Interpreter ～言語の文法表現を通訳する

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＋クラス

はじめに

ここではInterpreterパターンについて説明します。

「interpreter」とは「通訳者」「解釈者」という意味ですが、何を「通訳」「解釈」するのでしょうか？それは言語の文法表現です。

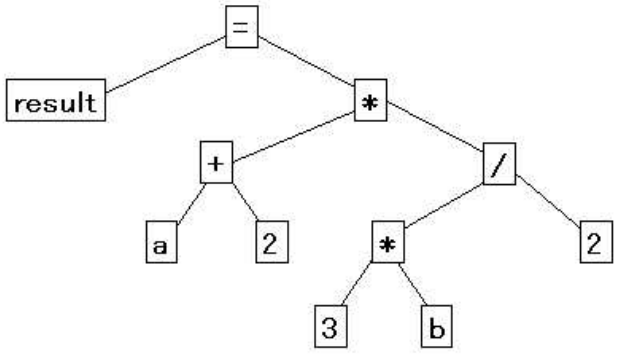
「言語」といっても「ある規則に従った文字列」と捉えると、HTMLやXMLなど文字通りの「言語」だけではなく、CSVなどのデータフォーマットも「言語」と言えるでしょう。

身近な例としてプログラミング言語を考えてみましょう。プログラミング言語には本書で取り上げているPHPを始め、PerlやRubyなどのコンパイルが不要な言語と、JavaやCなどのコンパイルが必要な言語があります。どのプログラミング言語を使っても最終的には記述したコードが実行されます。

ここで、コードが実行される手順を大まかに説明すると、次のようになります。

- 記述されたコードの文字を意味のある字句(トークン)に分解する(字句解析)
- 字句解析の結果を基に文法に従っているかどうかチェックする(構文解析)
- 中間コードや最終的なコードを生成
- コードを実行

1番目の字句解析と2番目の構文解析がおこなわれた結果は木構造として表すことができ、構文木(Abstract Syntax Tree)と呼ばれます。たとえば、「$result = $a / 2 + 3 * $b / 2」の構文木は次のように表せます。



Interpreterパターンは、この得られた構文木を処理するための最適なパターンです。

たとえば

ある決まった問題がたびたび発生する場合、その問題を文章として表せられると便利なことがあります。これは「ミニ言語」と呼ばれる手法です。

ミニ言語は比較的古くから使われており、PHPのsprintf関数で指定する「%5d」のような表記や正規表現などがあります。

Interpreterパターンは、このようなミニ言語を実装する場合に適用されるパターンです。

たとえば、バッチ処理を考えてみましょう。バッチ処理では、ある決まった基本的な処理を順番に実行したり繰り返し実行して、大きな処理をおこないます。通常、この基本的な処理はOS固有のコマンドですが、「バッチ処理言語」として実装することもできます。この場合、バッチ処理言語固有の規則で処理を記述することになりますが、複雑な処理を1つの命令として記述できたり、具体的なOSや言語に依存しないようになります。

Interpreterパターンとは？

Interpreter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。

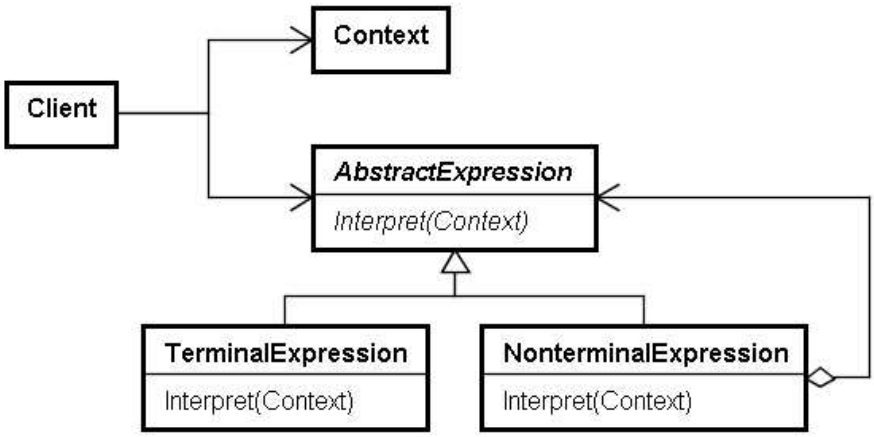
> 言語に対して、文法表現と、それを使用して文を解釈するインタプリタを一緒に定義する。

Interpreterパターンは、ある文法に従って記述されたものを解析した結果を利用するパターンと言えます。

Interpreterパターンでは文法で定義されている規則をクラスとして表し、それに対する振る舞いを併せて定義します。この規則とは、構文木における節や葉に相当します。そして、そのクラスのインスタンスを繋げることで構文木そのものを表現しつつ、構文木を処理します。

Interpreterパターンの構造

Interpreterパターンのクラス図と構成要素は、次のようになります。



AbstractExpressionクラス

構文木の要素に共通なAPIを定義します。

TerminalExpressionクラス

AbstractExpressionクラスのサブクラスで、構文木の葉に相当する末端の規則を表します。また、AbstractExpressionクラスで定義されたメソッドを実装します。

NonterminalExpressionクラス

AbstractExpressionクラスのサブクラスで、構文木の節に相当します。内部には、他の規則へのリンクを保持しています。また、AbstractExpressionクラスで定義されたメソッドを実装します。

Contextクラス

構文木を処理するために必要な情報を保持するクラスです。

Clientクラス

AbstractExpressionクラスを利用するクラスです。処理する構文木を作成したり、外部から与えられたりします。

Interpreterパターンのメリット

Interpreter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。

規則の追加や変更が簡単

Interpreterパターンの特徴の1つに、「1つの規則を1つのクラスで表す」というものが挙げられます。つまり、新しい規則を追加する場合はAbstractExpressionクラスのサブクラスを追加するだけで良くなります。

また、規則を修正する場合も、AbstractExpressionクラスのサブクラスを修正するだけです。

Interpreterパターンの適用例

Interpreterパターンの適用例として、簡単なミニ言語を作ってみましょう。

このミニ言語の文法は、次のように定義しました。

```
<Job> ::= begin <CommandList>
<CommandList> ::= <Command>* end
<Command> ::= diskspace | date | line
```

簡単に文法を説明しますと、この言語は「begin」という文字列で始まります。その間にコマンドの一覧を記述します。また、そのコマンド一覧は0個以上のコマンドで構成されており、「end」という文字列で終わります。コマンドは「diskspace」「date」「line」のいずれかになります。

この記述方法は、**BNF**（Backus Naur Form）と呼ばれる記法で、コンピュータ言語の文法を定義する際に使われます。また、**RFC**（Request for Comment：IETF（インターネットに関する技術標準を定める団体）が正式に発行する文書）でもよく使われています。

さて、この文法をInterpreterパターンを使って表すとどうなるでしょうか？

何となくお気づきの方もいらっしゃるかも知れませんね。そう、BNFで記述された文法の一番左にある「**<Job>**」「**<CommandList>**」「**<Command>**」の単位でクラスが作成されます。そして、そのクラスの中で文法に従っているかを判断し、適切に処理を実行するのです。

では、早速コードを見ていきましょう。

まずは、AbstractExpressionクラスに相当するCommandインターフェースからです。

Commandインターフェースでは、構文木に共通な**API**であるexecuteメソッドを定義しているだけです。

●Commandクラス（Command.class.php）

```
<?php
interface Command {
    public function execute(Context $context);
}
?>
```

次は、BNFにおける「<Job>」に相当するクラスのJobCommandクラスです。

●JobCommandクラス（JobCommand.class.php）

```
<?php
class JobCommand implements Command {
    public function execute(Context $context) {
        if ($context->getCurrentCommand() !== 'begin') {
            throw new RuntimeException('illegal command ' . $context->getCurrentCommand());
        }
        $command_list = new CommandListCommand();
        $command_list->execute($context->next());
    }
}
?>
```

executeメソッドに注目してください。このクラスではミニ言語の文法のうち、「**<Job>**」の部分だけを担当していることが分かりますか？

```
<Job> ::= begin <CommandList>
```

次は、「<CommandList>」に相当するCommandListCommandクラスです。

●CommandListCommandクラス（CommandListCommand.class.php）

```
<?php
class CommandListCommand implements Command {
    public function execute(Context $context) {
        while (true) {
            $current_command = $context->getCurrentCommand();
            if (is_null($current_command)) {
                throw new RuntimeException('"end" not found ');
            } else if ($current_command === 'end') {
                break;
            } else {
                $command = new CommandCommand();
                $command->execute($context);
            }
            $context->next();
        }
    }
}
?>
```

コマンドの一覧からコマンドを取り出して1つずつ実行し、「end」が現れるとそこで終了します。このクラスも「<CommandList>」に相当する処理だけをおこなっていますね。

```
<CommandList> ::= <Command>*
```

続けてCommandCommandクラスです。このクラスはTerminalExpressionクラス、つまり構文木の「葉」にあたるクラスです

●CommandCommandクラス（CommandCommand.class.php）

```
<?php
class CommandCommand implements Command {
    public function execute(Context $context) {
        $current_command = $context->getCurrentCommand();
        if ($current_command === 'diskspace') {
            $path = './';
            $free_size = disk_free_space('./');
            $max_size = disk_total_space('./');
            $ratio = $free_size / $max_size * 100;
            echo sprintf('Disk Free : %5.1dMB (%3d%%)<br>',
                         $free_size / 1024 / 1024,
                         $ratio);
        } else if ($current_command === 'date') {
            echo date('Y/m/d H:i:s') . '<br>';
        } else if ($current_command === 'line') {
            echo '----------------<br>';
        } else {
            throw new RuntimeException('invalid command [' . $current_command . ']');
        }
    }
}
?>
```

このクラスも「<Command>」に相当する処理だけをおこなっていることを確認してください。

```
<Command> ::= diskspace | date | line
```

さて、ここまで見てきたコードに必要なクラスで、まだ説明していないクラスがあります。構文木の情報を保持するためのContextクラスです。

このクラスは、現在構文木のどこを解析しているか、つまり、現在解析の対象となっているコマンドや次に出てくるコマンドを管理しています。

●Contextクラス(Context.class.php)

```
<?php
class Context {
    private $commands;
    private $current_index = 0;
    private $max_index = 0;
    public function __construct($command) {
        $this->commands = split(' +', trim($command));
        $this->max_index = count($this->commands);
    }

    public function next() {
        $this->current_index++;
        return $this;
    }

    public function getCurrentCommand() {
        if (!array_key_exists($this->current_index, $this->commands)) {
            return null;
        }
        return trim($this->commands[$this->current_index]);
    }
}
?>
```

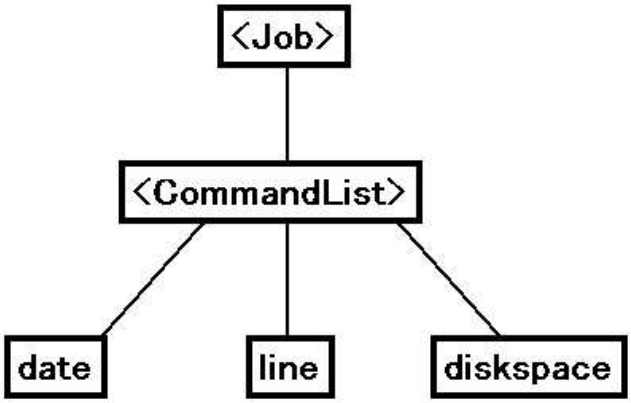
それでは最後のコードということで、クライアント側のコードを示します。クライアントは、入力フォームに入力されたコマンドの実行結果を表示するものになっていますので、色々な組み合わせのコマンドを確認できます。

●interpreter_clientクラス(interpreter_client.php)

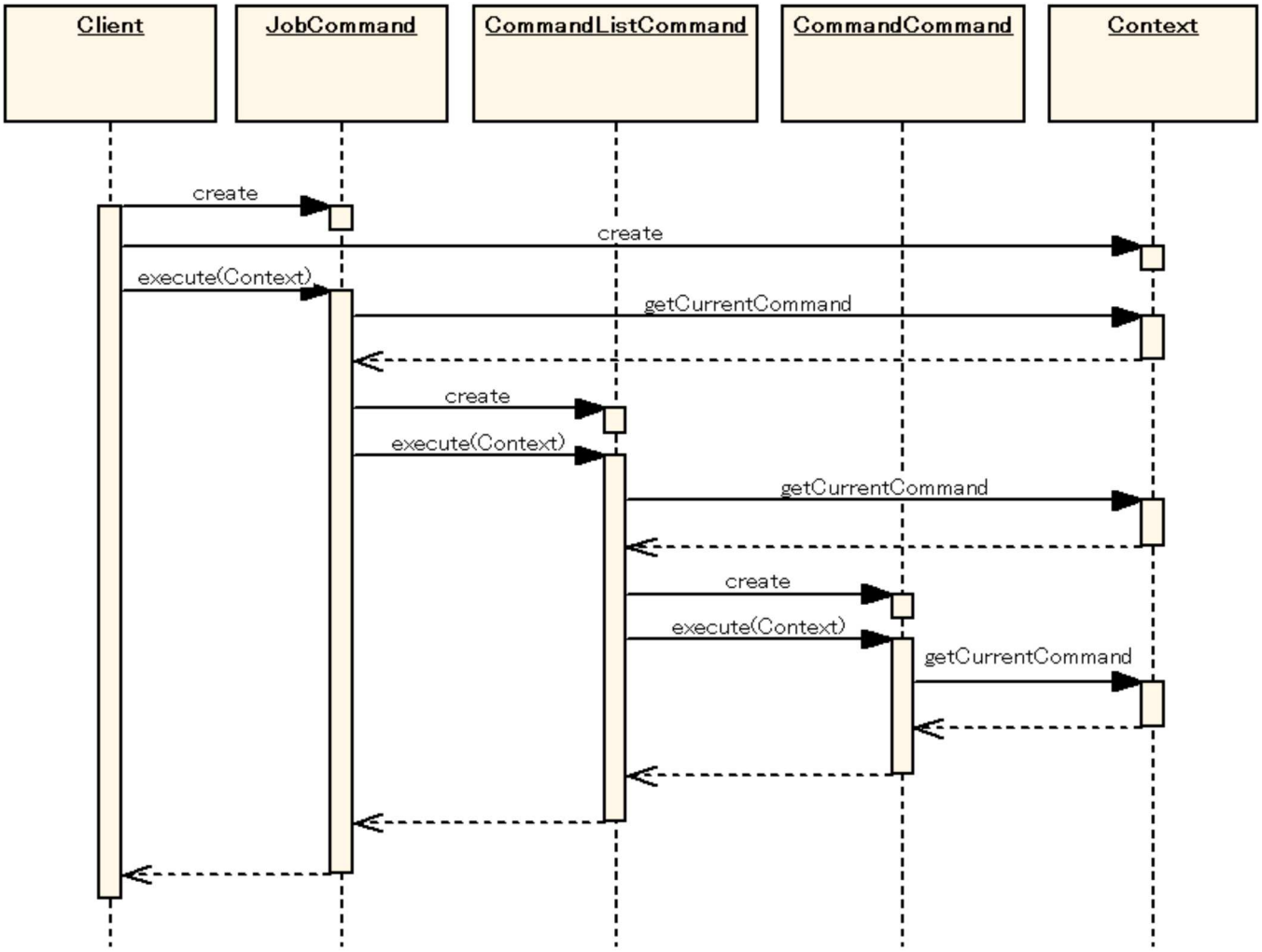
```
<?php
require_once 'Context.class.php';
require_once 'Command.class.php';
require_once 'CommandCommand.class.php';
require_once 'CommandListCommand.class.php';
require_once 'JobCommand.class.php';

function execute($command) {
    $job = new JobCommand();
    try {
        $job->execute(new Context($command));
    } catch (Exception $e) {
        echo htmlspecialchars($e->getMessage(), ENT_QUOTES);
    }
    echo '<hr>';
}
?>
<?php
    $command = (isset($_POST['command'])? $_POST['command'] : '');
    if ($command !== '') {
        execute($command);
    }
?>
<form action="" method="post">
input command:<input type="text" name="command" size="80" value="begin date line diskspace end">
<input type="submit">
</form>
```

なお、コマンドの初期値として「begin date line diskspace end」というコマンドを設定していますが、このコマンドの構文木は次のようになります。



ここで、どのように構文木が処理されていくのか、シーケンス図を使って示しておきます。NonerminalExpressionクラスに相当するJobCommandオブジェクトとCommandListCommandオブジェクトは次のコマンドを実行する役割を担いますが、TerminalExpressionクラスに相当するCommandCommandでは処理をおこなったあとに処理結果が戻されている様子が分かると思います。また、構文木の情報はすべてContextオブジェクトに問い合わせて受け取っていることも分かります。



最後にサンプルコードのクラス図を確認しておきましょう。

Interpreterパターン適用したサンプルのクラス図

Interpreterパターンのオブジェクト指向的要素

Interpreterパターンは「ポリモーフィズム」を非常に活用したパターンです。

ここまで見てきたように、AbstractExpressionクラスではそれぞれの規則に共通な**API**を提供しています。そして、AbstractExpressionクラスのサブクラスであるTerminalExpressionクラスやNonterminalExpressionクラスで、それぞれの規則の具体的な処理内容をこのこのAPIに実装しています。

また、NonterminalExpressionクラスは内部に他の規則、つまりTerminalExpressionオブジェクトもしくはNonterminalExpressionオブジェクトを保持しています。自身の処理によって、内部に保持したオブジェクトに次の処理を依頼します。

ここで、TerminalExpressionクラスもNonterminalExpressionクラスも同じAbstractExpression型のオブジェクトとして扱うことができます。ということは、内部に保持したオブジェクトに処理を依頼する際、そのオブジェクトが**TerminalExpression**オブジェクトなのか**NonterminalExpression**オブジェクトなのかを意識する必要がなくなります。

この「具体的な実装を意識する必要がない」ため、新しい規則を表すクラスを容易に追加できるのです。

関連するパターン

Compositeパターン

気づいた方もいるかと思いますが、構文木を形成するAbstractExpressionクラスとそのサブクラスの構造はCompositeパターンと非常に似ています。

Flyweightパターン

TerminalExpressionクラスにFlyweightパターンが適用される場合があります。

Visitorパターン

構文木の各要素に対する振る舞いを1つのクラスにまとめたい場合、Visitorパターンが利用できます。

まとめ

ここでは構文木を構成・処理するInterpreterパターンについて見てきました。

こう見ると、Interpreterパターンは他のGoFパターンと比べても用途が非常に具体的なパターンと言えますね。