


## Adapterパターン～APIを変更する

 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

構造+クラス、オブジェクト

### はじめに

ここではAdapterパターンについて説明します。

adaptという単語には「適合させる」とか「なじませる」「適応させる」という意味があります。つまり、adapterとは「適合させるもの」という意味になります。

日本語でも、「アダプタ」という言葉はあちこちで聞く言葉だと思います。ACアダプタやネットワークアダプタ、一昔前に全盛を誇ったISDN回線で必要になるターミナルアダプタ(TA)などいろいろあります。

デザインパターンの一つ、**Adapterパターン**は、APIの異なるクラスどうしを「適合させる」パターンです。

### たとえば

ここでは、すでに作成済みで過去に充分テストされており、多くの利用実績のあるクラスを再利用する場面を考えてみましょう。

このクラスは、次のようなクラスです。

●ShowFile.class.php

```
<?php
/**
 * 指定されたファイルの内容を表示するクラスです
 */
class ShowFile
{
    /**
     * 内容を表示するファイル名
     *
     * @access private
     */
    private $filename;

    /**
     * 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;
    }

    /**
     * プレーンテキストとして表示します
     */
    public function showPlain()
    {
        echo '<pre>';
        echo htmlspecialchars(file_get_contents($this->filename), ENT_QUOTES);
        echo '</pre>';
    }
}

/**
```

```

    * キーワードをハイライトして表示します
    */
    public function showHighlight()
    {
        highlight_file($this->filename);
    }
}
?>

```

このShowFileクラスには、コンストラクタと2つのメソッドshowPlainメソッドとshowHighlightメソッドが定義されています。コンストラクタは、引数としてファイル名を1つ受け取ります。showPlainメソッドは、コンストラクタ引数として渡されたファイル名のファイル内容をそのまま表示します。一方のshowHighlightメソッドは、showPlainメソッドと同じく、ファイルの内容を表示しますが、キーワードをハイライト表示します。

このクラスを直接利用するクライアント側コードは、次のようになります。

●adapter\_sample\_client.php

```

<?php
require_once 'ShowFile.class.php';
?>
<?php
/**
 * ShowFileクラスをインスタンス化する。
 * 内容を表示するファイルは、「ShowFile.class.php」
 */
try {
    $show_file = new ShowFile('./ShowFile.class.php');
}
catch (Exception $e) {
    die($e->getMessage());
}

/**
 * プレーンテキストとハイライトしたファイル内容をそれぞれ
 * 表示する
 */
$show_file->showPlain();
echo '<hr>';
$show_file->showHighlight();
?>

```

ShowFileクラスをインスタンス化し、それぞれのメソッドを呼び出せば、問題なく利用することができます。

しかし、すでに次のインターフェースが宣言されている場合を考えてみましょう。

●DisplaySourceFile.class.php

```

<?php
interface DisplaySourceFile
{
    /**
     * 指定されたソースファイルをハイライト表示する
     */
    public function display();
}
?>

```

DisplaySourceFileインターフェースには、ソースファイルをハイライト表示するためのメソッドdisplayが宣言されています。一方、テキスト表示させるメソッドは定義されていません。これは利用側に「利用させない」ようにするためです。

つまり、これを利用するクライアント側のコードは、次のようにしなければなりません。

●displayメソッドを呼び出すコード

```
<?php
:
$object->display();
:
?>
```

しかし、ShowFileクラスにはdisplayメソッドは定義されていません。何らかの方法で、クライアント側が利用できるよう、displayメソッドを用意してやる必要があります。

また、ShowFileクラスで定義されているshowPlainメソッドを、何らかの方法でクライアント側から隠す必要もあります。

まずは、ShowFileクラスをコピー＆ペーストして、新しいクラスを作成することを方法を考えてみます。確かに最も簡単な解決方法かもしれませんが、しかし、対象となるクラスの数が多い場合は、いくらコピー＆ペーストといえども大変な作業になります。さらに、コピー＆ペースト作業中にコーディングミスをしてしまい、その原因調査に無駄な時間を取られてしまう可能性もあります。

また、ShowFileクラスにバグなどによる修正やバージョンアップなどが行われた場合も、問題が発生します。予想できた人もいるかと思いますが、コピー＆ペーストして作成したすべてのファイルに対して、ShowFileクラスと同じ修正を行う必要が出てきてしまいます。そうしない限り、新しいShowFileクラスに差し替えることができません。この問題は、重大なバグやセキュリティホールへの修正が行われた場合に大きな影響を与えます。

次に、ShowFileクラスにdisplayメソッドを追加修正する方法を考えてみます。これもかなり簡単な解決方法です。場合によっては、うまくいきそうです。

#### ●ShowFile\_bad.class.php

```
<?php
require_once 'DisplaySourceFile.class.php';
?>
<?php
/**
 * 指定されたファイルの内容を表示するクラスです
 */
class ShowFile implements DisplaySourceFile
{
    /**
     * 内容を表示するファイル名
     *
     * @access private
     */
    private $filename;

    /**
     * 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;
    }

    /**
     * プレーンテキストとして表示します
     */
    public function showPlain()
    {
        echo '<pre>';
        echo htmlspecialchars(file_get_contents($this->filename), ENT_QUOTES);
        echo '</pre>';
    }

    /**
     * キーワードをハイライトして表示します
     */
    public function showHighlight()
    {
```

```
        highlight_file($this->filename);
    }

    /**
     * キーワードをハイライトして表示します
     * DisplaySourceFileインターフェースの実装
     */
    public function display()
    {
        highlight_file($this->filename);
    }
}
?>
```

しかしこの方法は、十分にテストされ使用実績があるクラスを変更することになります。ShowFileクラスがもっと複雑な場合、この方法を採用することで、ShowFileクラスの動作確認テストをすべて行わなければならないになります。また、再利用する上での重要事項である使用実績が失われることになります。

また、継承だけではshowPlainメソッドを隠すことができません。DisplaySourceFileインターフェースの意図と異なり、ShowFileクラスが提供する他のメソッドをクライアント側から実行することができてしまいます。

さらに、この場合もコピー＆ペーストの場合と同様、バージョンアップの際に問題になります。ShowFileクラスのコードを修正していますので、バージョンアップされたコードを再度追加修正するなどの対応が必要になります。

通常、すべてのコードをゼロからプログラミングするとは限りません。この例のように、今までに作成したコードやクラスを再利用することがよくあります。開発生産性や品質の向上を目的として、幅広くコードの再利用が行われています。特に、ShowFileクラスのように、利用実績もあり、テストも充分行われているコードであれば、再利用しない手はありません。しかも、**再利用するコード**については**一切変更を加えないで、必要となる機能を提供**できるよう変更したい...

**Adapterパターン**は、まさにこのために存在するパターンです。

## Adapter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。

あるクラスのインターフェースを、クライアントが求める他のインターフェースへ変換する。  
Adapterパターンは、インターフェースに互換性のないクラス同士を組み合わせることができるようになる。

Adapterパターンはクラスやオブジェクトの構造に注目したパターンで、**APIの異なるクラスどうしを繋ぎ合わせる**こと目的としています。

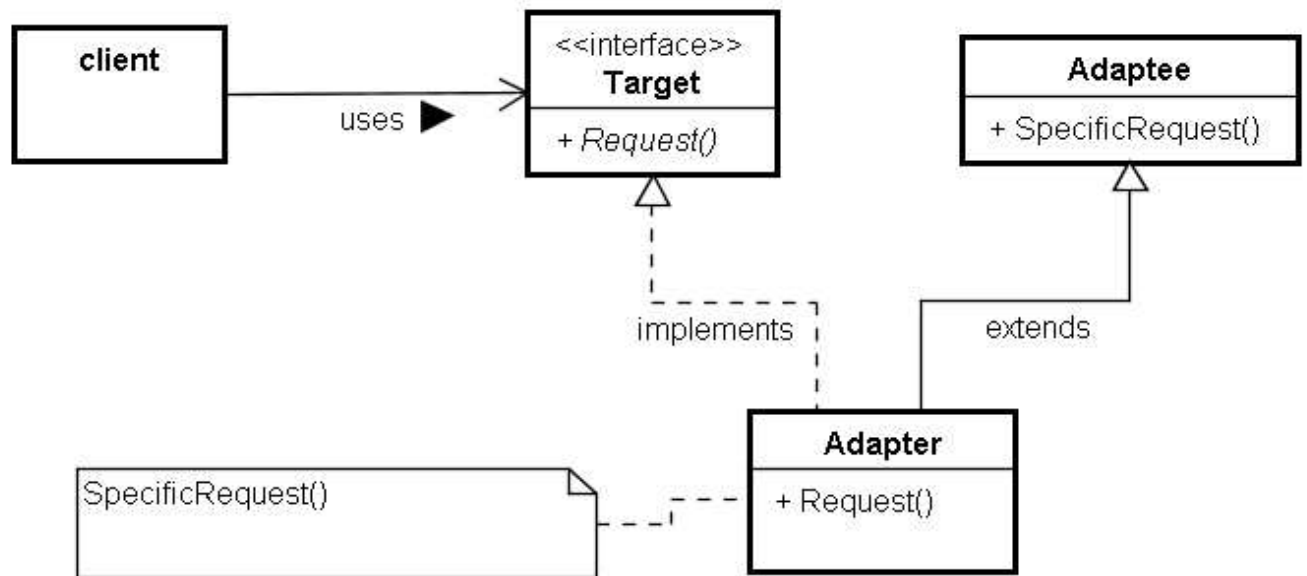
たとえば、すでに存在しているクラスを再利用する場合を考えてみましょう。このクラスは十分にテストされており、色々な場面で使用された実績を持っているクラスです。しかし、利用側が期待するAPIと異なったAPIが提供されているため、利用側を変更するには多大な労力を余儀なくされる事が予想されます。逆に、既存のクラスを変更することも考えられますが、それまでの利用実績や信頼性が損なわれてしまうことになります。

このような場合、既存クラスと利用側、**2つの異なるAPIを吸収してやるアダプタを用意してやる**ことで、変更範囲をおさえてクラスどうしを結合することができます。

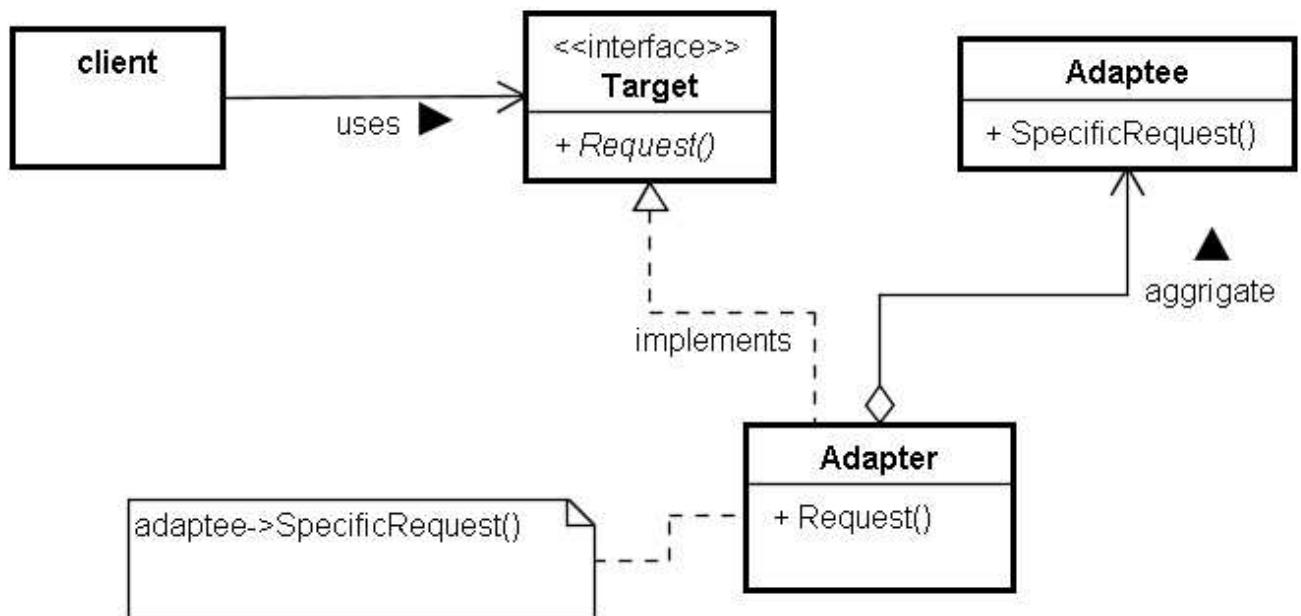
## Adapterパターンの構造

Adapterパターンの実装には、2種類あります。一つは**継承**を使ったAdapterパターン、もう一つは**委譲**を使ったAdapterパターンです。

継承を使った**Adapterパターン**のクラス図



委譲を使ったAdapterパターンのクラス図



## Adapterパターンの構成要素

いずれの場合も、構成要素は次のとおりです。

### Targetクラス

clientが要求するAPIを提供します。

### Adapteeクラス

AdapterクラスによってAPIを変更される側のクラスです。具体的な処理を提供します。

### Adapterクラス

Adapteeが提供するAPIをTargetが提供するAPIに変換するクラスです。

ここで「委譲」についてちょっと説明しておきます。委譲とは「権限などを他に任せて譲ること」です。ニュースでも「国・県から市町村への権限委譲」などと報道されますね。デザインパターンにおける「委譲」も同じ意味合いで、**具体的な処理内容を他のクラスに任せる**、という意味になります。

## Adapterパターンのメリット

Adapterパターンのメリットとしては、次のものが挙げられます。

既存のコードを修正することなく再利用できる

Adapterパターンでは、既存のクラスを変更するのではなく、**一枚皮をかぶせる**ようなクラスを作ります。このクラスに必要となるメソッドや機能を実装します。こうすることで、既存のクラスを一切変更することなく、新しいAPIとして提供することができます。この皮を被せることで既存のクラスを包み込んでいる様子から、**Wrapper(ラッパー)パターン**と呼ばれることもあります。

また、既存のクラスを変更しないため、バグが発生した場合の切り分けが非常に楽になります。特に、既存のクラスにはバグがないことが分かっている場合は、新たに作成したAdapterクラスを集中的に調べるだけで良いことになります。

利用側はアダプタの向こう側にある実装を意識する必要がない

クライアント側は、アダプタの向こう側のクラスが**どの様に実装されているかを意識する必要がありません**。

利用側がアダプタの向こう側にある実装を意識しない、もしくは意識する必要がない、ということは、利用側をまったく変更せずに**アダプタの向こう側にある実装を切り替える**ことができます。このことは、オブジェクト指向プログラミングを行う上で、非常に重要な位置付けにあります。

公開するAPIを制限する

Adapterパターンは異なるAPIを結びつけるパターンですが、結びつける際に**意図的に公開するAPIを制限することにも適用**することができます。

逆に、公開されていないprotectedメソッドを利用するために、Adapterパターンを適用することもできます。

## Adapterパターンの適用例

ここでは、冒頭に出てきたクラスをAdapterパターンを適用して再利用してみましょう。

まずは、継承を使ったAdapterパターンです。DisplaySourceFileインターフェースとShowFileクラスの間に、アダプタの役割を担うクラスを用意します。ここでは、DisplaySourceFileImplクラスとしておきます。このクラスは、新しいAPIを提供するDisplaySourceFileインターフェースを実装し、既存のShowFileクラスを継承したクラスです。

また、displayメソッドは、親クラスであるShowFileクラスのshowHighlightメソッドを使って実装します。

●継承を使った場合のDisplaySourceFileImpl.class.php

```
<?php
require_once 'DisplaySourceFile.class.php';
require_once 'ShowFile.class.php';
?>
<?php
/**
 * DisplaySourceFileを実装したクラス
 */
class DisplaySourceFileImpl extends ShowFile implements DisplaySourceFile
{
    /**
     * コンストラクタ
     */
    public function __construct($filename)
    {
        parent::__construct($filename);
    }

    /**
     * 指定されたソースファイルをハイライト表示する
     */
    public function display()
    {
        parent::showHighlight();
    }
}
?>
```

また、これを利用するクライアント側のコードは、次のようになります。

## ●adapter\_client.php

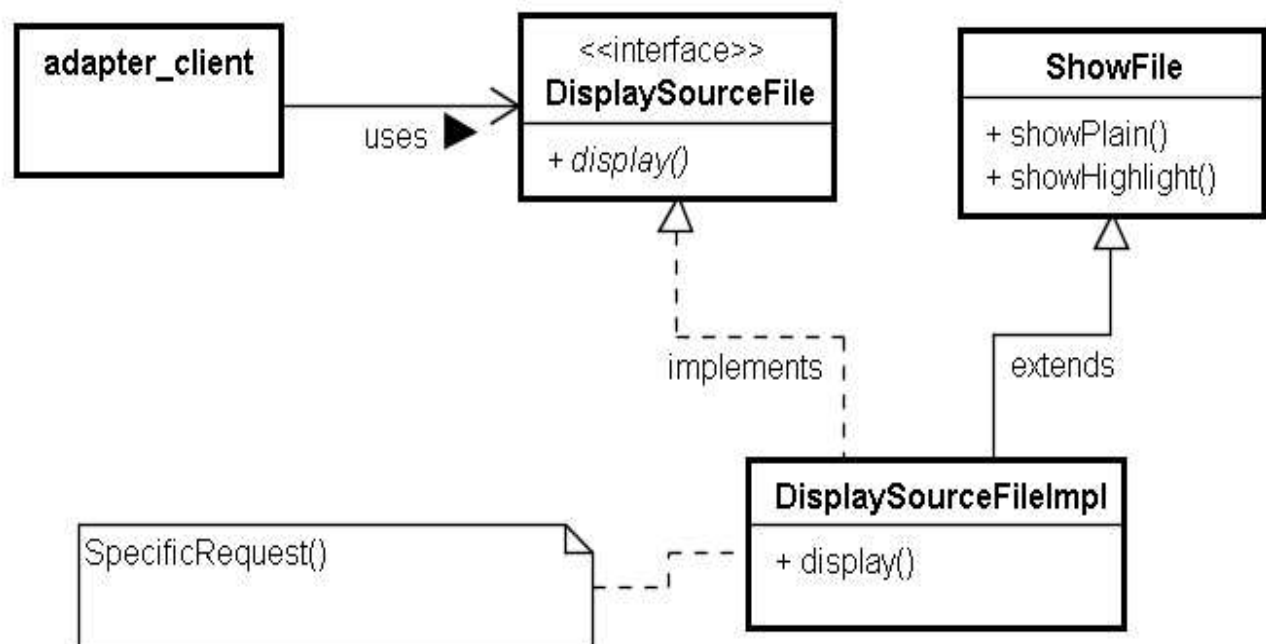
```

<?php
require_once 'DisplaySourceFileImpl.class.php';
?>
<?php
/**
 * DisplaySourceFileImplクラスをインスタンス化する。
 * 内容を表示するファイルは、「ShowFile.class.php」
 */
$show_file = new DisplaySourceFileImpl('./ShowFile.class.php');

/**
 * プレーンテキストとハイライトしたファイル内容をそれぞれ
 * 表示する
 */
$show_file->display();
?>

```

ここまでのコードをクラス図にすると、次のようになります。**adapter\_client**から**ShowFile**クラスは直接見えませんが、間接的にアクセスしていることが分かります。



adapter\_client.phpをブラウザから呼び出すと、画面にShowFile.class.phpの内容がハイライト表示されるはずです。どうでしょうか？表示されましたか？

次に委譲を使ったAdapterパターンです。継承を使った場合と同様、DisplaySourceFileインターフェイスとShowFileクラスの間、アダプタの役割を担うDisplaySourceFileImplクラスを用意します。しかし、今度はShowFileクラスは継承せず、DisplaySourceFileインターフェイスを実装したクラスになります。その代わり、**DisplaySourceFileImplクラスのコンストラクタ内で、ShowFileクラスをインスタンス化**します。そして、DisplaySourceFileImplクラスのメンバー変数として保持します。この「**既存のクラスを包み込んでいる様子**」がWrapperパターンとも呼ばれる理由です。

また、displayメソッドは、メンバー変数に格納されたShowFileインスタンスのshowHighlightメソッドを使って実装します。

## ●委譲を使った場合のDisplaySourceFileImpl.class.php

```

<?php
require_once 'DisplaySourceFile.class.php';
require_once 'ShowFile.class.php';
?>
<?php
/**
 * DisplaySourceFileを実装したクラス
 */

```



```

class DisplaySourceFileImpl implements DisplaySourceFile
{
    /**
     * ShowFileクラスのインスタンスを保持する
     */
    private $show_file;

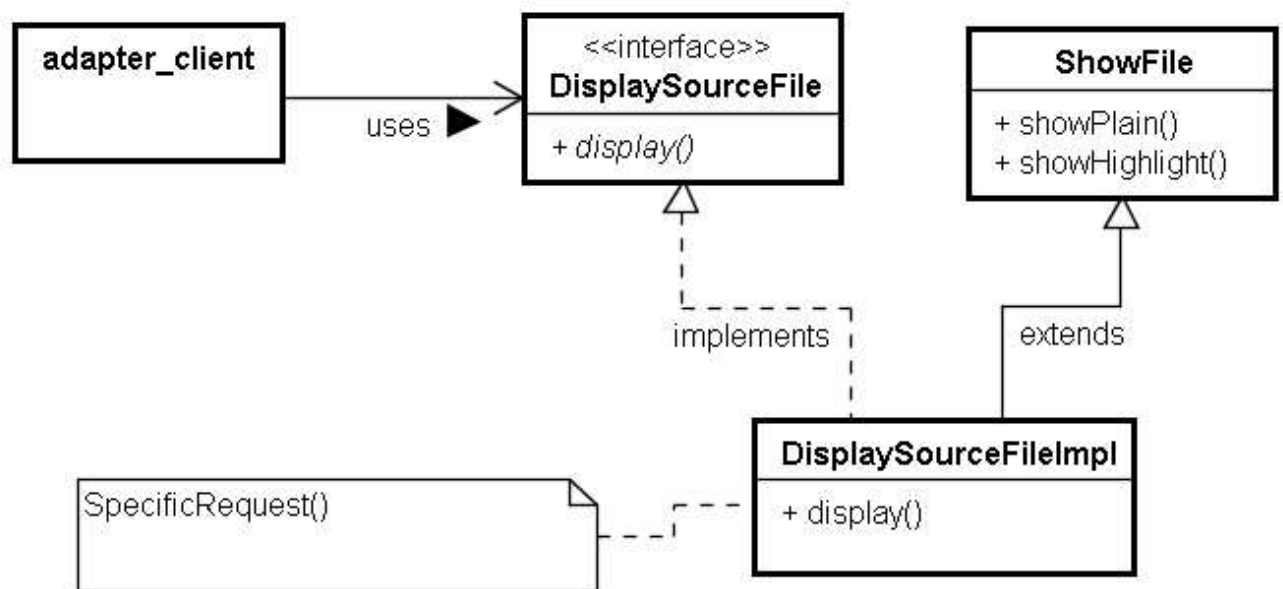
    /**
     * コンストラクタ
     */
    public function __construct($filename)
    {
        $this->show_file = new ShowFile($filename);
    }

    /**
     * 指定されたソースファイルをハイライト表示する
     */
    public function display()
    {
        $this->show_file->showHighlight();
    }
}
?>

```

なお、クライアント側のコードは、継承を使ったAdapterパターンの場合とまったく同じです。

こちらもクラス図を見てみましょう。継承を使った場合とほとんど同じですが、ShowFileクラスとDisplaySourceFileImplクラスの関係が違ってきます。



こちらもadapter\_client.phpをブラウザから呼び出してみましょう。継承を使ったAdapterパターンの場合と同じ内容が、画面に表示されたでしょうか？

ここで、Adapterパターンを適用するメリットを、おさらいします。

継承、委譲いずれの場合も、**既存クラスであるShowFileクラスには、一切変更が加わっていない**ことが分かります。つまり、ShowFileクラスの品質や利用実績を損なうことなく、再利用できたことを意味します。

また、**クライアント側に目的のメソッドのみを提供**することに成功しています。継承、委譲いずれの場合も、DisplaySourceFileインターフェースを通すことで、不要なメソッドを隠蔽しています。

一方のクライアント側のコードは、**DisplaySourceFileインターフェースに対してコーディングを行っておくことで、ShowFileクラスの実装を意識する必要がなくなります**。つまり、ShowFileクラスの修正やバージョンアップが行われた場合も、クライアント側のコードを修正する必要はありません。新たに実装したDisplaySourceFileImplクラスのみ修正すればよいからです。さらに、ShowFileクラスと同じ機能を持つ別のクラスや、より高機能、高性能なクラスに切り替える場合も、容易に行うことができます。

## Adapterパターンのオブジェクト指向的要素



Adapterパターンは、**継承**と**委譲**、**ポリモーフィズム**を利用したパターンです。

これまで見てきたように、継承を使ったAdapterパターンの場合、既存のクラスを継承して、新しいAPIを持つクラスを実装します。一方、委譲を使ったAdapterパターンの場合、既存のクラスを包み込むクラスを用意し、そのクラスで新しいAPIを提供します。ここで、具体的な処理自体は、包み込んだ既存のクラスに任せてしまうような実装を行います。

いずれの場合も、クライアント側にはインターフェースを通じて、新しいAPIを提供します。つまり、処理を実装したクラスは、クライアント側に提供されるインターフェースを実装しています。これにより、**クライアント側からはインターフェースしか見えなくなり、その向こうにある処理の実装を意識する必要がなくなります**。この「実装を意識することがなくなる」ため、実装を切り替える場合もアダプタに相当するクラスのみ修正するだけですみます。

## 関連するパターン

---

### Bridgeパターン

BridgeパターンはAdapterパターンの構造と非常によく似ていますし、本質的にも変わりありません。ただ、**適用する理由**が少々異なっています。

Adapterパターンは「既存クラスを再利用するために繋ぎ合わせる」といった後天的な理由で用いられます。

一方のBridgeパターンは「設計の段階で実装と機能を分離し、それぞれを繋ぎ合わせる」といった先天的な理由で導入されます。

### まとめ

---

ここではAdapterパターンについて見てきました。

既存のクラスを包み込むためのクラスを用意し、そのクラスにAPIを用意しました。これにより、既存クラスを一切変更することなく、利用側に新しいAPIを提供できることが理解できたと思います。