


Abstract Factory～関連する部品をまとめて作る工場

 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

生成+オブジェクト

はじめに

ここではAbstract Factoryパターンについて説明します。

「abstract factory」を直訳すると「抽象的な工場」となりますね。抽象的な工場・・・これは一体何なのでしょう？

GoFパターンの一つにFactory Methodパターンがあります。Factory Methodパターンは製品であるオブジェクトを作る「工場」を用意するパターンです。ここで見ていくAbstract Factoryパターンも同様にオブジェクトを生成するパターンの1つで、関連し合うオブジェクトの集まり、つまり部品の集まりを生成します。

ここで、「抽象的な」というところがポイントです。この工場は「抽象的」な工場で、生成される部品も「抽象的」なものになります。

オブジェクト指向プログラミングでは、「物を抽象化する」ということが重要なポイントとなります。つまり、具体的な物を直接扱うのではなく、「具体的な物を抽象化した物」を扱うということです。

Abstract Factoryパターンは、具体的なクラスを明確にすることなく、関連し合うオブジェクトの集まりを生成するパターンです。

たとえば

データベースを使ったアプリケーションを構築する場合を考えてみましょう。

近年の開発・設計方法論では、データベースに関する処理内容とそれ以外のアプリケーション固有の処理(ビジネスロジック)とを分けて設計・実装することが主流になっています。なぜなら、接続やデータの取得、トランザクションなどデータベースだけにに関する処理は、本来ビジネスロジックとは関係がないためです。

この手法は**DAO(Data Access Object)**パターンと呼ばれます。DAOパターンを適用する場合、通常エンティティ(データベースの表)の単位でクラスを作成します。

さて、データベースを使ったアプリケーションには、当然ですが何らかのデータベースが必要になります。しかし、プログラムを作成できても肝心のデータベースがないと動かない、ひいてはテストできないといった状況になりがちです。しかし、場合によってはデータベースが用意できないまま先行して開発を行わなければならないこともあります。このような場合、データベースアクセスをすると見せかける擬似的なオブジェクト(**モックオブジェクト**)を使うと非常に有効です。このオブジェクトは「アクセスしているように見せかける」だけで、実際にはデータベースにアクセスしません。つまり、データベースがなくても、ビジネスロジック側を作成したりテストできます。

しかし、モックオブジェクトを使って開発した場合でも、最終的には実際にデータベースにアクセスするクラスに切り替える必要があります。

これはかなり大変な作業になります。しかし、ここでビジネスロジックを記述したコードを書き換えてしまっただけでは再度テストをおこなわなければなりません。これでは何のためにモックオブジェクトを用意したのか分からなくなってしまいます。

データベースアクセスクラスの集まりを整合性を保ったまま簡単に切り替える・・・このような場面で**Abstract Factoryパターン**が活躍します。

Abstract Factoryパターンとは？

Abstract Factory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。

互いに関連したり依存し合うオブジェクト群を、その具象クラスを明確にせずに生成するためのインターフェースを提供する。

Abstract Factoryパターンでは、部品の役割を持つクラスとその部品を作る工場の役割を持つクラスが存在します。

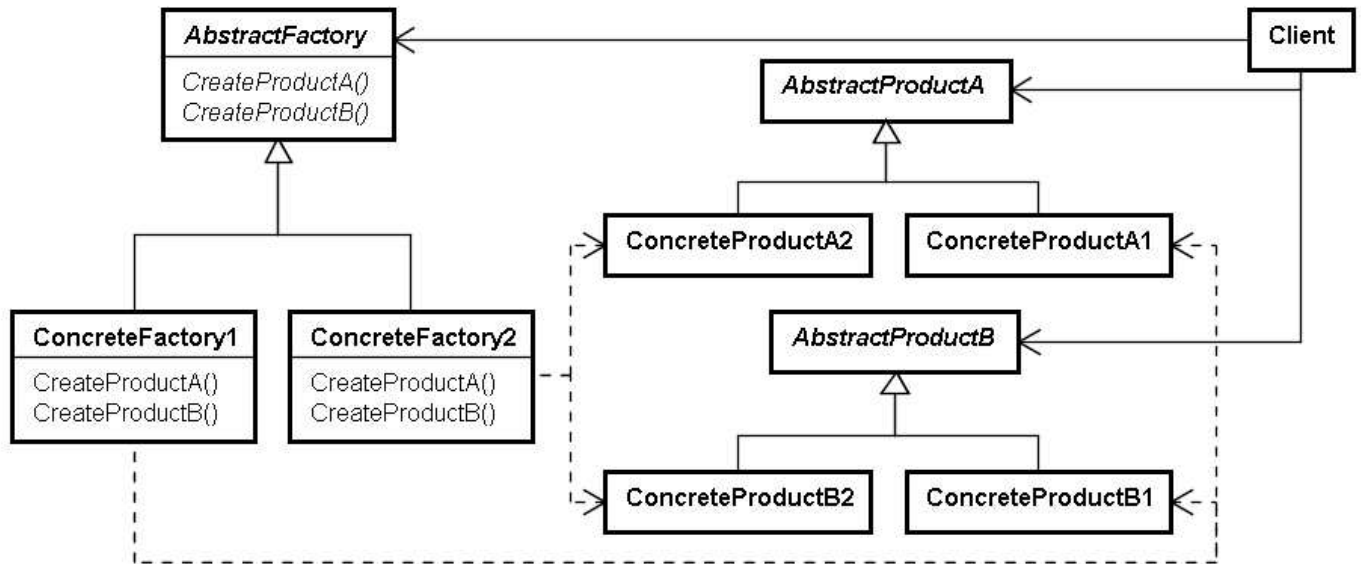
ただし、その工場には関連し合う部品を生成するためのメソッドがそれぞれ用意されます。また、関連し合う部品群の種類に応じて、その工場自身も「工場を生成するための工場」によって生成されます。

これにより、状況に応じて生成される具体的な部品群を切り替えることができます。

まとめると、Abstract Factoryパターンでは工場から生成される部品は抽象化されており、部品の**具体的な内容や生成手順をクライアントが意識しなくて済む**ようになります。

Abstract Factoryパターンの構造

Abstract Factoryパターンのクラス図と構成要素は、次のようになります。



AbstractFactoryクラス

部品であるAbstractProductクラスを生成するためのAPIを定義するクラスです。生成するのがConcreteProductクラスではなくAbstractProductクラスとして定義されているところがポイントです。

ConcreteFactoryクラス

AbstractFactoryクラスのサブクラスで、定義された生成メソッドを実装するクラスです。ここには、具体的な部品であるConcreteProductクラスを返すよう実装されます。

AbstractProductクラス

部品ごとのAPIを定義するクラスです。

ConcreteProductクラス

AbstractProductクラスのサブクラスで、ConcreteFactoryクラスから返される具体的な部品クラスです。

Clientクラス

AbstractFactoryクラスとAbstractProductクラスのAPIのみを利用してプログラミングをおこないます。

Abstract Factoryパターンのメリット

Abstract Factory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。

具体的なクラスをクライアントから隠蔽する

クライアントは具体的な工場や部品に直接アクセスするのではなく、抽象化された工場や部品のAPIのみを使って部品オブジェクトを生成したりアクセスしたりできます。このため、クライアントを変更することなく、具体的な工場や部品を変更することが可能です。

利用する部品群の整合性を保つ

関連し合うオブジェクトを扱う場合、その整合性を保つことが重要になります。Abstract Factoryパターンを適用することで、オブジェクト群の整合性を容易に保つことができます。

部品群の単位で切り替えができる

ConcreteFactoryクラスは関連する部品の集まりを生成します。つまり、ConcreteFactoryクラスを切り替えることで、関連する部品の集まりを容易に切り替えることができます。

Abstract Factoryパターンの適用例

Abstract Factoryパターンの適用例を見てみましょう。

ここでは、商品情報と注文情報を取得するアプリケーションを示しています。Abstract Factoryパターンを適用し、実データを扱う部品群とダミーデータを扱う部品群を一度に切り替えられるようにしています。また、先ほどのDAOパターンを適用して、データの取得部分を隠蔽しています。

まずは、AbstractFactoryクラスに相当するクラスから見ていきましょう。

DaoFactoryインターフェースは、2つの部品を生成するためのAPIが定義されています。createItemDaoメソッドとcreateOrderDaoメソッドです。特に難しいところはありません。

●DaoFactoryインターフェース(DaoFactory.class.php)

```

<?php
interface DaoFactory {
    public function createItemDao();
    public function createOrderDao();
}
?>
  
```

次は、DaoFactoryインターフェースを実装したクラスです。それぞれのcreateItemDaoメソッドとcreateOrderDaoメソッドに注目してください。DbFactoryクラスは実データ、MockFactoryクラスはダミーデータをそれぞれ扱うDAOオブジェクトを生成していることが分かります。

●DbFactoryクラス(DbFactory.class.php)

```
<?php
require_once 'DaoFactory.class.php';
require_once 'DbItemDao.class.php';
require_once 'DbOrderDao.class.php';
?>
<?php
class DbFactory implements DaoFactory {
    public function createItemDao() {
        return new DbItemDao();
    }
    public function createOrderDao() {
        return new DbOrderDao($this->createItemDao());
    }
}
```

●MockFactoryクラス(MockFactory.class.php)

```
<?php
require_once 'DaoFactory.class.php';
require_once 'MockItemDao.class.php';
require_once 'MockOrderDao.class.php';
?>
<?php
class MockFactory implements DaoFactory {
    public function createItemDao() {
        return new MockItemDao();
    }
    public function createOrderDao() {
        return new MockOrderDao();
    }
}
```

続いて、DAOに関連するクラス群を見ていきましょう。

ItemDaoインターフェースは商品情報、OrderDaoインターフェースは注文情報をそれぞれ取得するためのAPIを定義しています。これらのインターフェースは、AbstractProductクラスに相当します。

このサンプルアプリケーションでは、findByIdメソッドにIDを渡すことで情報を取り出せるようにしています。

●ItemDaoクラス(ItemDao.class.php)

```
<?php
interface ItemDao {
    public function findById($item_id);
}
```

●OrderDaoクラス(OrderDao.class.php)

```
<?php
interface OrderDao {
    public function findById($order_id);
}
```

次はConcreteProductクラスに相当するクラスです。

先のAbstractFactoryクラスとConcreteFactoryクラスの関係と同様に、実データを扱うクラスとダミーデータを扱うクラスに分かれています。

実データを扱うクラスはDbItemDaoクラスとDbOrderDaoクラスです。このサンプルアプリケーションでは、商品情報と注文情報をテキストファイルに保存しており、DbItemDaoクラスとDbOrderDaoクラスがインスタンス化されるときに読み込まれ、商品・注文の各オブジェクトが生成されるようになっています。

データファイルと商品クラス、注文クラスについては、後ほど説明します。

●DbItemDaoクラス(DbItemDao.class.php)

```
<?php
require_once 'ItemDao.class.php';
require_once 'Item.class.php';
?>
<?php
class DbItemDao implements ItemDao {
    private $items;
    public 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));

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

fclose($fp);

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

```

●DbOrderDaoクラス(DbOrderDao.class.php)

```

<?php
require_once 'OrderDao.class.php';
require_once 'Order.class.php';
?>
<?php
class DbOrderDao implements OrderDao {
    private $orders;
    public function __construct(ItemDao $item_dao) {
        $fp = fopen('order_data.txt', 'r');

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

        $this->orders = array();
        while ($buffer = fgets($fp, 4096)) {
            $order_id = trim(substr($buffer, 0, 10));
            $item_ids = trim(substr($buffer, 10));

            $order = new Order($order_id);
            foreach (split(',', $item_ids) as $item_id) {
                $item = $item_dao->findById($item_id);
                if (!is_null($item)) {
                    $order->addItem($item);
                }
            }
            $this->orders[$order->getId()] = $order;
        }

        fclose($fp);

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

```

一方のダミーデータを扱うクラスは、MockItemDaoクラスとMockOrderDaoクラスとなります。これらのクラスは、決まった商品データと注文データを返します。

●MockItemDaoクラス(MockItemDao.class.php)

```

<?php
require_once 'ItemDao.class.php';
require_once 'Item.class.php';
?>
<?php
class MockItemDao implements ItemDao {
    public function findById($item_id) {
        $item = new Item('99', 'ダミー商品');
        return $item;
    }
}
?>

```

●MockOrderDaoクラス (MockOrderDao.class.php)

```
<?php
require_once 'OrderDao.class.php';
require_once 'Order.class.php';
?>
<?php
class MockOrderDao implements OrderDao {
    public function findById($order_id) {
        $order = new Order('999');
        $order->addItem(new Item('99', 'ダミー商品'));
        $order->addItem(new Item('99', 'ダミー商品'));
        $order->addItem(new Item('98', 'テスト商品'));

        return $order;
    }
}
```

次に、商品クラスと注文クラスを説明します。これらのクラスは、純粋に商品の情報や注文の情報を内部に保持するだけの役割を担っています。

商品を表すItemクラスは、コンストラクタに商品情報(商品IDと商品名)を受け取り、その値にアクセスするためのメソッドが用意されているだけの単純なクラスです。

●Itemクラス (Item.class.php)

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

注文を表すOrderクラスは、コンストラクタで注文IDを受け取り、注文情報に含まれる商品情報はaddItemメソッドを通じて追加します。また、注文IDや含まれている商品情報へアクセスするためのメソッドも用意されています。

●Orderクラス (Order.class.php)

```
<?php
class Order {
    private $id;
    private $items;
    public function __construct($id) {
        $this->id = $id;
        $this->items = array();
    }
    public function addItem(Item $item) {
        $id = $item->getId();
        if (!array_key_exists($id, $this->items)) {
            $this->items[$id]['object'] = $item;
            $this->items[$id]['amount'] = 0;
        }
        $this->items[$id]['amount']++;
    }
    public function getItems() {
        return $this->items;
    }
    public function getId() {
        return $this->id;
    }
}
```

ここで、実データについて見ておきましょう。

商品情報と注文情報は、固定長データとして用意しました。フォーマットはデータファイル内の先頭行にもありますが、詳細は表を参照してください。

●商品情報 (item_data.txt)

商品ID	商品名
1	限定Tシャツ
2	ぬいぐるみ
3	クッキーセット

項目 開始位置 終了位置 備考

商品ID 1 20

商品名 21 -

●注文情報 (order_data.txt)

注文ID 商品ID

1 3

2 1, 3

3 1, 2, 3

4 2

5 2, 3

項目 開始位置 終了位置 備考

注文ID 1 10

商品ID 11 - 複数の場合、商品IDをカンマ区切りで繋げる

ようやく最後のクライアント側コードの説明です。注目していただきたいのが、このコードには具体的な部品クラス、つまり**ConcreteProduct**クラスに相当するクラスが一切出てきていないということです。

唯一登場しているのがConcreteFactoryクラスに相当するDbFactoryクラスとMockFactoryクラスです。つまり、ConcreteFactoryクラスを切り替えるだけで、部品群を綺麗に切り替えることができます。

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

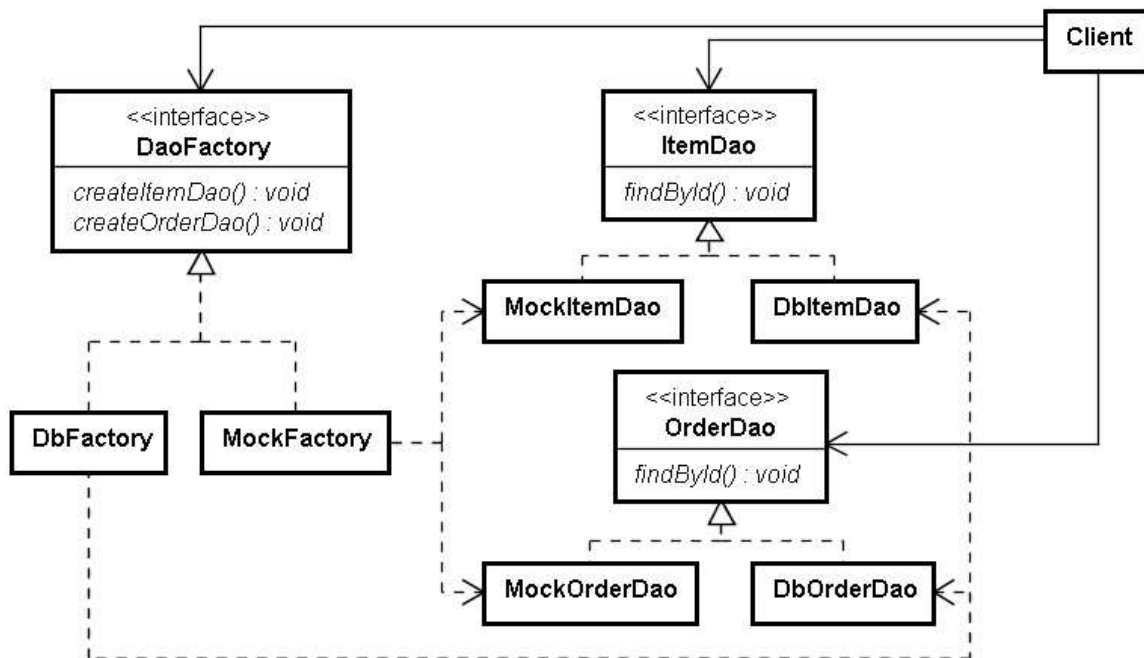
```
<?php
if (isset($_POST['factory'])) {
    $factory = $_POST['factory'];

    switch ($factory) {
        case 1:
            include_once 'DbFactory.class.php';
            $factory = new DbFactory();
            break;
        case 2:
            include_once 'MockFactory.class.php';
            $factory = new MockFactory();
            break;
    }

    $item_id = 1;
    $item_dao = $factory->createItemDao();
    $item = $item_dao->findById($item_id);
    echo 'ID=' . $item_id . 'の商品は「' . $item->getName() . '」です<br>';

    $order_id = 3;
    $order_dao = $factory->createOrderDao();
    $order = $order_dao->findById($order_id);
    echo 'ID=' . $order_id . 'の注文情報は次の通りです。';
    echo '<ul>';
    foreach ($order->getItems() as $item) {
        echo '<li>' . $item['object']->getName();
    }
    echo '</ul>';
}
??
<hr>
<form action="" method="post">
    <div>
        DaoFactoryの種類:
        <input type="radio" name="factory" value="1">DbFactory
        <input type="radio" name="factory" value="2">MockFactory
    </div>
    <div>
        <input type="submit">
    </div>
</form>
```

最後にAbstract Factoryパターンを適用したアプリケーションのクラス図は次のようになっていますので確認してみてください。



Abstract Factoryパターンのオブジェクト指向的要素

Abstract Factoryパターンは「**ポリモーフィズム**」を活用したパターンです。

具体的な部品クラスであるConcreteProductクラスは、それぞれの部品ごとに共通のAPIを持っています。これは親クラスであるAbstractProductクラスで定義されています。つまり、クライアント側は**AbstractProductクラスのAPI**だけを使ってプログラミングすることで、**具体的なConcreteProductクラスに依存しないコード**になります。

このことは、部品工場であるAbstractProductクラスとConcreteFactoryクラスにも言えます。この2つのクラスには親子関係がありますが、親クラスであるAbstractFactoryクラスで提供されるAPIだけを使うことで、**どのConcreteFactoryクラスなのかを意識する必要がなくなり**、具体的なConcreteFactoryクラスに依存しないコードになります。

また、工場から生成される部品はAbstractProductクラスとして生成されますので、ここでもどのConcreteProductクラスなのかを意識する必要がありません。

このため、どのConcreteFactoryクラスを使うかを切り替えるだけで、芋づるのように生成されるConcreteProductクラス群を切り替えることができます。

オブジェクト指向プログラミングでは、具体的なクラスではなくインターフェースや抽象化されたクラスのAPIに対してプログラミングすることが重要になります。

関連するパターン

Factory Methodパターン、Prototypeパターン

AbstractFactoryクラスはFactory Methodパターンを使って実装される場合が多いですが、Prototypeパターンを適用して実装することも可能です。

Singletonパターン

ConcreteFactoryクラスにSingletonパターンが適用される場合が多いです。

まとめ

ここでは関連し合う部品群を生成するAbstract Factoryパターンについて見てきました。