


Factory Method～インスタンスを生成する工場

 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＋クラス

はじめに

ここでは、Factory Methodパターンについて説明します。

「factory」とはもちろん「工場」という意味です。パターン名に使われているほどですから、もちろんそれなりの理由があります。工場とは一般的に何かを製造する施設ですよね・・・ここまで言うと、カンのするどい人でしたらもう気づいているかもしれません。

では、なぜ「工場」という名前が使われているのか、これから説明していきます。

たとえば

たとえば、作曲家と作品のデータがCSVファイルとして保存されており、これを表示する場合を考えてみましょう。

CSVファイルの1レコードには作曲家名とその作曲家の作品名が存在します。表示ルールとして、複数の作品を持つ作曲家の場合は初めに作曲家名を表示し、続けて作品名を表示するものとします。

●Music.csv

ルートヴィヒ・ヴァン・ベートーヴェン, ピアノソナタ第23番ヘ短調「熱情」
ヴォルフガング・アマデウス・モーツァルト, 魔笛
ヴォルフガング・アマデウス・モーツァルト, セレナーデ長調 K. 525「小夜曲」
アントニン・ドヴォルザーク, 交響曲第9番ホ短調「新世界より」

ここで、簡単に以下のようなコードを書いてみます。

●DispMusic.class.php

```
<html lang="ja">
<head>
<title>作曲家と作品たち</title>
</head>
<body>
<?php
    $handle = fopen ("Music.csv", "r");
    $column = 0;
    $tmp = "";
    while ($data = fgetcsv ($handle, 1000, ",", "")) {
        $num = count ($data);
        for ($c = 0; $c < $num; $c++) {
            if($c == 0) {
                if($column != 0 && $data[$c] != $tmp) {
                    echo "</ul>";
                }
                if($data[$c] != $tmp) {
                    echo "<b>" . $data[$c] . "</b>";
                    echo "<ul>";
                    $tmp = $data[$c];
                }
            }
        }
    }
}
```

```

        }else {
            echo "<li>";
            echo $data[$c];
            echo "</li>";
        }
        $column++;
    }
    echo "</ul>";
    fclose ($handle);
?>
</body>
</html>

```

DispMusic.class.phpでは、CSVファイルを読み込みながらHTMLを出力しています。よくありがちなコードですね。特に問題はないでしょう。

しかし世の中そうは簡単にいかないものです。たとえば、次のような修正をおこなう必要が出てきたとします。

- CSV形式のデータを利用する他にXML形式のデータをサポートする
- 表示はCSVの場合と同じとする
- 読み込み、表示に利用するデータファイルは外部から渡される
- CSV、XMLデータのどちらを利用するかは、渡されたファイルの拡張子で判断する

単純に考えると、if文を使ってCSVファイルなのかXMLファイルを判定しながら、データの読み込みと表示をおこなえば、ひとまずは要求を満たすことができるでしょう。

●DispMusic2.class.php

```

<html lang="ja">
<head>
<title>作曲家と作品たち</title>
</head>
<body>
<?php
/**
 * 外部からの入力ファイルです
 */
$filename = "Music.csv";

/**
 * ファイル形式を判断します
 */
$poscsv = strpos($filename, '.csv');
$posxml = strpos($filename, '.xml');

if($poscsv !== false) {
    $handle = fopen ("Music.csv", "r");
    $column = 0;
    $tmp = "";
    while ($data = fgetcsv ($handle, 1000, ",", "")) {
        $num = count ($data);
        for ($c = 0; $c < $num; $c++) {
            if($c == 0) {
                if($column != 0 && $data[$c] != $tmp) {
                    echo "</ul>";
                }
                if($data[$c] != $tmp) {
                    echo "<b>" . $data[$c] . "</b>";
                    echo "<ul>";
                    $tmp = $data[$c];
                }
            }
            }else {
                echo "<li>";
                echo $data[$c];
            }
        }
    }
}

```

```

        echo "</li>";
    }
    $column++;
}
echo "</ul>";
fclose ($handle);
} else if($posxml != false) {
    $xml = simplexml_load_file('Music.xml');
    foreach ($xml->artist as $artist) {
        echo "<b>";
        echo convert($artist['name']) . "</b>";
        echo "<ul>";
        foreach ($artist->music as $music) {
            echo "<li>";
            echo convert($music['name']);
            echo "</li>";
        }
        echo "</ul>";
    }
} else {
    echo "not readable";
}

/**
 * 文字コードの変換を行います
 */
function convert($str) {
    return mb_convert_encoding($str, mb_internal_encoding(), "auto");
}
?>
</body>
</html>

```

しかし、さらなる別形式のデータをサポートする必要がある場合、またデータの読み込み、表示コードを別書き、条件分岐して・・・と、繰り返していくうちに条件分岐の羅列になってしまい、**メンテナンス性が悪くなる**事がわかります。

また、こうなってはデータの読み込み部分を再利用することは難しくなります。

ここで考え方を考えてみましょう。CSV形式やXML形式のデータを利用して達成したい事は、データを表示することです。表示するためにはデータを読み込む必要があります。データの形式がなんであれデータを読み込み、表示する。この2つが共通な機能として存在します。

共通の機能といっても、当然具体的な読み込み方法や表示方法は共通ではありません。このコードの違いを埋める仕組みができれば、より分かりやすくメンテナンスしやすいコードになりそうです。

このような状況で活躍するのが、**Factory Methodパターン**です。

Factory Methodパターンとは？

Factory Method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。

オブジェクトを生成するときのインタフェースだけを規定して、実際二度のクラスをインスタンス化するかはサブクラスが決めるようにする。Factory Methodパターンは、インスタンス化をサブクラスに任せる。

Factory Methodパターンは、**オブジェクトの生成の方法に注目したパターン**です。Factory Methodパターンとは、その名の通り「工場」のような振る舞いをします。何を作る工場かという、「**クラスのインスタンス**」という製品を製造する工場です。

さて、この「Factory」で生成される「オブジェクト」は**どのように実装されているのかはわかりませんが**、どのように実装されていても、**同様の「機能」を提供**します。この同様の「機能」を実現するために「**インターフェース**」を定義し、Factoryで生成されるクラスが実装を行っています。言い換えると、**Factoryで生成されるインスタンスはどのクラスのものであれ、同様の「機能」を持っている**と言えます。

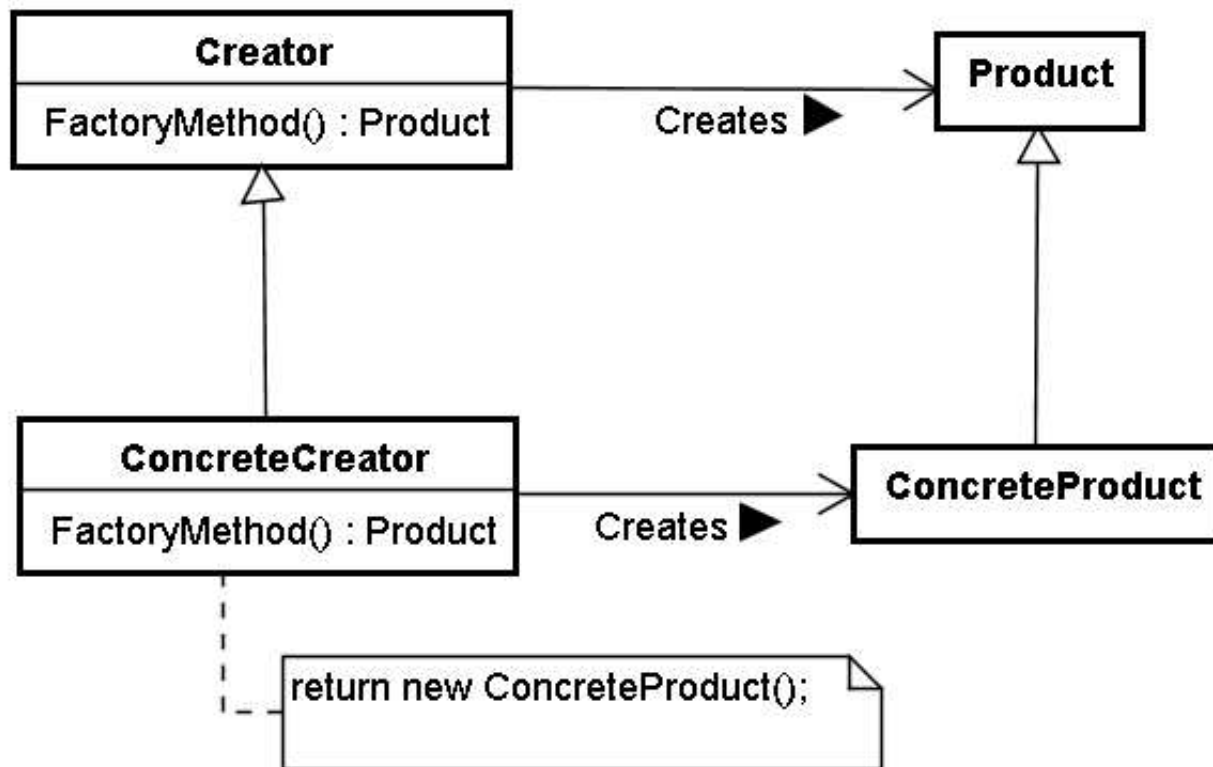
Factory Methodパターンを利用することで、オブジェクトの利用側はどのインスタンスが返ってくるのかを知る必要がなくなります。繰り返しになりますが、**利用側が知る必要のあるのは製品の「機能」であり、具体的にどのクラスのインス**

タンスであるかを知る必要はありません。

また、通常、オブジェクトはnew演算子を使って生成されますが、Factory Methodパターンではその代わりとなるインスタンス生成用のクラスが用意されます。このため、Virtual Constructor(仮想的なコンストラクタ)と呼ばれる事もあります。

Factory Methodパターンの構造

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



Factory Methodパターンの構成要素

Factory Methodパターンの構成要素は、次のとおりです。

Productクラス

オブジェクト生成メソッド(工場)で生成されるオブジェクト(製品)のAPIを定義するクラスです。オブジェクト生成メソッドは、「factory method」とも呼ばれます。

ConcreteProductクラス

Productクラスのサブクラスで、Productクラスで定義されたAPIを実装したクラスです。

Creatorクラス

オブジェクト生成メソッドを提供するクラスです。このメソッドは、Product型のオブジェクトを返します。また、あるConcreteProductオブジェクトを返すために、デフォルトの実装がなされる場合もあります。

ConcreteCreatorクラス

Creatorクラスを継承したサブクラスです。ConcreteProductクラスのインスタンスを返します。

Factory Methodパターンのメリット

Factory Method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。

オブジェクトの生成処理と使用処理を分離できる

複数のオブジェクトを扱う場合、if文やswitch文を使ってオブジェクトの生成コードを記述し、それらを利用するコードも同じコード内に記述してしまいがちです。こういった場合、オブジェクト生成のコードと利用側のコードを分けておくと、後々のメンテナンスが楽になります。

Factory Methodパターンは、「オブジェクト生成側」と「オブジェクトの利用側」を分離するパターンです。「オブジェクト生成側」はCreatorクラスとConcreteCreatorクラス、「オブジェクト利用側」はクライアント側のコードがそれぞれ担います。その間をやりとりするオブジェクトが、ProductクラスとConcreteProductクラスになります。これにより、生成側はProductクラスを返すコード、一方の利用側はProductクラスを利用するコードを記述するだけで良くなり、それぞれの処理内容に専念することができます。

オブジェクトの利用側とオブジェクトのクラスの結びつきを低くする

Factory Methodパターンを使用することで、オブジェクトの利用側とオブジェクトの結びつきを低くする事ができます。これは、利用側でオブジェクトを直接生成しない、つまり、利用側のコードに「new クラス名」と直接書かなくてすむ、ということの意味します。この結果、利用側とオブジェクトの結びつきがゆるくなります。たとえば、生成するクラスの種類や生成手順が変更された場合でも、ファクトリ側を手直しするだけですみます。

Factory Methodパターンの適用例

ここでは冒頭に出てきたデータを表示する例にFactory Methodパターンを適用してみましょう。

まずは、CSVデータとXMLデータを扱うクラスの共通機能をインターフェースとして定義しましょう。ここではReaderとします。このクラスでは共通機能であるデータの読み込みと表示を行うメソッドを定義します。

●Reader.class.php

```
<?php
/**
 * 読み込み機能を表すインターフェースクラスです
 */
interface Reader {
    public function read();
    public function display();
}
?>
```

次に、CSVデータを扱うクラスCSVFileReaderを作成し、このクラスでReaderインターフェースを実装します。

●CSVFileReader.class.php

```
<?php
require_once("Reader.class.php");

/**
 * CSVファイルの読み込みを行なうクラスです
 */
class CSVFileReader implements Reader
{
    /**
     * 内容を表示するファイル名
     *
     * @access private
     */
    private $filename;

    /**
     * データを扱うハンドラ名
     *
     * @access private
     */
    private $handler;

    /**
     * コンストラクタ
     */
}
```

```

*
* @param string ファイル名
* @throws Exception
*/
public function __construct($filename)
{
    if (!is_readable($filename)) {
        throw new Exception('file "' . $filename . '" is not readable !');
    }
    $this->filename = $filename;
}

/**
 * 読み込みを行ないます
 */
public function read()
{
    $this->handler = fopen ($this->filename, "r");
}

/**
 * 表示を行ないます
 */
public function display()
{
    $column = 0;
    $tmp = "";
    while ($data = fgetcsv ($this->handler, 1000, ",", "")) {
        $num = count ($data);
        for ($c = 0; $c < $num; $c++) {
            if($c == 0) {
                if($column != 0 && $data[$c] != $tmp) {
                    echo "</ul>";
                }
                if($data[$c] != $tmp) {
                    echo "<b>";
                    echo $data[$c] . "</b>";
                    echo "<ul>";
                    $tmp = $data[$c];
                }
            } else {
                echo "<li>";
                echo $data[$c];
                echo "</li>";
            }
        }
        $column++;
    }
    echo "</ul>";
    fclose ($this->handler);
}
}
?>

```

同様に、XMLデータを扱うクラスXMLFileReaderも作成しましょう。

●XMLFileReader.class.php

```

<?php
require_once("Reader.class.php");

/**
 * XMLファイルの読み込みを行なうクラスです
 */
class XMLFileReader implements Reader
{

```

```
/**
 * 内容を表示するファイル名
 *
 * @access private
 */
private $filename;

/**
 * データを扱うハンドラ名
 *
 * @access private
 */
private $handler;

/**
 * コンストラクタ
 *
 * @param string ファイル名
 * @throws Exception
 */
public function __construct($filename)
{
    if (!is_readable($filename)) {
        throw new Exception('file "' . $filename . '" is not readable !');
    }
    $this->filename = $filename;
}

/**
 * 読み込みを行いません
 */
public function read()
{
    $this->handler = simplexml_load_file($this->filename);
}

/**
 * 文字コードの変換を行います
 */
private function convert($str) {
    return mb_convert_encoding($str, mb_internal_encoding(), "auto");
}

/**
 * 表示を行いません
 */
public function display()
{
    foreach ($this->handler->artist as $artist) {
        echo "<b>" . $this->convert($artist['name']) . "</b>";
        echo "<ul>";
        foreach ($artist->music as $music) {
            echo "<li>";
            echo $this->convert($music['name']);
            echo "</li>";
        }
        echo "</ul>";
    }
}

?>
```

次にFactoryクラスを作成しましょう。ここではReaderFactoryとします。このクラスはReaderクラスのインスタンスを生成するクラスになります。

ReaderFactoryクラスは、どの形式が指定された場合にどのReaderクラスのインスタンスを生成するかを判断します。

今回はファイルの拡張子で判断するのでしたね。ファイル名が「～.csv」の場合はCSVデータであり、「～.xml」の場合はXMLデータとします。

●ReaderFactory.class.php

```
<?php
require_once('Reader.class.php');
require_once('CSVFileReader.class.php');
require_once('XMLFileReader.class.php');

/**
 * Readerクラスのインスタンス生成を行なうクラスです
 */
class ReaderFactory
{
    /**
     * Readerクラスのインスタンスを生成します
     */
    public function create($filename)
    {
        $reader = $this->createReader($filename);
        return $reader;
    }

    /**
     * Readerクラスのサブクラスを条件判定し、生成します
     */
    private function createReader($filename)
    {
        $poscsv = strpos($filename, '.csv');
        $posxml = strpos($filename, '.xml');

        if($poscsv !== false) {
            $r = new CSVFileReader($filename);
            return $r;
        } else if($posxml !== false) {
            return new XMLFileReader($filename);
        } else {
            die('This filename is not supported : ' . $filename);
        }
    }
}
?>
```

最後に大元のコードを変更しましょう。CSVファイルやXMLファイルの読み込み部分を、今回作成したクラスを利用するように変更します。

●DispMusic3.class.php

```
<?php
require_once('ReaderFactory.class.php');
?>
<html lang="ja">
<head>
<title>作曲家と作品たち</title>
</head>
<body>
<?php
    /**
     * 外部からの入力ファイルです
     */
    $filename = 'Music.xml';
```



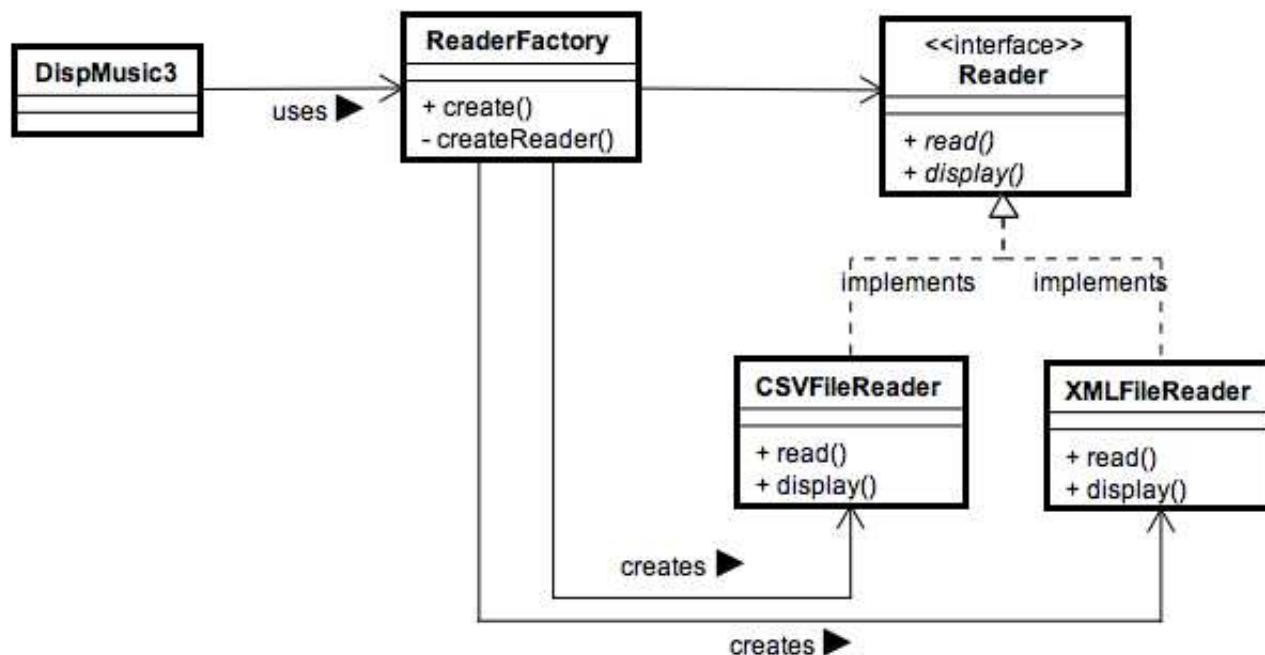
```

$factory = new ReaderFactory();
$data = $factory->create($filename);
$data->read();
$data->display();
?>
</body>
</html>

```

どうでしょうか？利用側のコードが非常にすっきりした事がわかると思います。

今後他に様々な形式のデータを利用する場合やデータをDBから取得したりインターネットを通じてやり取りする場合、クライアント側のコードを変更する必要がなくなります。なお、Factory Methodパターンを適用したサンプルのクラス図は次のようになります。



Factory Methodパターンのオブジェクト指向的要素

Factory Methodパターンは「継承」を利用しているパターンです。

ProductクラスとConcreteProductクラス、CreatorクラスとConcreteCreatorクラス、それぞれの間で継承関係があります。親クラスであるProductクラスとCreatorクラスは、親クラスどうしでどういった連携を行うかを決めます。具体的には、「オブジェクト生成メソッドからProduct型のオブジェクトを返す」といったものです。その具体的な実装は、それぞれのサブクラスであるConcreteProductクラスとConcreteCreatorクラスに任せています。この

処理の大枠を親クラスで規定し、具体的な処理内容をサブクラスに任せる

という部分は、まさに**Template Methodパターン**となっています。

関連するパターン

Template Methodパターン

Factory Methodパターンは、Template Methodパターンの代表的な適用例です。通常、親クラスで処理の大枠を定義したメソッド(template methodと言います)が、factory methodになります。

Singletonパターン

Creatorクラスは、Singletonパターンとして作られることが多いです。これは、プログラム内で同じ工場が複数必要になることがほとんどないためです。

Abstract Factoryパターン

Abstract Factoryパターンもfactory methodを使って実装されることが多いパターンです。

まとめ

ここでは、Factory Methodパターンについて見てきました。

Factory Methodパターンを利用することで、**オブジェクトの生成処理と使用処理を分離し、利用者側から隠すことでクラス間の結びつきが低くなります**。これにより、再利用性が高まり柔軟な対応が可能となります。

ただ、Factory Methodパターンを意味もない箇所で適応しすぎてしまうと、余計なFactoryクラスが増えてしまい煩雑になることもあるため、設計段階から利用可否の判断をして実装する事をお勧めします。設計思想を後にメンテナンスする方にわかるよう、コメントを残すと良いでしょう。