Builder ～生成の手順と手段を分離する

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

生成＋オブジェクト

はじめに

ここではBuilderパターンについて見ていきましょう。

「builder」とは、「建築者」や「建設業者」の意味を持つ単語ですが、たとえば、家を建てることを考えてみましょう。最終的に建てられる家は、「どの様な順序で、どこに何を配置していくか」という「手順」と、「柱や壁、屋根に何を使うか」という「材料」によって大きく違ってきます。

Builderパターンは、この「手順」と「材料」を分けておき、同じ手順で異なるオブジェクトを生成させるパターンです。

たとえば

ある条件によって作成するオブジェクトを変更するといった場合を考えてみましょう。

普通に考えると、if文やswitch文で処理を分岐し、生成するオブジェクトを切り替えるだけで済んでしまいそうですね。

しかし、オブジェクトを生成するのに複雑な手順が必要だった場合はどうでしょうか？大量のif文やswitch文で処理を分岐してやる必要がありそうですね。

こういった場合、「何を作るのか」に依存しないように具体的な生成手順をまとめておけると、簡単に色々なオブジェクトを生成できそうです。

もう一つ、あるフォーマットに従ったデータを読み込んで、そのデータを格納するクラスを考えてみましょう。この場合、データの解析処理はどこに記述すると良いでしょうか？

たとえば、このクラスの中でデータを処理させるとしましょう。この場合、データを渡すだけでクラスのインスタンスが生成でき、またデータの処理もクラスの内部に閉じこめることができます。

しかし、データの処理が非常に複雑な場合はどうでしょうか？このクラスのコードの大部分は、その処理に関するコードになってしまうでしょう。このクラスの本来の目的は、データを格納することのはずですが、データの解析処理も含めると非常に分かりにくいコードになってしまいます。

データを保持するクラスにはできるだけその目的に専念させ、データの処理部分は他のクラスに任せたいものです。そうすればクラスの「責任」がはっきりし、クラスの構造がシンプルになりますので、コードも分かりやすいものになるはずです。また、クラスの再利用性も高まると予想されます。

このように、オブジェクトを「何を生成するか」と「どのように生成するか」を分離するパターンが**Builderパターン**です。

Builderパターンとは？

Builder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。

> 複合オブジェクトについて、その作成過程を表現形式に依存しないものにすることにより、
> 同じ作成過程で異なる表現形式のオブジェクトを生成できるようにする

Builderパターンは、オブジェクトの生成に注目したパターンで、オブジェクトの「生成手順」と「生成手段」を分離するパターンです。

Builderパターンでは、まずクライアントがどのようなオブジェクトを生成するかを選択します。つまり「材料」を選択するということです。この材料は、最終的に生成されるオブジェクトの生成処理を知っています。この材料を「建築者」に渡すこ

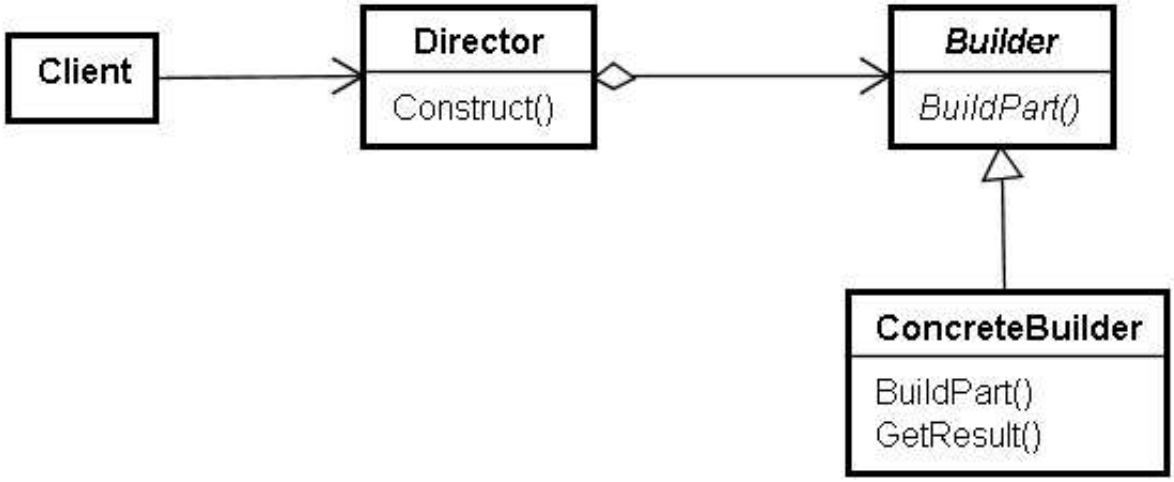
とで、実際のオブジェクトの生成をお願いします。

一方の「建築者」ですが、渡された材料からどの様なものができるのか知りません。ただ、自らが知っている手順に沿って、オブジェクトの生成をおこなうだけです。

1つのクラスに生成手順と生成手段をすべてまとめた場合、クラスが複雑になりすぎる傾向があります。ここでBuilderパターンを適用して生成手順を分離することで、構造がシンプルになり、再利用性も高まります。

Builderパターンの構造

Builderパターンのクラス図と構成要素は、次のとおりです。



Builderクラス

オブジェクトの「生成手段」を提供するクラス群で最上位に位置するクラスです。オブジェクトを生成するためのAPIを定義します。

ConcreteBuilderクラス

Builderクラスで提供されるAPIを実装するサブクラスです。また、生成したオブジェクトを取得するためのメソッドを提供します。

Directorクラス

「建築者」に相当するクラスで、Builderクラスで定義されたAPIを使ってオブジェクトを生成します。

Productクラス

最終的に生成されるオブジェクトのクラスです。

Builderパターンのメリット

Builder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。

Productオブジェクトの生成過程や生成手段を隠すことができる

Builderクラスでは、Directorクラスにオブジェクトを生成するためのAPIを提供しています。Directorクラスでは、このAPIを使ってのみProductオブジェクトを生成します。つまり、DirectorクラスはProductオブジェクトの生成過程やProductオブジェクトの生成手段を知りません。このため、新しいProductオブジェクトを作る必要がある場合、新しいConcreteBuilderクラスを追加するだけで済みます。

オブジェクトの生成過程や生成手段のコードを局所化できる

Builderパターンは、オブジェクトの生成過程と生成手段を分離するパターンです。Directorクラスにはオブジェクト生成過程のコードだけが、ConcreteBuilderクラスにはオブジェクトの生成手段のコードだけが記述されることになります。つまり、生成過程と生成手段それぞれに関するプログラムコードを凝縮できるということです。この結果、生成過程、生成手段を独立して修正・拡張することが可能になります。

Builderパターンの適用例

Builderパターンの適用例を見てみましょう。

これは、ニュース一覧をインターネット経由で取得し、一覧表示するアプリケーションです。

まずは、1つの記事を格納するNewsクラスから始めましょう。Newsクラスは、記事のタイトル、URL、記事の対象日付を保持するだけのクラスで、これら3要素をコンストラクタで受け取ります。特に問題はありませんね。

●Newsクラス(News.class.php)

```
<?php
class News {
    private $title;
    private $url;
    private $target_date;

    public function __construct($title, $url, $target_date) {
        $this->title = $title;
        $this->url = $url;
        $this->target_date = $target_date;
    }

    public function getTitle() {
        return $this->title;
    }

    public function getUrl() {
        return $this->url;
    }

    public function getDate() {
        return $this->target_date;
    }
}
?>
```

続けて、外部のサイトからニュース記事を取得し、Newsオブジェクトの配列を作成するクラスたちです。

まずは、最終的なNewsオブジェクトの配列を生成するNewsDirectorクラスです。Directorクラスに相当します。コンストラクタの引数に、NewsBuilder型のオブジェクトが指定できるようになっています。また、実際のNewsオブジェクトの生成は、内部に保持したNewsBuilderオブジェクトを使って生成しています。

●NewsDirectorクラス(NewsDirector.class.php)

```
<?php
require_once 'NewsBuilder.class.php';
?>
<?php
/**
 * Directorクラスに相当する
 */
class NewsDirector {
    private $builder;
    private $url;

    public function __construct(NewsBuilder $builder, $url) {
        $this->builder = $builder;
        $this->url = $url;
    }

    public function getNews() {
        $news_list = $this->builder->parse($this->url);
        return $news_list;
    }
```

```
}
?>
```

次に、先ほど出てきたNewsBuilderクラスについて説明しましょう。

これはインターフェースとして宣言しており、parseメソッドを定義しています。先のNewsDirectorクラスは、このparseメソッドを使ってNewsオブジェクトの配列を取得しています。

●NewsBuilderクラス(NewsBuilder.class.php)

```
<?php
/**
 * Builderクラスに相当する
 */
interface NewsBuilder {
    public function parse($data);
}
?>
```

先ほどのNewsBuilderクラスを実装するクラスが、RssNewsBuilderクラスです。名前からも分かるように、ニュース記事としてRSSからデータを取得します。

RSS(RDF Site Summary)は、ウェブログや新聞社、その他企業でも更新情報の配信などに利用されている文章フォーマットです。

今回は、PHP5から導入されたSimpleXML拡張機能を利用しています。

●RssNewsBuilderクラス(RssNewsBuilder.class.php)

```
<?php
require_once 'News.class.php';
require_once 'NewsBuilder.class.php';
?>
<?php
/**
 * ConcreteBuilderクラスに相当する
 */
class RssNewsBuilder implements NewsBuilder {
    public function parse($url) {
        $data = simplexml_load_file($url);
        if ($data === false) {
            throw new Exception('read data [' .
                                htmlspecialchars($url, ENT_QUOTES)
                                . '] failed !');
        }

        $list = array();
        foreach ($data->item as $item) {
            $dc = $item->children('http://purl.org/dc/elements/1.1/');
            $list[] = new News($item->title,
                               $item->link,
                               $dc->date);
        }
        return $list;
    }
}
?>
```

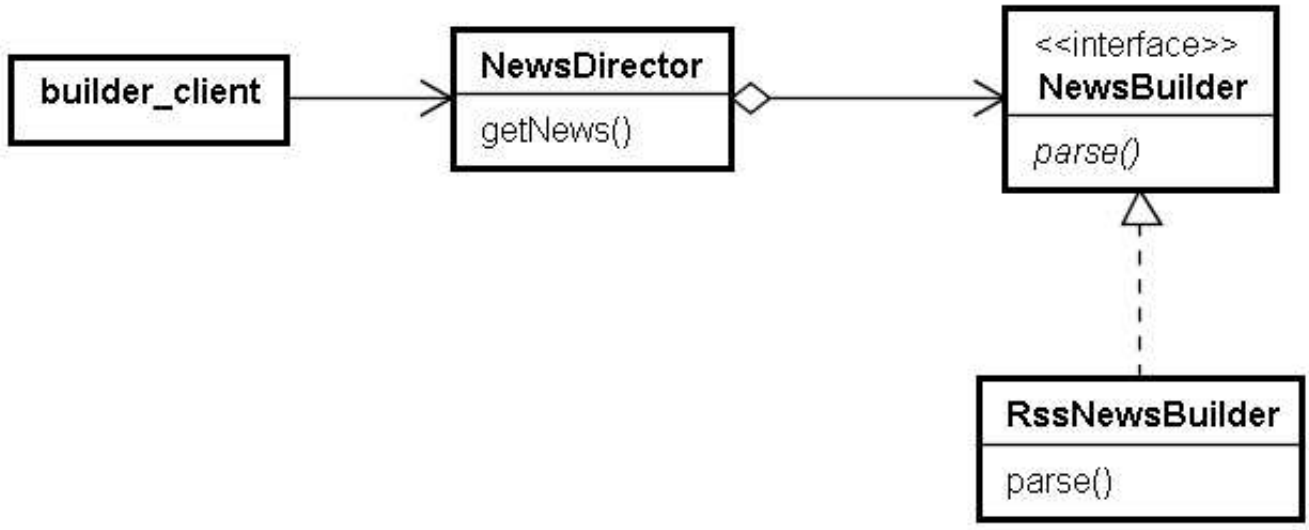
最後にクライアント側のコードです。NewsDirectorクラスのコンストラクタに、RssNewsBuilderオブジェクトとURLを渡していますね。たとえば、他のフォーマットで記述された記事データからNewsオブジェクトの配列を生成する場合、このRssNewsBuilderオブジェクトを変更するだけで対応可能になります。これは、「何を生成するか」と「どのように生成するか」を切り分けた結果、可能になるものです。

●クライアント側コード(builder_client.php)

```
<?php
require_once 'NewsDirector.class.php';
require_once 'RssNewsBuilder.class.php';
?>
<?php
    $builder = new RssNewsBuilder();
    $url = 'http://www.php.net/news.rss';

    $director = new NewsDirector($builder, $url);
    foreach ($director->getNews() as $article) {
        printf('<li>[%s] <a href="%s">%s</a></li>',
               $article->getDate(), $article->getUrl(), $article->getTitle());
    }
?>
```

最後にサンプルコードのクラス図を示しておきます。



Builderパターンのオブジェクト指向的要素

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

「建築者」であるDirectorクラスには、「材料」であるBuilder型のオブジェクト、具体的にはConcreteBuilderクラスのインスタンスが渡されます。ここで、渡されたBuilder型のオブジェクトが具体的にどのクラスのインスタンスなのかを知りません。しかし、Builderクラスで提供されているメソッドは知っていますし、それらを呼び出すことで目的とするオブジェクトが生成されることも知っています。ただ、具体的にどの様なオブジェクトが生成されるかは知りません。

ここまで見てきたように、DirectorクラスはBuilderクラスのAPIしか知りません。それらAPIの具体的な処理内容は、BuilderクラスのサブクラスであるconcreteBuilderクラスで実装されています。つまり、ConcreteBuilderクラスがどのようなものであれ、BuilderクラスのAPIが実装されているものであれば、Directorクラスは正しく機能するということになります。

この「具体的な実装が何かを知らない」ことが、オブジェクト指向プログラミングでは重要になってきます。また「知らないからこそ入れ替えが可能」になるのです。

また、DirectorクラスとBuilderクラスの間には、強い結びつきはありません。Builder型のオブジェクトをDirectorクラスの内部に保持することで、緩く結びつけられています。Directorクラスの処理内容は、内部に保持したBuilder型のオブジェクトのメソッドを呼び出し、具体的な処理を任せています。この関係を委譲と呼びます。この緩い結びつけのため、

Builder型のオブジェクトをプログラムの実行中に変更したりできるのです。

関連するパターン

Abstract Factoryパターン

Abstract Factoryパターンも複雑なオブジェクトを生成するパターンです。Builderパターンは複雑なオブジェクトを段階的に生成していく手順に注目したパターンですが、Abstract Factoryパターンは、それぞれの部品の集まりに注目したパターンです。

Compositeパターン

Builderパターンによって生成されるオブジェクトは、Compositeパターンになる場合があります。

まとめ

ここでは、「何を生成するか」と「どのように生成するか」を切り分けるBuilderパターンを見てきました。