▼第3章

デザインパターンを学ぶ

オブジェクト指向と頻出パターンの紹介

㈱クレイフィッシュ 小山 浩之 *OYAMA Hiroyuki oyama@crayfish.co.jp*

はじめに

昨今オブジェクト指向は，さまざまな言語のさまざまな分野で活発に研究されています．オブジェクト指向の概念はプログラミング言語を横断して共有できるため，研究結果を共通の資産として活用することができます．つまりオブジェクト指向機能を備えたPerlにもその研究結果，知識，ノウハウを流用することができるのです．

ところが，CGI用スクリプトとして流通するコード[注1]で参考にできるものが少ない上，手続き指向言語としてのノウハウの蓄積が多いためか，Perlのオブジェクト指向機能はあまり積極的に利用されていないのが現実です．オブジェクト指向がもたらすデータのカプセル化，ポリモーフィズム，継承などによるメリットを無視するのはあまりにもったいないことです．

今回はPerlのオブジェクト指向機能の解説からはじまり，Webアプリケーション開発中に発生する問題に対してデザインパターンを適用して解決してゆく方法を述べていきます．

オブジェクト指向の難しさ

オブジェクト指向の構文自体は単純で，誰でもすぐにコードを記述することができます．しかしオブジェクト指向による設計，それも変更に強く再利用可能な設計は正直難しいことです．

ではどのようにすれば巧い設計ができるようになるのでしょうか．

1つは経験を積むことです．もう1つは先人の設計，特に有用性が認められ幅広く利用されている設計を真似ることです．経験は時間をかけて積むとして，どのような手段でその巧い設計を探し出せば良いのでしょうか．その解はデザインパターンを紐解くことです．

デザインパターンとは

デザインパターンはオブジェクト指向のソフトウェアを設計する際の定石で，この定石を集めたパターンカタログは書籍やオンラインで参照することができます．デザインパターンはオブジェクト指向のソフトウェアで繰り返し利用される重要な設計に，体系的に名前をつけ，説明・評価し，その設計経験を第三者が効果的に利用できるように整理したもの，つまりお約束集・ネタ帳と言えます．

パターンカタログ

デザインパターンのパターンカタログには

- 名前：そのパターンの名称
- 目的：そのパターンの主な目的
- 問題：どのような問題に当てはめられるか
- 解法：そのようにその問題を解決するか
- 結果：その設計によってどのようなメリットとデメリットが生まれるか

注1）Perl普及の原動力の1つであったとも言えますが．

がまとめられています．ある程度複雑さをもったものを言い表すときに名前が決まっていると，意思疎通や記憶がスムーズにいきます．自分以外の人にどんな設計を採用しているか，どんな設計をすれば良いかを伝えるとき「ここは〜を使っているよ」「そこは〜にすれば良くない？」と一言で言えたら便利ですよね．

パターンの種類

デザインパターンはErich Gamma，Richard Helm，Ralph Johnson，John Vlissides ら4人による通称GoF（Gang Of Four）パターンが有名です．

パターンはこのほかにもさまざまな種類がまとめられています．ビジネスモデリングのパターンをまとめたアナリシスパターンなどマクロ的な視点のパターンもあれば，GoFパターンなどのようにミクロ的な視点のパターンもあります．

◇ ◇ ◇

今回はさまざまなパターンの中からオブジェクト指向プログラミングのスキルアップや効率化に有効と思われる，GoFパターンを主に扱います．

Perlのオブジェクト指向のおさらい

デザインパターンの話に移る前に，まずPerlのオブジェクト指向機能のおさらいをします．

オブジェクト

オブジェクトは，あるデータ構造とその処理方法を持っています．要は変数とそれを操作するための関数を集めたものとも言えます．オブジェクトを利用するためには以下のような手続きを踏みます．

①クラスを定義する
②クラスをインタンス化する
③インスタンスを操作する
④インスタンスを破棄する

それでは日付のデータを日付の文字列に変換するDateStringクラスを例にして，オブジェクトの利用手順と用語を確認していきます．

リスト1に挙げたDateStringクラスはDateString.pmというファイル名で保存します．DateStringクラスを利用するスクリプト（リスト2）を実行すると

```
Sun Oct 21 16:59:24 2001
```

といった現在の日付を表す文字列が出力されます．

■クラスを定義する

クラスの定義はpackageオペレータを使用します．packageを宣言してからコードの終端に達するか別のpackageを宣言する，もしくはスコープを外れるまでが1つのクラスを表します．この例では

```
package DateString;
```

でDateStringクラスの定義を始めることを宣言しています．

クラスを利用するためにはクラスのインスタンスを得る必要があり，そのインスタンスを生成するためのメソッドをコンストラクタと呼びます．

コンストラクタはクラスに結び付けたデータのリファレンスを返すメソッドで，DateStringクラスではサブルーチンnew( )がコンストラクタです．コンストラクタnewは，引数で受け取ったdelta秒を格納するハ

●リスト1　DateStringクラス

```
package DateString;
use strict;

sub new
{
    my $class = shift;
    my $time = shift || time();
    return bless { time => $time }, $class;
}

sub to_string
{
    my $self = shift;
    return scalar localtime $self->{time};
}
1;
__END__
```

●リスト2　DateStringクラスの使用

```
#!/usr/bin/perl
use DateString;

my $date = DateString->new;
print $date->to_string;
__END__
```

第3章 デザインパターンを学ぶ

オブジェクト指向と頻出パターンの紹介

ッシュを作成しそのリファレンスをクラスに結び付けて返します．引数が指定されない場合はデフォルトの動作として現在時刻を格納します．

これを少し冗長に記述すると

```
sub new
{
    my $class = shift;
    my $time = shift || time();
    my $date = { time => $time };
    return bless($data, $class);
}
```

といった記述になります．

また，Perlのコンストラクタには任意の名前をつけることができます．DateStringクラスの例ではnewという一般的な名称を使用していますが，instanceなど別の名前でもかまいません．

コンストラクタの第1引数には自動的にクラス名が渡されます．よってnewメソッドの変数$classはクラス名の文字列‘DateString’を受け取っています．コンストラクタに引数を指定する場合は

```
my $tomorrow = time() + 60 * 60 * 24;
my $date = DateString->new($tomorrow);
```

と指定します．コンストラクタの内部では第1引数にクラス名が渡され，指定した引数は第2引数以降に渡されます（図1）．

データとクラスの結び付けは，blessオペレータが行います．blessは第1引数にデータのリファレンス，第2引数にそのデータを結び付けるクラスの名前を指定します．blessされるデータはハッシュでも配列でもスカラでも型グロブでも，リファレンスであれば何でも構いません．

JavaやC++などデフォルトコンストラクタが用意される言語とは異なり，Perlは明示的にコンストラクタを定義する必要があります．これはオブジェクトが内包するデータに任意のデータ型を使用でき，これを指定するためです．

◇ ◇ ◇

クラスをインスタンス化しただけでは何も仕事ができませんので，コンストラクタに続いて日付のデータをctime形式の文字列に変換するto_stringメソッドを定義します．

```
sub to_string
{
    my $self = shift;
    return scalar localtime $self->{time};
}
```

メソッドの第1引数は自動的にそのオブジェクトのインスタンスが渡されます（図2）．この例では変数$selfにインスタンスを受け取っています．このインスタンスを通してコンストラクタで定義したデータ構造にアクセスできます．DateStringクラスで使用するデータはハッシュなので，ハッシュのリファレンスとしてアクセスします．

```
$self->{time};
```

仮にオブジェクトにデータをセットする場合は

```
my $yesterday = time() - 60 * 60 * 24;
$self->{time} = $yesterday;
```

図1／コンストラクタの呼び出し

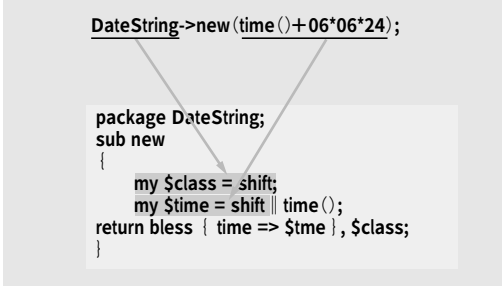
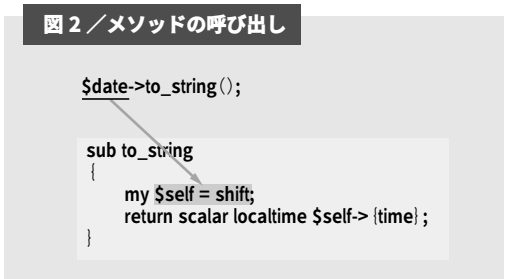


図2／メソッドの呼び出し



といった記述でデータをセットできます．

■クラスをインスタンス化する

これでクラスの定義が終わりました．続いてこのクラスを使用してみます．クラスをインスタンス化するためにはそのクラスのコンストラクタを呼び出します．

```
my $date = DateString->new;
```

この記述で先程定義したコンストラクタnewが実行され，DateStringクラスのインスタンス（オブジェクト）を返します．インスタンスは任意の変数型で保持することができ，この例ではスカラ変数$dateにインスタンスを受け取っています．

■インスタンスを操作する

生成したインスタンスを操作するためにメソッドを呼び出します．

```
print $date->to_string;
```

これによりDateStringクラスで定義したメソッドto_stringが実行され，変換された日付の文字列が返されるので，標準出力に

```
Sun Oct 21 16:59:24 2001
```

と出力します．

■インスタンスを破棄する

使用されなくなったインスタンスは自動的に破棄されるので，プログラマが明示的に行う必要はありません．Perlは誰からも参照されなくなった変数（インスタンス）をガーベッジコレクションの対象として破棄します．つまりレキシカル変数はスコープを外れるなど，他のオブジェクトから参照されなくなった時点でガーベッジコレクションの対象になります．

また，インスタンスが消滅するタイミングで呼び出される特別なメソッド（デストラクタ）を定義することもできます．DESTROYという名前で定義したメソッドがデストラクタになり，インスタンスが消滅する直前に呼び出されます．

継承

継承とは，既存のクラスを元にして，機能を追加した新しいクラスを少ない手数で定義できる方法です．また，継承は重複したコードを抜き出して1つにまとめる手段としても利用されます．

継承したクラスは継承する元のクラス（スーパクラス）が実装しているメソッドを引き継ぎます．その引き継いだメソッドをオーバライド（上書き）して動作をカスタマイズしたり新しくメソッドを追加したりといった，差分だけをコーディングすることが可能になります．これにより既存のコードの編集や新たにコピー&ペーストするような，リスクを伴う変更を行わずに済みます．

◇ ◇ ◇

では先ほどのDateStringクラスを継承して，HTTPで利用されるRFC1123形式の日付の文字列を返す

●リスト3 DateStringRfc1123クラス

```
package DateStringRfc1123;
use base 'DateString';

use constant RFC1123_FORMAT => '%s, %02d %s %04d %02d:%02d:%02d GMT';

sub to_string
{
    my $self = shift;
    my @gmt = gmtime $self->{time};
    return sprintf RFC1123_FORMAT,
        (qw(Sun Mon Tue Wed Thu Fri Sat))[$gmt[6]],
        $gmt[3],
        (qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))[$gmt[4]],
        $gmt[5] + 1900,
        $gmt[2], $gmt[1], $gmt[0];
}
1;
__END__
```

第3章 デザインパターンを学ぶ

オブジェクト指向と頻出パターンの紹介

DateStringRfc1123クラスを作成します．リスト3を見てください．

Perlでクラスを継承する場合はbaseプラグマを使用します．DateStringRfc1123クラスはDateStringクラスを継承するために以下のように宣言します．

```
package DateStringRfc1123;
use base 'DateString';
```

これによりDateStringRfc1123はDateStringクラスを継承したクラスとして宣言されます．version 5.004などの若干古いバージョンのperlを使用する場合は，baseプラグマを使わずに配列@ISAに継承するクラス名をセットして指定します．

```
package DateStringRfc1123;
@ISA = qw(DateString);
```

この方法は最新のversion 5.6.1などでも使用することができますが，見通しの良さからbaseプラグマを使用する方法をお勧めします．

仮に複数のクラスから継承（多重継承）する場合は，baseプラグマに配列でスーパクラスの名前を指定します．

```
use base qw(DateString JapaneseString);
```

配列@ISAで指定する場合も同様です．

DateStringクラスを継承したDateStringRfc1123クラスでは，to_stringメソッドだけが実装されていますが，DateStringから受け継いだコンストラクタnewを同様に使用することができます．

リスト4のコードを実行するとRFC1123の形式で現在の日付の文字列を得ることができます．

```
Sun, 21 Oct 2001 16:59:24 GMT
```

DateStringクラスで定義したto_stringメソッドではなく，DateStringRfc1123クラスで定義したto_stringメソッドが呼び出されていることがわかります．このようにスーパクラスで定義されているメソッドを上書きしてカスタマイズすることをオーバライドと呼びます．

同様にDateStringクラスを継承してRFC850の形式の日付文字列を出力するDateStringRfc850クラスを定義しましょう（リスト5）．このクラスで定義したto_stringメソッドは

```
Sunday, 21-Oct-01 16:59:24 GMT
```

という形式で日付の文字列を出力します．

●リスト4　DateStringRfc1123クラスの使用

```
#!/usr/bin/perl
use DateStringRfc1123;

my $date = DateStringRfc1123->new;
print $date->to_string;
__END__
```

動的束縛とポリモフィズム

オブジェクト指向プログラミングで得られる最も大きなメリットの1つとして，動的束縛とポリモフィズ

●リスト5　DateStringRfc850クラス

```
package DateStringRfc850;
use base 'DateString';

use constant RFC850_FORMAT => '%s, %02d-%s-%02d %02d:%02d:%02d GMT';

sub to_string
{
    my $self = shift;
    my @gmt = gmtime $self->{time};
    return sprintf RFC850_FORMAT,
        (qw(Sunday Monday tuesday Wednesday Thursday Friday Saturday))[$gmt[6]],
        $gmt[3],
        (qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec))[$gmt[4]],
        $gmt[5],
        $gmt[2], $gmt[1], $gmt[0];
}
1;
__END__
```

ムを利用できる点が挙げられます．例としてリスト6を見てください．

このコードはDateString，DateStringRfc1123，DateStringRfc850それぞれのインスタンスを配列@dateに格納した後，配列@dateからインスタンスを取り出して，無造作にto_stringメソッドを呼び出します．これを実行すると次のような出力が得られます．

```
Sun Oct 21 16:59:24 2001
Sun, 21 Oct 2001 16:59:24 GMT
Sunday, 21-Oct-01 16:59:24 GMT
```

もしこのような処理をオブジェクト指向ではない別の記述の仕方をしたならばどうなるでしょう．おそらく日付の形式ごとに別々の名前で関数を用意し，どの形式で出力すべきか調べてから適切な関数を個別に呼び出すようなコードになるでしょう．

それをこのコードではインスタンス化した後は，ただ

```
$date->to_string;
```

とメソッドを呼び出すだけで済んでいます．条件フラグも，突然表れる謎の識別番号も必要ありません．サブルーチンと処理が固定された関係にあるのとは異なり，この例のようにメソッドに対応した処理が実行時に決定されることを動的束縛と呼びます．

また例の中のスカラ変数$dataはDateStringクラスのインスタンスであるときもあれば，DateStringRfc1123クラスやDateStrinRfc850クラスのインスタンスであるときもあります．このように同じメソッドを受け取ることができるオブジェクト同士を，置き換えて処理できるような特性をポリモフィズムと呼びます．

さらにPerlで動的束縛が行われる場面はインスタンスとメソッドの関係に留まらず，クラスとコンストラクタやメソッド自身に対しても行われます．リスト7は先ほどあげたコードと同じ結果を出力します．つまりPerlはクラス名を元にしてそのクラスを直接インスタンス化できるのです．たとえば以下のような記述でもインスタンスを生成できます．

```
"DateString"->new;
```

また，メソッド名を文字列で指定することで呼び出すメソッドを動的に指定することができます．

```
my $method = 'to_string';
$date->$method();
```

静的なプログラミング言語を使っていて感じる「こんな風に書けたら良いのになぁ」をPerlでは簡素に直接記述することができます．

その他の情報源

駆け足で説明を進めてしまいましたが，Perlのオブジェクト指向機能は本稿末の参考文献で挙げた書籍などで詳細に解説されています．そこでもし不明な点があったならば，Perlのオブジェクト指向機能についてのメーリングリスト“Object-oriented with Perl”を覗いて，主旨をご理解いただいた上で参加してみてください．何かお役にたてるかもしれませんので．

●リスト6　ポリモフィズムの例

```
#!/usr/bin/perl
use DateString;
use DateStringRfc1123;
use DateStringRfc850;

my @date;
push @date, DateString->new;
push @date, DateStringRfc1123->new;
push @date, DateStringRfc850->new;

foreach  my $date (@date) {
    print $date->to_string, "\n";
}
__END__
```

●リスト7　クラスを直接インスタンス化する例

```
#!/usr/bin/perl
use DateString;
use DateStringRfc1123;
use DateStringRfc850;

for my $class (qw(DateString DateStringRfc1123 DateStringRfc850)){
    my $date = $class->new;
    print $date->to_string, "\n";
}
__END__
```

第3章 デザインパターンを学ぶ

オブジェクト指向と頻出パターンの紹介

デザインパターンの実例

Webアプリケーションで行われる処理は比較的定型的な処理が多く，発生するコーディング上の問題や解決すべき技術的なポイントも似通っています．ここでは一般的なWebアプリケーションの開発中に表れるコーディング上の問題を題材とし，デザインパターンを参考にしてそれらを解決していってみましょう．

Template Methodパターン

掲示板の参照と書き込みを行う場合を考えてみます．Webアプリケーションのほとんどは一定処理手順を踏んで実行されるのですが，要求の種類を判定するための条件分岐が処理フェーズごとに表れてしまうコードをよく見かけます（リスト8）．

このようにコードに条件分岐がちりばめられていると理解の妨げになるばかりか，コードに変更を加えるたびに各々の条件分岐を意識しながら修正する必要が生まれるため，修正に要するコストが高くなってしまいます．

ヒントを得るためにデザインパターンのパターンカタログを見ると，目的に「1つのオペレーションにアルゴリズムのスケルトンを定義しておき，その中のいくつかのステップについては，サブクラスの定義に任せることにする」を掲げるTemplate Methodパターンが参考になりそうです．

ここで言うアルゴリズムは

- 要求の解析
- 必要な処理
- コンテンツの生成
- コンテンツの出力

などWebアプリケーションで行われる一連の流れを指します．まずこの流れを定義するApplicationHandler

●リスト8　Template Method適用前

```
#!/usr/bin/perl
use strict;
my @message = qw(
    MESSAGE1
    MESSAGE2
    MESSAGE3
);

my $mode;
if ($ENV{QUERY_STRINGS} =~ m{^mode=write}) {
    $mode = 'write';
}
else {
    $mode = 'show';
}

if ($mode eq 'write') {
    if ($ENV{QUERY_STRINGS} =~ m{^mode=write?message=(.*)$}) {
        push @message, $1;
    }
}

my $content;
if ($mode eq 'write') {
    $content = join "\n", 'In Write Mode', @message;
} else {
    $content = join "\n", 'In Show Mode', @message;
}

print 'Content-Type: text/html', "\n\n";
print $content;
__END__
```

●リスト9　ApplicationHandlerクラス

```
package ApplicationHandler;
use strict;

sub new
{
    my $class = shift;
    my $query = shift || $ENV{QUERY_STRINGS};
    bless {
        content => '',
        query   => $query,
        message => [qw(MessageA MessageB MessageC)],
    }, $class;
}

sub compute
{
    my $self = shift;
    $self->do_task;
    $self->do_create_content;
    $self->do_return_content;
}

sub do_task {}

sub do_create_content
{
    my $self = shift;
    my $messages = $self->{message};
    $self->{content} = join "\n", @$messages;
}

sub do_return_content
{
    my $self = shift;
    print "Content-Type: text/plain\n";
    print $self->{content};
}
1;
__END__
```

クラス（リスト9）を作成します．

ApplicationHandlerクラスではフェーズごとに割り当てられたメソッドdo_task，do_create_content，do_return_contentが定義されています．これらのメソッドはcomputeメソッドから逐次呼び出されます（図3）．

ApplicationHandlerクラスを使用するコード，リスト10を実行すると

```
Content-Type: text/plain
Hello Object World
```

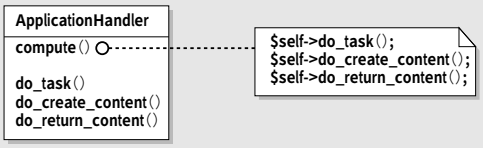
という出力が得られます．

続いてメッセージを参照する場合の処理を定義するShowHandlerクラス（リスト11）を作成します．また同様にメッセージを書き込む場合の処理を定義するWriteHandlerクラス（リスト12）を作成します．

ShowHandlerとWriteHandlerクラスは，ApplicationHandlerクラスを継承します．ApplicationHandlerクラスで定義したcomputeメソッドはこれらのクラスにも受け継がれるため，リスト13のように使用します．

このようにそれぞれの処理フェーズに散らばっていた条件分岐は，生成するインスタンスを決定する部分だけに集約され，見通しの良いコードになりました．Template Methodパターンはオブジェクト指向プログラミングの中でも基礎の基礎で，あらゆるところでその実装を見ることができます．

図3／ApplicationHandlerクラスのメソッド呼び出し



●リスト10　ApplicationHandlerクラスの使用

```
#!/usr/bin/perl

use ApplicationHandler;
use strict;

my $application = ApplicationHandler->new;
$application->compute;
__END__
```

●リスト13　ShowHandlerとWriteHandlerの使用

```
#!/usr/bin/perl

use ShowHandler;
use WriteHandler;
use strict;

my $application;
if  ($ENV{QUERY_STRINGS} =~ /mode=write/) {
    $application = WriteHandler->new;
} else {
    $application = ShowHandler->new;
}
$application->compute;
__END__
```

Factory Methodパターン

先ほど挙げたコードの冒頭に，書き込みモードか読み込みモードかを判断して必要なクラスのインスタンスを生成する処理があります．要求を解釈して必要なクラスのインスタンスを生成して返すという単純な処理ですが，この判断のための知識をカプセル化するこ

●リスト11　ShowHandlerクラス

```
package ShowHandler;
use base 'ApplicationHandler';
use strict;

sub do_create_content
{
    my $self = shift;
    my $messages = $self->{message};
    $self->{content} = join "\n", 'In Show Mode',  @$messages;
}
1;
__END__
```

●リスト12　WriteHandlerクラス

```
package WriteHandler;
use base 'ApplicationHandler';
use strict;

sub do_task
{
    my $self = shift;
    my $query_strings = $self->{query};
    my $messages = $self->{message};

    if ($query =~ /new_message=(.*)$/) {
        push @$message, $1;
    }
}

sub do_create_content
{
    my $self = shift;
    my $messages = $self->{message};
    $self->{content} = join "\n", 'In Write Mode', @$messages;
}
1;
__END__
```

第3章 デザインパターンを学ぶ

オブジェクト指向と頻出パターンの紹介

とで，さらに見通しが良くなるはずです．仮にその判断の処理が複雑だったとしても，カプセル化してあれば利用する側は単にその機能を呼び出す手続きを知っているだけでよいのです．

そのための実装はリスト14のようになり，このApplicationFactoryクラスはリスト15のようなコードで使用します．

結果このコードは，

- 生成するクラスつまり実行する機能を決定するApplicationFactoryクラス
- 書き込みモードの動作を決定するWriteHandlerクラス
- 読み出しモードの動作を決定するShowHandlerクラス

といった風に明確に役割が分担され，条件分岐が少ないため理解しやすい構成になっています．

このようにインスタンスを生成するための手続きを提供し，インスタンス化するクラスを決定するための知識をカプセル化する構成をFactory Methodパターンと呼びます．単純にコンストラクタを使用してインスタンスを生成する場合と比べ，Factory Methodパターンを利用した場合は，実際に生成するインスタンスを変化させることができるために柔軟性が高まります．

身近な例では，RDBMSのAPIであるDBIのconnect()メソッドがFactory Methodパターンの実例と言えます．

ちなみにPerlはクラスとコンストラクタの関係にも動的束縛を使用することができるので，Factory Methodパターンを利用することなく生成するインスタンスを変化させることができます．このクラスとコンストラクタの動的束縛を要する場合，インスタンス化するクラスを文字列で指定します．

```
my $class_name = 'ShowHandler';
my $application = $class_name->new;
```

この記述で，設定ファイルによってインスタンス化するクラスを指定するなどの高いカスタマイズ性を得ることができます．ただよほど単純な場面を除き，素直にFactory Methodパターンを使用したほうが好ましいでしょう．

●リスト14　ApplicationFactoryクラス

```
package ApplicationFactory;
use WriteHandler;
use ShowHandler;
use strict;

sub create
{
    if  ($ENV{QUERY_STRINGS} =~ /mode=write/) {
        return WriteHandler->new;
    } else {
        return ShowHandler->new;
    }
}
1;
__END__
```

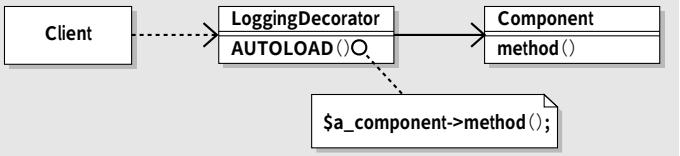
●リスト15　ApplicationFactoryクラスの使用

```
#!/usr/bin/perl

use ApplicationFactory;
use strict;

my $application = ApplicationFactory->create;
$application->compute;
__END__
```

●リスト16　LoggingDecoratorクラス

```
package LoggingDecorator;
use strict;
our $AUTOLOAD;

sub new
{
    my $class = shift;
    my $component = shift;
    bless { component => $component }, $class;
}

sub get_component
{
    my $self = shift;
    $self->{component};
}

sub AUTOLOAD
{
    my $self = shift;
    my @args = @_;
    my $method = $AUTOLOAD;
    $method =~ s/.+:://;
    return if $method eq 'DESTROY';

    printf STDERR "%s - %s->(%s)\n",
        scalar localtime,
        ref$self->get_component,$method,join(', ', @args);
    $self->get_component->$method(@_);
}
1;
__END__
```

図 4 ／要求の転送



Decoratorパターン

アプリケーションの挙動を追跡するために詳細なトレースログを出力していると，デバッグの労力を大幅に軽減できる場合があります．しかしログを出力するためのコードを各クラスに定義していると，本来の目的ではない処理が混入して簡潔性が失われてしまいます．また継承しオーバライドしてログを出力するよう拡張すると，それぞれのクラスごとにログ出力用のサブクラスが必要になってしまい，クラスの数が膨れ上がる恐れがあります．

デザインパターンのパターンカタログを見ると，目的に「オブジェクトに責任を動的に追加する．サブクラス化よりも柔軟な機能拡張方法を提供する」と記されたDecoratorパターンが参考にできそうです．

Decoratorパターンはオブジェクトを包み込み装飾（デコレーション）することで，そのオブジェクトに新しい機能を追加します．さらに包み込んだオブジェクトに対して要求を転送するので，そのオブジェクトと同じように振る舞うことができます．これらの拡張を継承などの静的な手段ではなく，実行時に動的に機能を拡張できることがポイントです．

ここでは，リスト16のように任意のオブジェクトにトレースログ出力機能をデコレーションするLoggingDecoratorクラスを定義します．

LoggingDecoratorクラスは，任意のオブジェクトにSTDERRへのトレースログ出力機能を追加します．あるクラスで定義されていないメソッドを呼び出されたとき，PerlはAUTOLOADメソッドを呼び出します．呼び出そうとしていたメソッドは特種変数$AUTOLOADを参照することで得ることができ，それを元に本来その要求を受け取るべきオブジェクトに要求を転送することができます（図4）．この方法はオブジェクトを自動転送する手段として一般的なテクニックで，これらの機能をまとめたDelegateモジュールなどがCPANに登録されています．

作成したLoggingDecoratorクラスの使用方法を見てみます．LoggingDecoratorクラスは，コンストラクタの引数にトレースログ出力機能を追加したいオブジェクトを指定します（リスト17）．

生成したインスタンス$logging_dbhはLoggingDecoratorクラスのインスタンスですが，受け取った要求をすべてDBIのオブジェクトに転送するため，普通のDBIのデータベースハンドラと同じように振る舞います．そのためDBIのデータベースハンドラがもつ機能をそのままに，すべての操作に対してメソッド名・引数・タイムスタンプをSTDERRに出力する拡張が加えられています．

```
Mon Nov 19 17:38:03 2001 - DBI::db->prepare(
                DELETE FROM users WHERE age < ?
        )
Mon Nov 19 17:38:04 2001 - DBI::db->commit()
Mon Nov 19 17:38:04 2001 - DBI::db->disconnect()
```

この例ではDBIのオブジェクトにロギング機能を追

●リスト17　LoggingDecoratorクラスの使用

```
#!/usr/bin/perl

use LoggingDecorator;
use DBI;
use strict;

my $dbh = DBI->connect('dbi:driver:dsn', '', '', {
                    RaiseError => 1, AutoCommit => 0
                     });
my $logging_dbh = LoggingDecorator->new($dbh);
eval {
    my $state = $logging_dbh->prepare(q{
        DELETE FROM users WHERE age < ?
    });
    $state->execute(18);
    $logging_dbh->commit;
};
$logging_dbh->rollback if $@;
$logging_dbh->disconnect;
__END__
```

第3章 デザインパターンを学ぶ

オブジェクト指向と頻出パターンの紹介

加しましたが，他にもCGIモジュールやIO::Socketモジュールなど，任意のオブジェクトにロギング機能を追加することができます．またget_componentメソッドで元のオブジェクトを取り出すこともできるので，任意のタイミングでこの機能を取り外すことも可能です．

■ホワイトボックスな再利用

継承は，継承するクラスの内部構造に関する知識をある程度要求するため，ホワイトボックスな再利用と呼ばれます．つまり継承関係を持つことはそのクラスどうしが非常に強い結びつきを持つことであり，ベースクラスの変更がサブクラスへ直接影響する危険性があります．これはベースクラスの一部を変更すればすべてのサブクラスに対して反映されるメリットの裏返しのデメリットと言えます．

■ブラックボックスな再利用

逆にDecoratorパターンなどのように委譲によってオブジェクトの機能を拡張・変更してゆく方法は，ブラックボックスな再利用と呼ばれます．ブラックボックスな再利用はあるオブジェクトの振る舞いを拡張したい場合でも，そのオブジェクトの内部構造を知る必要がないのです．また継承とは異なり，オブジェクトの振る舞いや責任を実行時に決定・変更できる強力な柔軟性も見のがすことはできません．

●リスト18　SendmailCommandクラス

```
package SendmailCommand;
use base 'Command';
use strict;

use constant MAIL_TEMPLATE => <<__MAIL_TEMPLATE__;
From: <%>
To: <%s>
Subject: %s

%s
__MAIL_TEMPLATE__

sub to
{
    my $self = shift;
    $self->{to} = shift if @_;
    $self->{to};
}

sub execute
{
    my $self = shift;
    open MAIL, '| /usr/lib/sendmail -f info@example.jp';
    printf MAIL MAIL_TEMPLATE,
        'info@example.jp', $self->to, 'Demo Message', 'Hello World';
    close MAIL;
}
1;
__END__
```

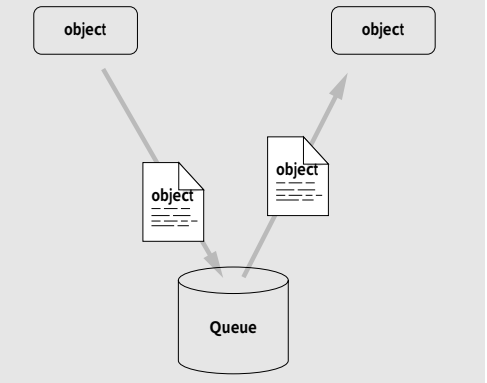
Commandパターン

Webアプリケーションは，いかに資源の占有時間を短くするか，試行錯誤の連続と言えます．特にフロントエンドのApacheは利用できるスロットの数が限られているため，長時間スロットを占有するようなアプリケーションでは新たな要求をさばくためのスロットが確保できず，全体的な待ち時間が増大してしまいます．

この状況を回避する手段として処理時間を短縮することが基本として挙げられますが，これにも限界があります．そのため，大量のトラフィックをさばくWebアプリケーションでは，処理の要求を一時的に保存して別のタイミングで処理を実行する非同期処理を行う場合もあります．非同期処理を実現するための方法はいくつか存在しますが，ここでは手軽なキューを利用した非同期処理を実装します．

デザインパターンのパターンカタログを眺めると，目的に「要求をオブジェクトとしてカプセル化することによって，異なる要求や，要求からなるキューやログによりクライアントをパラメータ化する．また，取り消し可能なオペレーションをサポートする」を掲げるCommandパターンがこの例に適切であることがわかります．Commandパターンを参考にしたキューの実装QueueクラスとCommandクラスをhttp://perl.

図5／Queueクラスの動作



infoware.ne.jp/source/Queue-0.1.tar.gzからダウンロードできるので今回はこれを題材に説明します．

Queueクラスはファイルを使用したキューの実装で，任意のオブジェクトをキューに蓄積することができます．QueueクラスはStorableモジュールを使用してオブジェクトを文字列に変換してファイルに格納し，取り出すときはファイルの文字列を評価してオブジェクトを復元します（図5）．

キューに投入する処理の定義は，Commandクラスのサブクラスで定義します．メール送信処理を行うSendmailCommandクラス（リスト18）を定義してみます．

SendmailCommandクラスは，toメソッドで送信先をセットした後にexecuteメソッドを実行するとメールの送信を始めます．単体で利用することもできますが，キューを使用する場合は送信先を設定した状態でキューに投入します（リスト19）．

Webアプリケーション側ではこの操作で終了し，次の要求に備えます．今度はこの要求をキューから取り出して実行する別のプロセスを定義します（リスト20）．

このコードを起動するとキューからすべてのCommandオブジェクト順番に取り出しexecuteメソッドを呼び出して逐一実行します．Perlのref関数で取得したCommandオブジェクトのクラス名を元にクラスを動的にロードしています．

この例では，キューに投入され$commandへ取り出されるオブジェクトはSendmailCommandクラスのインスタンスだけですが，CreateNewUserAccountCommandクラスなど他のオブジェクトがキューに投入されていたとしても，executeメソッドを実装したオブジェクトであれば問題なく実行することができます．ここでもポリモフィズムを有効活用することで条件分岐が不要になることがわかります．

この例では投入済みのCommandオブジェクトをすべてキューから取り出して実行していますが，サーバの負荷に応じて一定量以上同時に処理しないようにすることも可能です．さらにネットワークを経て共有できるキューを利用した場合は，Commandオブジェクトの実行を複数のマシンで分散処理することも可能です．要求をするプロセスと要求を処理するプロセスを分断することで，応答性の確保ばかりでなく処理能力のスケーラビリティも同時に得ることができるのです．

●リスト19　SendmailCommandクラスの使用（クライアント側）

```
#!/usr/bin/perl
use Queue;
use SendmailTask;
use strict;

my $command = SendmailTask->new;
$command->to('oyama@crayfish.co.jp');

my $queue = Queue->new('/path/to/directory');
$queue->add($command);
__END__
```

Singletonパターン

設定ファイルを参照することで，アプリケーションが出力するメッセージや使用するクラスを変更することが可能になり，柔軟なソフトウェアができあがります．ところが設定ファイルを参照するために複数のクラスで個別に設定ファイルの読み込み処理を実装したり，個々に設定ファイルを読み込むインスタンスを生成していたりしては無駄が多くなります．もちろんグローバル変数にそのインスタンスを保持して使い回す方法も採れますが，横断的に利用されるグローバル変数はコードの理解の妨げになり，保守上の障害になる可能性も高いため，できるだけ利用は避けたいものです．

パターンカタログの目的に「あるクラスに対してインスタンスが1つしか存在しないことを保証し，それにアクセスするためのグローバルな方法を提供する」を掲げるSingletonパターンが参考になりそうなので，

●リスト20　SendmailCommandクラスの使用（サーバ側）

```
#!/usr/bin/perl
use Queue;
use strict;

my $queue = Queue->new('/path/to/directory');
while (my $command = $queue->get) {
    eval {
        my $class = ref $command;
        eval "require $class";
        $command->execute;
    };
    print STDERR $@ if $@;
}
__END__
```

第3章 デザインパターンを学ぶ

オブジェクト指向と頻出パターンの紹介

●リスト21 SingletonConfクラス

```
package SingletonConf;
use base 'Conf';
use strict;

our $Instance;
sub instance
{
    my $class = shift;
    return $Instance if defined $Instance;

    $Instance = $class->new(@_);
    return $Instance;
}

1;
__END__
```

このパターンを利用して無駄を省いてみましょう．

Singletonパターンを実装する方法はいくつかありますが，今回は最も手軽なクラス変数を利用した実装を行います．すでに設定ファイルを読み込むConfクラスが存在するとします．そのクラスを基にインスタンスを1つだけ生成する拡張をします（リスト21）．

作成したSingletonConfクラスには，新しくinstanceメソッドを追加しました．instanceメソッドは，SingletonConfクラスのインスタンスがすでに生成されているときはその生成済みのインスタンスを返し，生成されていないときは新しくインスタンスを生成して返します．

これによって設定ファイルの情報を参照したい場所ではリスト22のように，決まった文法でインスタンス化して参照することができるようになります．Singletonパターンを利用することで生成するConfクラスのインスタンスを1つだけに抑えることができたうえ，グローバル変数に頼る必要もなくなります．

定石という宝石

デザインパターンは，ソフトウェア設計の定石を集めたものだと冒頭で述べました．ソフトウェアの世界も，たとえば囲碁の世界と同様で，定石を外した設計は素人相手には通用するかもしれませんが，プロには通用しません．

CPANなどのライブラリを活用することと同様に，自分の取り組んでいる分野がすでにパターン化されていないか，まず調べてみてください．もしあなたがオブジェクト指向システムに慣れ親しんだ人であれば，パターンによって設計を洗練させ知識を整理し，人に伝えることが容易になるはずです．もしあなたがオブジェクト指向システムに触れて間もない人ならば，パターンから良い設計や定石を学びとることができるはずです．

●リスト22 SingletonConfクラスの使用

```
#!/usr/bin/perl
use strict;
use SingletonConf;

my $conf = SingletonConf->instance;
my $class_name = $conf->get('DEFAULT_LOGGING_CLASS');
my $log = $class_name->new;
__END__
```

今回紹介したのはパターンのごく一部で，この他にも有用なパターンがたくさん公開されています．ソフトウェアの開発に携わる人はぜひパターンに触れて，見つけて，そして共有してみてください．それはあなたとあなたたちのソフトウェア開発経験をさらに豊かにしてくれるはずです．

参考文献

- **『オブジェクト指向における再利用のためのデザインパターン―改訂版』**
ISBN4-7973-1112-6，ソフトバンクパブリッシング，Erich Gamma/Richard Helm/Ralph Johnson/John Vlissides著，本位田真一／吉田和樹 監訳
- **『リファクタリング』**
ISBN4-89471-228-8，ピアソン・エデュケーション，Martin Fowler著，児玉公信／友野晶夫／平澤章／梅澤真史訳
- **『実用Perlプログラミング』**
ISBN4-900900-82-6，オライリージャパン，Sriram Srinivasan著，須田隆久訳
- **「Tom's object-oriented tutorial for perl」**
http://www.perldoc.com/perl5.6.1/pod/perltoot.html，Tom Christiansen著
- **「Object-oriented with Perlメーリングリスト」**
http://perl.infoware.ne.jp/
- **perldoc**
% perldoc perl