Observerパターンの構成要素

Observerパターンのクラス図と構成要素は、次のようになります。



Subjectクラス

観測対象のクラスです。内部には観測クラスであるObserver型のオブジェクトを保持しており、Observer型のオブジェクトを登録、削除するAPI、またObserver型のオブジェクトに通知をおこなうAPIが提供されています。通知をおこなう場合、Subjectクラスからは Observerクラスで定義されたAPIを通じて、ConcreteObserverクラスで実装されている具体的な処理を呼び出します。

Observerクラス

観測クラスです。通知を受け取るためのAPIを定義します。具体的な処理内容は、サブクラスのConcreteObserverクラスで実装します。Listener、Handlerと呼ばれる場合もあります。

ConcreteSubjectクラス

Subjectクラスのサブクラスで、Observerクラスに影響する状態を保持しています。また、その状態を取得するためのAPIを提供しています。この保持した状態が変化したとき、Observer型のオブジェクトに通知を送ります。

ConcreteObserverクラス

Observerクラスのサブクラスで、Observerクラスで定義されたAPIを実装したクラスです。このクラスに通知を受け取った場合の具体的な処理内容を記述します。

なお、**ConcreteObserver**クラスどうしは、通知を受ける順番が変わっても正しく動作するよう設計される必要があります。

Observerパターンのメリット

Observer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。

ConcreteSubjectクラスとConcreteObserverクラスを独立して拡張できる

Observerパターンは、オブジェクトどうしをゆるく結合したまま協調動作をさせることを可能にします。つまり、ConcreteSubjectクラスとConcreteObserverクラスの間に直接の関係はありません。両クラスの関係を作っているのは、それぞれの親クラスであるSubjectクラスとObserverクラスです。このため、それぞれ独立して修正・拡張することができます。また、新しいConcreteObserverクラスを追加する場合もSubjectクラスやConcreteSubjectクラスを修正する必要はありません。

Observerパターンの適用例

ここではECサイトでよく見かけるショッピングカートを取り上げ、ショッピングカートの状態が変化するたびに様々なオブジェクトと連係動作をおこなう様子を見てみましょう。

このサンプルでは、観察対象としてショッピングカート、観察者として次の2つを用意しました。

ショッピングカートにある特定の商品が追加された場合、プレゼント商品を追加する観察者

ショッピングカートの内容を出力する観察者

これら観察者の具体的なコードは後ほど出てきますので、まずは観察対象であるショッピングカートを表すCartクラスから見てみましょう。

●Cartクラス（Cart.class.php）

```
<?php
/**
 * Subjectクラス＋ConcreteSubjectクラスに相当する
```

```
 */
class Cart {

    private $items;
    private $listeners;

    public function __construct() {
        $this->items = array();
        $this->listeners = array();
    }

    public function addItem($item_cd) {
        $this->items[$item_cd] = (isset($this->items[$item_cd]) ? ++$this->items[$item_cd] : 1);
        $this->notify();
    }

    public function removeItem($item_cd) {
        $this->items[$item_cd] = (isset($this->items[$item_cd]) ? --$this->items[$item_cd] : 0);
        if ($this->items[$item_cd] <= 0) {
            unset($this->items[$item_cd]);
        }
        $this->notify();
    }

    public function getItems() {
        return $this->items;
    }

    public function hasItem($item_cd) {
        return array_key_exists($item_cd, $this->items);
    }

    /**
     * Observerを登録するメソッド
     */
    public function addListener(CartListener $listener) {
        $this->listeners[get_class($listener)] = $listener;
    }

    /**
     * Observerを削除するメソッド
     */
    public function removeListener(CartListner $listener) {
        unset($this->listeners[get_class($listener)]);
    }

    /**
     * Observerへ通知するメソッド
     */
    public function notify() {
        foreach ($this->listeners as $listener) {
            $listener->update($this);
        }
    }
}
?>
```

Cartクラスには、まずショッピングカートとして必要となるメソッドが用意されています。商品を追加するメソッド(addItem)、削除するメソッド(removeItem)、全ての商品を取り出すメソッド(getItems)、商品がすでにショッピングカートに入れられているかどうかを返すメソッド(hasItem)です。

getItemsメソッドとhasItemメソッドについては、観察者のところでも出てきますので、もうちょっと待っていてください。

このほかにaddListenerメソッドとremoveListenerメソッド、notifyメソッドがありますね。

addListenerメソッドとremoveListenerメソッドの2つは、ショッピングカートの状態変化を観察する観察者を登録、削除するメソッドです。

notifyメソッドは登録された観察者に「状態が変わりましたよ」と通知するためのメソッドです。通知をおこなう際、変化した具体的な内容を観察者に渡す必要がありますが、今回はCartオブジェクト自身を引数として渡しています。

また、登録された全ての観察者に対して通知をおこなっている事が分かると思います。

この通知を受け取った観察者は、その変化に対応する動作をおこなうことになります。

次に、観察者側のクラスたちを見ていきます。

まずは、観察者に共通するAPIを定義するCartListenerクラスです。ここではinterfaceとして定義しており、唯一のメソッドであるupdateメソッドを定義しています。引数としては、Cartオブジェクトを受け取ります。この渡されたCartオブジェクトから、変更内容を直接取り出すことができます。この具体的に取り出す様子は、次のPresentListenerクラスとLoggingListenerクラスで見てみましょう。

●CartListenerクラス（CartListener.class.php）

```
<?php
/**
 * Observerクラスに相当する
 */
interface CartListener {
    public function update(Cart $cart);
}
?>
```

PresentListenerクラスとLoggingListenerクラスはCartListenerクラスを実装したクラスで、具体的な観察者となります。

PresentListenerクラスは、特定の商品が追加された場合にプレゼント商品を追加します。具体的には、クッキーセットがショッピングカートに含まれる場合、プレゼント商品として「プレゼント」をショッピングカートに追加します。

●PresentListenerクラス（PresentListener.class.php）

```
<?php
require_once 'CartListener.class.php';
?>
<?php
/**
 * ConcreteObserverクラスに相当する
 */
class PresentListener implements CartListener {

    private static $PRESENT_TARGET_ITEM = '30:クッキーセット';
    private static $PRESENT_ITEM = '99:プレゼント';

    public function __construct() {
    }

    public function update(Cart $cart) {
        if ($cart->hasItem(self::$PRESENT_TARGET_ITEM) &&
            !$cart->hasItem(self::$PRESENT_ITEM)) {
            $cart->addItem(self::$PRESENT_ITEM);
        }
        if (!$cart->hasItem(self::$PRESENT_TARGET_ITEM) &&
            $cart->hasItem(self::$PRESENT_ITEM)) {
            $cart->removeItem(self::$PRESENT_ITEM);
        }
    }
}
?>
```

LoggingListenerクラスは、ショッピングカートの状態が変化するたびにvar_dump関数を使ってショッピングカートに含まれている商品と個数を出力します。

●LoggingListenerクラス（LoggingListener.class.php）

```
<?php
require_once 'CartListener.class.php';
?>
<?php
/**
 * ConcreteObserverクラスに相当する
 */
class LoggingListener implements CartListener {

    public function __construct() {
    }

    public function update(Cart $cart) {
        var_dump($cart->getItems());
    }
}
?>
```

PresentListenerクラスとLoggingListenerクラス共に、updateメソッドに渡されたCartオブジェクトからショッピングカートの状態を取得し、処理をおこなっていることが分かったと思います。PresentListenerクラスではhasItemメソッド、LoggingListenerクラスではgetItemsメソッドを使っていますね。こうやって、観察者はうまい具合に観測対象の情報を取得することができています。

なお、観察対象のクラスは観察者に「どこまで情報を公開するか」を検討する必要があります。これはObserverパターンを適用する上で重要なポイントになってきます。

最後に利用側のコードです。

ショッピングカートの生成をcreateCart関数にまとめてあります。

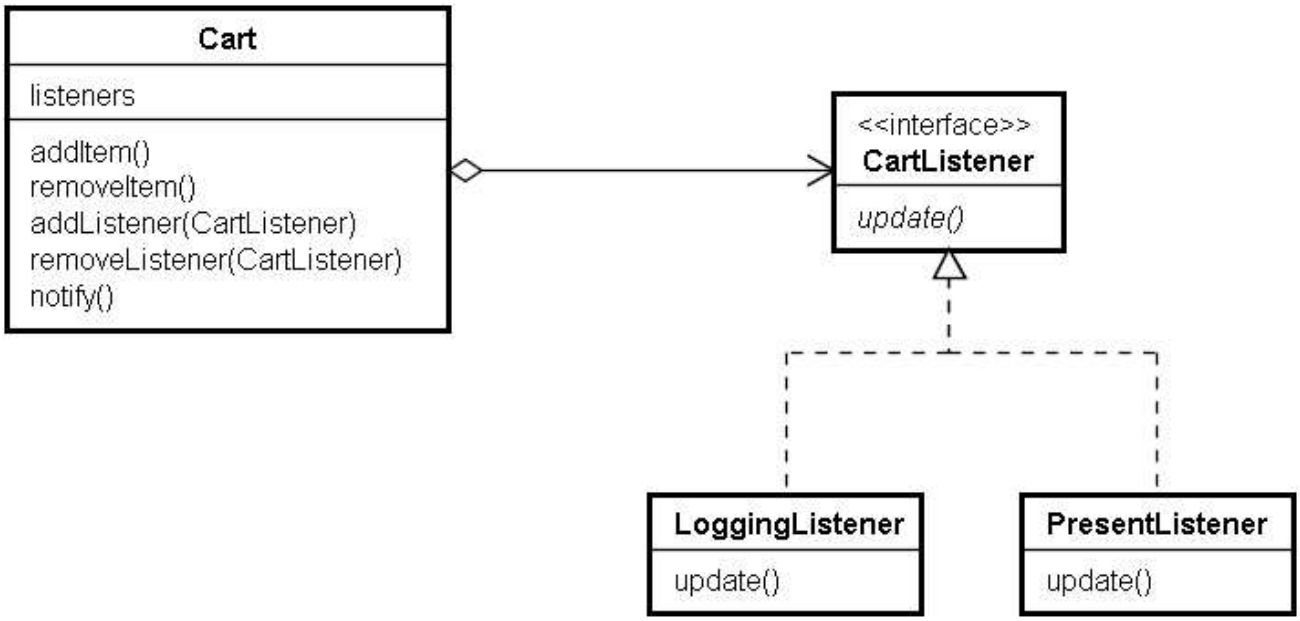
また、Observerパターンの特徴の1つとして観察者を動的に追加、削除することが挙げられますが、createCart関数の中でCartオブジェクトを生成したあとに観察者であるPresentListenerクラスとLoggingListenerクラスの各インスタンスを追加していることが分かると思います。

たとえば、「開発中はロギングするが、リリースするときはロギングしない」といった「ある条件の場合だけ別の観察者を追加・削除する」といったことも可能になります。

●クライアント側コード(observer_client.php)

```
<?php
require_once 'Cart.class.php';
require_once 'PresentListener.class.php';
require_once 'LoggingListener.class.php';
?>
<?php
function createCart() {
    $cart = new Cart();
    $cart->addListener(new PresentListener());
    $cart->addListener(new LoggingListener());

    return $cart;
}
?>
<?php
    session_start();

    $cart = isset($_SESSION['cart']) ? $_SESSION['cart'] : null;
    if (is_null($cart)) {
        $cart = createCart();
    }

    $item = (isset($_POST['item']) ? $_POST['item'] : '');
    $mode = (isset($_POST['mode']) ? $_POST['mode'] : '');
    switch ($mode) {
    case 'add':
        echo '<p style="color: #aa0000">追加しました</p>';
        $cart->addItem($item);
        break;
    case 'remove':
        echo '<p style="color: #008800">削除しました</p>';
        $cart->removeItem($item);
        break;
    case 'clear':
        echo '<p style="color: #008800">クリアしました</p>';
        $cart = createCart();
        break;
    }

    $_SESSION['cart'] = $cart;

    echo '<h1>商品一覧</h1>';
    echo '<ul>';
    foreach ($cart->getItems() as $item_name => $quantity) {
        echo '<li>' . $item_name . ' ' . $quantity . '個</li>';
    }
?>
<form action="" method="post">
<select name="item">
<option value="10:Tシャツ">Tシャツ</option>
<option value="20:ぬいぐるみ">ぬいぐるみ</option>
<option value="30:クッキーセット">クッキーセット</option>
</select>
<input type="submit" name="mode" value="add">
<input type="submit" name="mode" value="remove">
<input type="submit" name="mode" value="clear">
</form>
```

まとめとして、このサンプルコードのクラス図を示しておきます。



Observerパターンのオブジェクト指向的要素

Observer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。

観測対象クラスの親クラスであるSubjectクラスでは、Observer型のインスタンスを内部に保持しています。このインスタンスは、Observerクラスを継承したConcreteObserverクラスのインスタンスです。Observerクラスは、内部の状態が変更したとき、保持したObserverインスタンスに「状態が変化しましたよ」という通知を送ります。一方のObserverクラスは、観測対象クラスから送られた通知を元におこなう処理だけを定義・実装しています。

Subjectクラスは**Observer**クラスに通知するだけ、**Observer**クラスは受けた通知を処理するだけ、という関係しかありません。「いつ通知を出すか」「通知を元にどういった処理をするか」は、それぞれのサブクラスに任されています。つまり、ConcreteSubjectクラスとConcreteObserverクラスは直接お互いを知りませんが、親クラスどうしで作られた関係を利用してお互いにやりとりを行います。

この結果、ConcreteObserverクラスを簡単に差し替えたり、追加したりできるのです。

関連するパターン

Mediatorパターン

Observerパターンと同様、状態変化を通知するパターンです。Observerパターンでは、Observerクラス自体が通知処理をおこないますが、Mediatorパターンは通知の仲介をおこなうだけです。また、Observerパターンでは、単一オブジェクトの状態変化を複数のオブジェクトへ通知しますが、Mediatorパターンでは複数のオブジェクトの状態変化があるオブジェクトに集約されて、そこから他のオブジェクトに通知されます。

まとめ

ここではオブジェクトどうしの協調動作を実現するObserverパターンを見てきました。