CODE > DESIGN PATTERNS

Design Patterns: The Adapter Pattern

by Avinash Zala 8 Nov 2014

Difficulty: Intermediate Length: Short Languages: English

Design Patterns PHP OOP Programming Fundamentals The Adapter Pattern Web Development

This post is part of a series called Design Patterns in PHP.

Design Patterns: The Facade Pattern

Design Patterns: The Decorator Pattern

In the last article, we looked at how the facade design pattern can be employed to simplify the employment of any large and complex system using only a simple facade class.

In this article, we will continue our discussion on design patterns by taking a look at the **adapter design pattern**. This particular pattern can be used when your code is dependent on some external API, or any other class that is prone to change frequently. This pattern falls under the category of "structural patterns" because it teaches us how our code and our classes should be structured in order to manage and/or extend them easily.

Again, I'd like to reiterate that design patterns have nothing new over traditional classes. Instead, they show us a better way to structure our classes, handle their behavior, and manage their creation.

The Problem

```php
<?php
class PayPal {

    public function __construct() {
        // Your Code here //
    }

    public function sendPayment($amount) {
        // Paying via Paypal //
        // Paying via Paypal //
        echo "Paying via PayPal: ". $amount;
    }
}

$paypal = new PayPal();
$paypal->sendPayment('2629');
```

In the above code, you can see that we are utilizing a PayPal class to simply pay the amount. Here, we are directly creating the object of the PayPal class and paying via PayPal. You have this code scattered in multiple places. So we can see that the code is using the `$paypal->sendPayment('amount here');` method to pay.

Some time ago, PayPal changed the API method name from `sendPayment` to `payAmount`. This should clearly indicate a problem for those of us who have been using the `sendPayment` method. Specifically, we need to change all `sendPayment` method calls to `payAmount`. Imagine the amount of code we need to change and the time we need to spend on testing each of the features once again.

The Solution

One solution to this problem is to use the adapter design pattern.

According to Wikipedia:

> *In software engineering, the adapter pattern is a software design pattern that allows the interface of an existing class*

to be used from another interface. It is often used to make existing classes work with others without modifying their source code.

In this case, we should create one wrapper interface which makes this possible. We will not make any changes in the external class library because we do not have control over it and it may change any time.

Let's dig into the code now, which shows the adapter pattern in action:

```
// Concrete Implementation of PayPal Class
class PayPal {

    public function __construct() {
        // Your Code here //
    }

    public function sendPayment($amount) {
        // Paying via Paypal //
        echo "Paying via PayPal: ". $amount;
    }
}

// Simple Interface for each Adapter we create
interface paymentAdapter {
    public function pay($amount);
}

class paypalAdapter implements paymentAdapter {

    private $paypal;

    public function __construct(PayPal $paypal) {
        $this->paypal = $paypal;
    }

    public function pay($amount) {
        $this->paypal->sendPayment($amount);
    }
}
```

Study the code above and you should be able to tell that we have not introduced any changes into the main `PayPal` class. Instead we have created one interface for our payment adapter and one adapter class for PayPal.

And so afterward we have made the object of the adapter class instead of the main `PayPal` class. While creating an object of adapter class we will pass the object of the main `PayPal` class as an argument, so that adapter class can have a reference to the main class and it can call the required methods of the main `PayPal` class.

Let's find out how we can utilize this method directly:

```
// Client Code
$paypal = new paypalAdapter(new PayPal());
$paypal->pay('2629');
```

Now imagine PayPal changes its method name from `sendPayment` to payAmount. Then we just need to make changes in `paypalAdapter`. Just have a look at the revised adapter code, which has just one change.

```
class paypalAdapter implements paymentAdapter {

    private $paypal;

    public function __construct(PayPal $paypal) {
        $this->paypal = $paypal;
        $this->paypal = $paypal;
    }

    public function pay($amount) {
        $this->paypal->payAmount($amount);
    }
}
```

So just one change and we are there.

Adding a New Adapter

At this point, we've seen how we can use the adapter design patten to overcome the aforementioned scenarios. Now, it's very easy to add a new class dependent on the existing adapter. Let's say the MoneyBooker API is there for payment.

Then instead of using the MoneyBooker class directly, we should be applying the same adapter pattern we just used for PayPal.

```
// Concrete Implementation of MoneyBooker Class
class MoneyBooker {

    public function __construct() {
        // Your Code here //
    }

    public function doPayment($amount) {
        // Paying via MoneyBooker //

        echo "Paying via MoneyBooker: ".  $amount;
    }
}
```

```
// MoneyBooker Adapter
class moneybookerAdapter implements paymentAdapter {

    private $moneybooker;

    public function __construct(MoneyBooker $moneybooker) {
        $this->moneybooker = $moneybooker;
    }

    public function pay($amount) {
        $this->moneybooker->doPayment($amount);
    }
}

// Client Code
$moneybooker = new moneybookerAdapter(new MoneyBooker());
$moneybooker->pay('2629');
```

As you can see, the same principles apply. You define a method that's available to third-party classes and then, if a dependency changes its API, you simply change the dependent class without exposing its external interface.

Conclusion

A great application is constantly hooked into other libraries and APIs, so I would propose that we implement the adapter method, so that we do not experience any trouble when a third-party API or library changes its code base.

I have tried my best to provide an elementary and yet useful example to demonstrate the adapter design pattern, but if you have additional comments or questions, please don't hesitate to add them in the feed below.

Advertisement

Avinash Zala

Avinash is a coder with over six years of experience in web development. With a strong focus on quality and usability, he is interested in delivering cutting edge applications. He is ready to work on

responsive applications targeted to various devices. He graduated with a bachelor degree in Information Technology. If you'd like to stay up to

date on his activities, refer to his blog or follow him on Twitter and Facebook.

XpertDevelopers

Advertisement

32 Comments Tuts+ Hub

1 Login

Recommend 7 Share

Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Marcos Borunda • 3 years ago

What I find amazing about Design Patterns is how stuff that you might already figure out by yourself, while coding, actually is a thing and has a name. Knowing the name for a design pattern and a structured definition make things clear and easy to express with your team.

19 • Reply • Share ›

techniczny Marcos Borunda • 3 years ago

What you really should find, is that if you knew about design patterns before when learning programming, you wouldn't need to figure it out later by yourself.

3 • Reply • Share ›

Mark Fox • 3 years ago

Nice example. Now, I will be a pedantic moron: class-names that aren't CamelCase look sloppy and weird.

6 • Reply • Share ›

Nathan Daly • 3 years ago

Great article :)

3 • Reply • Share ›

Xin Zheng • 3 years ago

Nice article! Thanks!!!

3 • Reply • Share ›

Michael Lawson • 3 years ago

Hooray for underutilized design patterns!

2 • Reply • Share ›

Mikle Kozachoc • 2 years ago

What about, if we alredy have used sendPayment() and in same time PayPal chenged name of method?

1 ^ | v • Reply • Share ›

Avinash ➦ Mikle Kozachoc • 2 years ago

In that case we need to replace new method name in `pay` method of our adapter class only.

```
public function pay($amount) {
     // Old Line
     $this->paypal->sendPayment($amount);

     // New line with new method name
     $this->paypal->sendPayment2($amount);
}
```

Hope this is clear

^ | v • Reply • Share ›

Mikle Kozachoc ➦ Avinash • 2 years ago

ok, but if we did not implement adapter before.

^ | v • Reply • Share ›

Avinash ➦ Mikle Kozachoc • 2 years ago

In that case you will need to change at all places where you have used that method. I have explained the same in problem section of this article. Hope that's clear.

^ | v • Reply • Share ›

Hari krishna • 3 years ago

What is the use of social adapter interface? What happens when we implement a class without interface?

1 ^ | v • Reply • Share ›

Wesam Alalem ➦ Hari krishna • 3 years ago

Program to interface, not implementation, check the first answer for SO question regarding this topic: http://stackoverflow.com/qu...

^ | v • Reply • Share ›

Bruce Lampson • 3 years ago

Good job Avinash Zala! Very comprehensible. I wrote a simple presentation few weeks ago explaining this pattern. In my example i use a global adapter that should be compatible with many mail clients via dependency injection. I would like to share this example with all of you. http://www.smsnica.com/reve... (Simply press your down arrow to checkout the implementation)

1 ^ | v • Reply • Share ›

Avinash ➦ Bruce Lampson • 3 years ago

Thanks for sharing... :)

1 ^ | v • Reply • Share ›

D3F ➦ Bruce Lampson • 2 years ago

Hi Bruce,

Nice presentation, however you are missing a semicolon in http://www.smsnica.com/reve... at line 2 ;)

En you say $clint i think this must be $client, but you are using it consistent so nothing breaks.

But nice presentation, very straightforward.

^ | v • Reply • Share ›

Bruce Lampson ➦ D3F • 2 years ago

Hey there D3F, Thanks for pointing that out!

^ | v • Reply • Share ›

Julius Koronci ➦ Bruce Lampson • 3 years ago

Nice presentation and very self explanatory :)

^ | v • Reply • Share ›

Nadeen Nilanka • 3 years ago

Understood, great article!

1 ^ | ˅ • Reply • Share ›

Paweł P. • 3 years ago

Patterns – *like always* in "**price**".

1 ^ | ˅ • Reply • Share ›

James Watadza • a year ago

Good tutorial again. Ignore the morons who have never written a single tutorial but spend hours criticizing and shooting down the work of those who cared enough to take the time to teach others.

^ | ˅ • Reply • Share ›

Mukesh Singh • 2 years ago

tutorial is amazing. please add more pattern

^ | ˅ • Reply • Share ›

Vhin Manansala • 2 years ago

Great article, but I have a question.

What's the difference of using interface when using

```
class paypalAdapter{
private $paypal;
public function __construct(PayPal $paypal) {
$this->paypal = $paypal;
}

public function pay($amount) {
$this->paypal->payAmount($amount);
}
}
```

Seems to work fine??

^ | v • Reply • Share ›

Mohamed Wael • 2 years ago

Very good article, thank you.

^ | v • Reply • Share ›

Avinash ➦ Mohamed Wael • 2 years ago

Glad you liked this.

^ | v • Reply • Share ›

manhee • 2 years ago

It'd be better to rename "interface paymentAdapter" to "
interface paymentInterface".

^ | v • Reply • Share ›

gundholmu • 2 years ago

Nice article and easy to understand. But I have some question about this pattern.

Why we need to create an interface class (paymentAdapter)? why we not just directly create an instantiate of PayPal class within the construct class of paypalAdapter?

^ | v • Reply • Share ›

Ankush Thakur ➦ gundholmu • 2 years ago

Because by making all classes implement an interface, we can be sure all of them use the same function name for payment. If you were directly wrapping classes, you might mistakenly name one of the adapter functions as pay() and the other as payNow(), causing confusion to yourself.

^ | v • Reply • Share ›

Yanwar • 2 years ago

what adapter can be combined with the facade?

^ | v • Reply • Share ›

Elfan Nofiari • 3 years ago

One umbrella term for this would be single source of authority, or DRY, Don't Repeat Yourself.

Advertisement

LOOKING FOR SOMETHING TO HELP KICK START YOUR NEXT PROJECT?