Facade ～シンプルな唯一の窓口

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

構造＋クラス、オブジェクト

はじめに

ここではFacadeパターンについて説明します。

「facade」は「ファサード」と読みます。あまり聞き慣れない言葉ですが、フランス語で「正面窓口」という意味です。

Facadeパターンは、複雑な関連を持つクラス群を簡単に利用するための「窓口」を用意するパターンです。

たとえば

世の中には個人的にしか利用しない簡単なシステムから世界規模で大々的に利用されるシステムまで、大小さまざまなシステムがあります。

最初はごく簡単な機能だけしかなかった小さなシステムも、度重なる機能拡張や修正で徐々にそのシステムは大きくなっていきます。システムの規模が大きくなればなるほどクラスの数は増えていき、それに従ってクラスどうしの関係も複雑になっていきます。

クラスどうしの関係が複雑になってくると、それを利用する側に急激な負担が発生します。それはクラスの使い方です。ありがちな話として、「このクラスはあのクラスと一緒に使う必要がある」とか「このクラスを使う場合は、前もってこちらのクラスのメソッドを呼び出し、あらかじめ処理をしておかなければいけない」といったものです。つまり、あるクラスを利用するために複数の他のクラスを知っている必要がある、ということです。

こういった状況では、利用する側にミスが発生しやすくなり、思わぬ不具合やシステム障害の温床になってしまいます。また、システムのセキュリティ面では、使い方が複雑になればなるほどセキュリティの確実な確保がしにくい状況になります。

もし、クラスどうしの複雑な関係を意識しなくても利用できるシンプルな「窓口」があればどうでしょうか？利用する側は非常に便利になり、ミスが少なくなりそうですね。またセキュリティの確保も、バラバラなクラスどうしのままよりは格段に改善されそうです。

このような場面で**Facadeパターン**が活躍します。

Facadeパターンとは？

Facade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。

> サブシステム内に存在する複数のインターフェースに1つの統一インターフェースを与える。
> Facadeパターンはサブシステムの利用を容易にするための高レベルインターフェースを定義する。

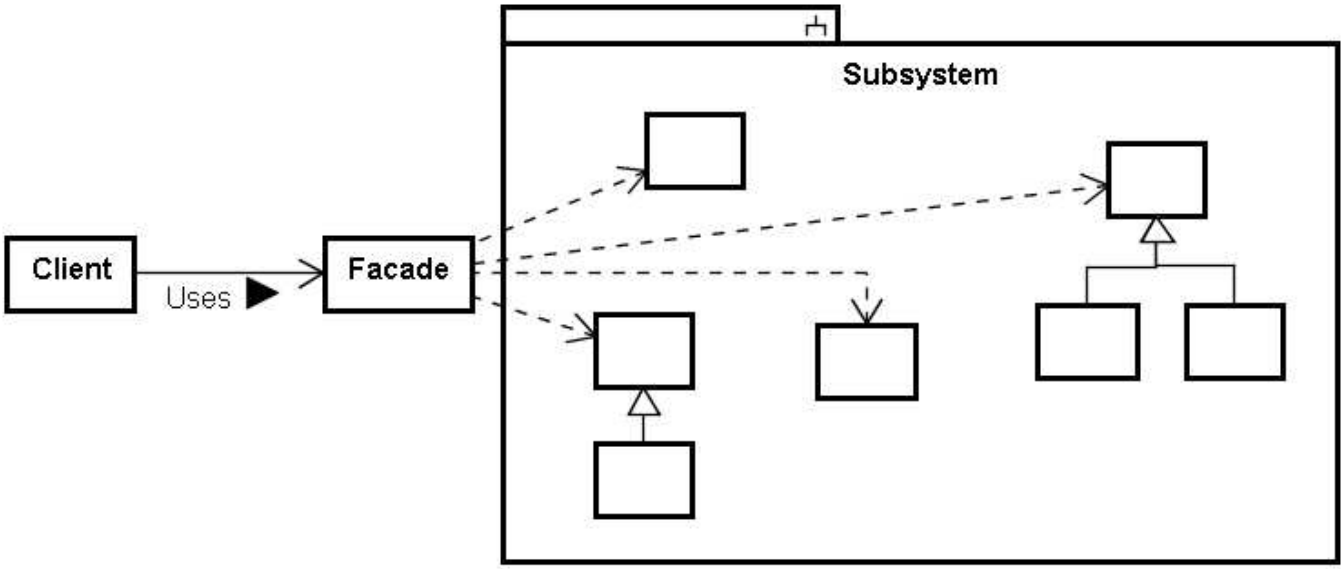
Facadeパターンは、複雑に関連しあうクラス群を隠蔽するようなクラスを用意し、そのクラスに統一されたAPIを実装します。利用側はそのAPIを通じてクラス群を利用します。

こう見てみると、みなさんもすでにどこかで使ったことがあるかも知れませんね。

一般的に、システムがある規模より大きくなると、より小さなサブシステム単位に分割されます。このサブシステムをシンプルに利用できるようにするためにFacadeパターンが利用されます。Facadeパターンを適用すると、あるAPIだけを呼び出すだけでサブシステムを利用できるようになります。また、サブシステムどうしの依存関係もシンプルになります。

Facadeパターンの構造

Facadeパターンのクラス図と構成要素は、次のようになります。



Facadeクラス

サブシステムで提供される統一APIを持つクラスです。サブシステム内のクラス同士の関係を知っています。また、クライアントからの要求を、サブシステム内の適切なオブジェクトに委譲します。

サブシステム内のクラス群

サブシステムを構成するクラス群です。Facadeクラスの存在は知りません。

Clientクラス

サブシステムを利用するクラスです。Facadeクラスを通じてサブシステムにアクセスします。

Facadeパターンのメリット

Facade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。

サブシステムの構成要素を隠蔽する

Facadeパターンを適用すると、クライアントからはサブシステムの入り口しか見えなくなります。結果、クライアントが意識しなければならないクラスの数を抑えることができ、そのサブシステムを簡単に利用できます。

サブシステムとクライアントの結びつきを緩くする

Facadeクラスを通じてサブシステムにアクセスすることで、サブシステムとクライアントの結びつきが緩くなります。つまり、クライアントとサブシステム内部との独立性がより高くなると言えます。独立性が高くなることで、クライアントのコードに影響を与えることなく、サブシステム内部を変更できます。

Facadeパターンの適用例

Facadeパターンの適用例を見てみましょう。

ここでは、注文処理をおこなうアプリケーションを取り上げます。この注文処理はサブシステムとして扱うことができるため、Facadeパターンを適用して注文処理を利用するためのシンプルなAPIを提供すると共に、クライアントとの結びつきを緩くしています。

まずは、商品クラスと注文商品クラス、注文クラスの3つから見ていきましょう。この3クラスはとてもシンプルなクラスです。

商品を表すItemクラスは、商品ID、商品名、単価の各情報を保持するだけのシンプルなものです。コンストラクタで全ての情報を引数として受け取り、それぞれの情報にアクセスするためのメソッドを用意しています。

●Itemクラス（Item.class.php）

```
<?php
class Item {
    private $id;
    private $name;
    private $price;
    public function __construct($id, $name, $price) {
        $this->id = $id;
        $this->name = $name;
        $this->price = $price;
    }
    public function getId() {
        return $this->id;
```

```
    }
    public function getName() {
        return $this->name;
    }
    public function getPrice() {
        return $this->price;
    }
}
?>
```

注文する商品を表すOrderItemクラスは、内部に商品クラスのインスタンスとその注文数量を保持します。Itemクラスと同様、コンストラクタでItemオブジェクトと数量を受け取り、それらにアクセスするためのメソッドもあります。

●OrderItemクラス（OrderItem.class.php）

```
<?php
require_once 'Item.class.php';
?>
<?php
class OrderItem {
    private $item;
    private $amount;
    public function __construct(Item $item, $amount) {
        $this->item = $item;
        $this->amount = $amount;
    }
    public function getItem() {
        return $this->item;
    }
    public function getAmount() {
        return $this->amount;
    }
}
?>
```

OrderクラスはOrderItemオブジェクトを格納するクラスです。OrderItemを追加するためのaddItemがあります。

●Orderクラス（Order.class.php）

```
<?php
require_once 'OrderItem.class.php';
?>
<?php
class Order {
    private $items;
    public function __construct() {
        $this->items = array();
    }
    public function addItem(OrderItem $order_item) {
        $this->items[$order_item->getItem()->getId()] = $order_item;
    }
    public function getItems() {
        return $this->items;
    }
}
?>
```

では、ここからは商品の注文処理をおこなうクラス群を見ていきましょう。

さて、商品の注文処理ではどの様な処理をおこなう必要があるでしょうか？

商品在庫の確認や引当処理、決済や確認メールの送信など様々な処理が考えられますが、ここでは簡易的に商品在庫の引当と注文情報の処理の2つをおこないます。この2つの処理は、実際にはデータベースなどにアクセスしてそれぞれの情報を更新したり追加したりしますが、このアプリケーションではメッセージを表示するだけとなっています。

これらの商品在庫の引当と注文情報の処理をおこなうクラスが、ItemDaoクラスとOrderDaoクラスです。これらのクラスには、データベースだけに関する処理をまとめるDAO（Data Access Object）パターンが適用されています。

ItemDaoクラスは、インスタンス化と同時に商品情報を初期化しています。また、Singletonパターンも適用されており、ItemDaoインスタンスを取得するためのgetInstanceメソッドが用意されています。また、商品IDからItemオブジェクトを取得するfindByIdメソッド、在庫の引当処理をおこなうsetAsideメソッドがあります。

●ItemDaoクラス（ItemDao.class.php）

```
<?php
require_once 'OrderItem.class.php';
?>
<?php
class ItemDao {
    private static $instance;
    private $items;
    private function __construct() {
        $fp = fopen('item_data.txt', 'r');

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

        $this->items = array();
        while ($buffer = fgets($fp, 4096)) {
            $item_id = trim(substr($buffer, 0, 10));
            $item_name = trim(substr($buffer, 10, 20));
            $item_price = trim(substr($buffer, 30));

            $item = new Item($item_id, $item_name, $item_price);
            $this->items[$item->getId()] = $item;
        }

        fclose($fp);
    }

    public static function getInstance() {
        if (!isset(self::$instance)) {
            self::$instance = new ItemDao();
        }
        return self::$instance;
    }

    public function findById($item_id) {
        if (array_key_exists($item_id, $this->items)) {
            return $this->items[$item_id];
        } else {
            return null;
        }
    }

    public function setAside(OrderItem $order_item) {
        echo $order_item->getItem()->getName() . 'の在庫引当をおこないました<br>';
    }
}
?>
```

setAsideメソッドでは、引当をおこなう商品名とメッセージが表示されるだけですが、本来はデータベースにアクセスして在庫情報を更新するコードが記述されます。

なお、商品情報は固定長データとしてファイルに登録されています。

●商品情報（item_data.txt）

```
商品ID    商品名              価格
1         限定Ｔシャツ        1500
2         ぬいぐるみ          2000
3         クッキーセット      800
```

データフォーマットは詳細は表を参照してください。

項目	開始位置	終了位置
商品ID	1	10
商品名	11	30
価格	31	-

次はOrderDaoクラスです。このクラスにはcreateOrderメソッドのみが宣言されています。このメソッドは、Orderオブジェクトを受け取り、注文情報を表示しています。本来はデータベースにアクセスし、注文情報を登録する処理が記述されることになります。

●OrderDaoクラス（OrderDao.class.php）

```php
<?php
require_once 'Order.class.php';
?>
<?php
class OrderDao {
    public static function createOrder(Order $order) {
        echo '以下の内容で注文データを作成しました';

        echo '<table border="1">';
        echo '<tr>';
        echo '<th>商品番号</th>';
        echo '<th>商品名</th>';
        echo '<th>単価</th>';
        echo '<th>数量</th>';
        echo '<th>金額</th>';
        echo '</tr>';

        foreach ($order->getItems() as $order_item) {
            echo '<tr>';
            echo '<td>' . $order_item->getItem()->getId() . '</td>';
            echo '<td>' . $order_item->getItem()->getName() . '</td>';
            echo '<td>' . $order_item->getItem()->getPrice() . '</td>';
            echo '<td>' . $order_item->getAmount() . '</td>';
            echo '<td>' . ($order_item->getItem()->getPrice() * $order_item->getAmount()) . '</td>';
            echo '</tr>';
        }
        echo '</table>';
    }
}
?>
```

お待たせしました。それでは、Facadeクラスに相当するOrderManagerクラスを見てみましょう。

OrderManagerクラスはクライアント側から注文処理をおこなうためのAPIを定義し、具体的な注文処理を隠蔽する役割を担います。orderメソッドがそれにあたります。

●OrderManagerクラス（OrderManager.class.php）

```php
<?php
require_once 'Order.class.php';
require_once 'ItemDao.class.php';
require_once 'OrderDao.class.php';
?>
<?php
class OrderManager {
    public static function order(Order $order) {
        $item_dao = ItemDao::getInstance();
        foreach ($order->getItems() as $order_item) {
            $item_dao->setAside($order_item);
        }

        OrderDao::createOrder($order);
    }
}
?>
```

Facadeパターンを適用しない場合、これらの処理を呼び出すコードをクライアント側に記述する必要があります。これでは、利用側は「注文処理」でおこなう処理や順序を全て知っている必要があり、クライアントとそれぞれの処理を担当するクラスの結びつきが強くなってしまいます。

Facadeパターンを適用して「注文する」というAPI（ここではorderメソッド）を用意することで、クライアント側はそのAPIを使うだけのシンプルなコードなり、またFacadeクラスだけに依存するコードになります。

では、Facadeパターンを適用した場合のクライアント側のコードはどの様になるか、見てみましょう。

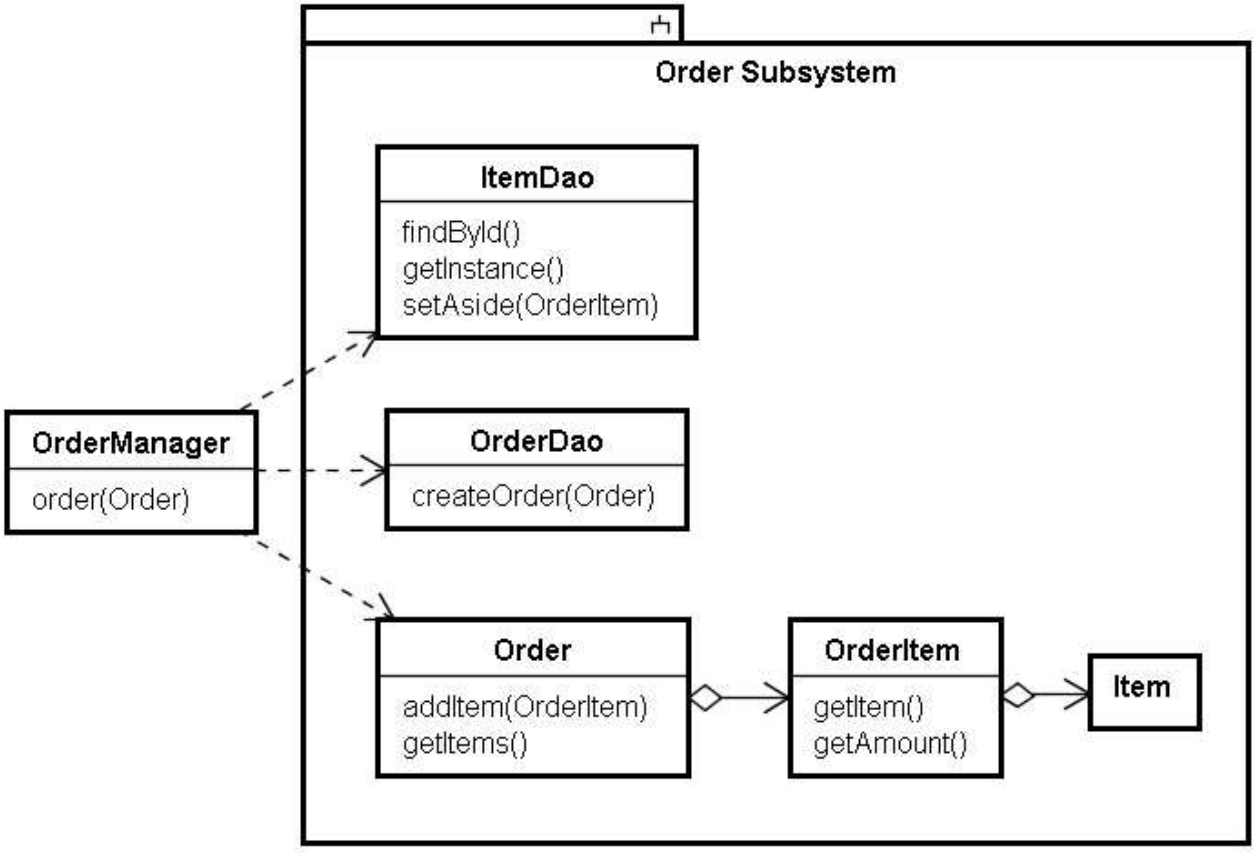
●facade_clientクラス（facade_client.php）

```php
<?php
require_once 'Order.class.php';
require_once 'OrderItem.class.php';
require_once 'ItemDao.class.php';
require_once 'OrderManager.class.php';
```

```
?>
<?php
    $order = new Order();
    $item_dao = ItemDao::getInstance();

    $order->addItem(new OrderItem($item_dao->findById(1), 2));
    $order->addItem(new OrderItem($item_dao->findById(2), 1));
    $order->addItem(new OrderItem($item_dao->findById(3), 3));

    /**
     * 注文処理はこの1行だけ
     */
    OrderManager::order($order);

?>
```

クライアント側ではOrderオブジェクトに商品を追加し、注文処理を実行しています。実際の注文処理が1行で実現されていることに注目してください。

最後にFacadeパターン適用後のクラス図を確認しておきましょう。



Facadeパターンのオブジェクト指向的要素

Facadeパターンは非常に大きな「カプセル化」をおこなうパターンです。

クライアントの視点でサブシステムを見てみると、Facadeクラスで提供された統一APIのみが見えます。ここで、サブシステム自体を「非常に大きなクラス」と捉えると、提供された統一APIはメソッドに相当します。そのAPIの向こうには何やら複雑なものがあるのですが、クライアントからは見えませんし、意識する必要もありません。実際には、Facadeクラスの内部では背後にあるサブシステム内部のクラス群を使って、本来クライアントがおこなうべき複雑な処理をおこないます。

つまりFacadeパターンは、クラスの集まりであるサブシステムに対してカプセル化をおこなっていると言えます。カプセル化とは、データとその操作をまとめて「オブジェクト」として定義し、オブジェクト内部の動作や構造を隠蔽することですが、Facadeパターンではサブシステムの内部状態や構造、またその複雑さを見事に隠蔽しています。

関連するパターン

Abstract Factoryパターン

プラットフォームに依存するクラスを隠蔽するために、Facadeパターンの代わりに利用される場合があります。

Mediatorパターン

Mediatorパターンもクラスの集まりを対象にしたパターンです。

Facadeパターンはサブシステムを利用するためのAPIを抽出しますが、Mediatorパターンはクラスどうしのやりとり自体を抽出するパターンです。

まとめ

ここでは複雑なクラス関係をシンプルにするための統一インターフェースを提供するFacadeパターンについて見てきました。