Use the adapter pattern to introduce new interface to your existing code

Published October 01, 2015

In order to understand what the **adapter pattern** is good for, try to imagine the following non-fictional scenario. After much sweat, tears, and toil you managed to incorporate a payment library in to your app (the payzilla library), but then you are asked to incorporate a new one (called paykal). Both libraries do (almost) the same job, but alas, the names that the paykal developers gave to the methods in their library is different than the names used by payzilla.

What a drag! Or maybe not, if only we use the **adapter pattern**.

The **adapter pattern** recommends using an **adapter class** to translate the [interface](https://phpenthusiast.com/object-oriented-php-tutorials/interfaces) of the new class to the interface of the existing class that we are used to use. We use the **adapter pattern** when two classes or more do essentially the same job, but have different names for their methods.

*The****adapter pattern****recommends using an****adapter class****to translate the interface of the new class to the interface of the existing class that we are used to use.*

How to write an **adapter class** we'll see in a minute, but first let's get the ball rolling by introducing the first class that we use in order to pay.

The original class

The PayWithPayzilla class is used for paying with the Payzilla payment provider company. It adds an item with the addItem method, and adds the item's price to the total sum to be paid with the addPrice method.

class PayWithPayZilla {

  function addItem($itemName)

  {

    var\_dump("1 item added: " . $itemName );

  }

  function addPrice($itemPrice)

  {

    var\_dump("1 item added to total with the price of: " . $itemPrice );

  }

}

The Customer class uses the PayWithPayZilla class in order to pay.

class Customer {

  private $pay;

  function \_\_construct($pay)

  {

    $this -> pay = $pay;

  }

  function buy($itemName, $itemPrice)

  {

    $this -> pay -> addItem($itemName);

    $this -> pay -> addPrice($itemPrice);

  }

}

Let's test the code by making the customer pay for a lollipop with the price of 2. For this purpose, we inject an instance of PayWithPayzilla to the Customer class, and then use the customer's buy method in order to pay.

$pay = new PayWithPayZilla();

$customer = new Customer($pay);

$customer -> buy("lollipop", 2);

And the result:

string '1 item added: lollipop' (length=22) string '1 item added to total with the price of: 2' (length=42)

The new class a.k.a. the problem

It is all fine and dandy, but now for some reason we need to use another payment provider. The reason might be the need to support another payment platform (Alipay vs. Paypal), or because we can no longer work with the same payment service provider. Whatever the reason may be, we face an issue that might cause us to want to re-write everything in our code that consumes the payment library. Or maybe there's a better way?

Take for example, the PayKal class. This code is used for another payment platform, and functions (almost) exactly like PayWithPayzilla, but the names that the developers of PayKal chose for their methods' names are different, and they also added a new method that doesn't exist in the PayWithPayzilla class.

That's the Paykal class:

class PayKal {

  function addOneItem($name)

  {

    var\_dump("1 item added: " . $name);

  }

  function addPriceToTotal($price)

  {

    var\_dump("1 item added to total with the price of: " . $price);

  }

  // Unique method

  addItemAndPrice($name,$price)

  {

    $this -> addOneItem($name);

    $this -> addPriceToTotal($price);

  }

}

The addOneItem method adds the name of the item in the PayKal class, in the same way that addItem do for the PayZilla class. And addPriceToTotal adds the item's price to the total, just as the addPrice does in the old class. But if we try to pass a PayKal object to the Customer class, instead of a PayZilla object, we are going to face a fatal error.

Let's give it a try:

$pay = new PayKal();

$customer = new Customer($pay);

$customer -> buy("lollipop", 2);

And the result is, not surprisingly, a fatal error:

Fatal error: Call to undefined method PayKal::addItem()

While the classes do essentially the same task, we cannot replace between the two classes because they have different names for their methods.

The **adapter pattern** suggests that we solve the incompatibility problem by translating the new class [interface](https://phpenthusiast.com/object-oriented-php-tutorials/interfaces) to the old class interface.

In our example, we need to translate the new PayKal class's methods to the old terminology which is used for the PayWithPayzilla class's methods names.

In order to translate the new interface to the old interface, the **adapter pattern** suggests writing an **adapter class** with the following two main features:

1. The **adapter class** needs to implement the interface of the original class.
2. The **adapter class** needs to hold a reference to the new class.

In order to write an **adapter class** we need the old class to have an interface, and since PayWithPayzilla doesn't have any, we first need to extract one.

1. Extract an interface

As we already explained, the old class needs to have an interface, but since it doesn't have an interface let's extract a PayZilla interface from the PayWithPayzilla class. Like the PayWithPayzilla class, the interface will have the following two methods:

interface PayZilla {

  function addItem($itemName);

  function addPrice($itemPrice);

}

Now, the original PayWithPayzilla class needs to implement the PayZilla interface that we have just extracted from it.

class PayWithPayZilla implements PayZilla {

  function addItem($itemName)

  {

    var\_dump("1 item added: " . $itemName );

  }

  function addPrice($itemPrice)

  {

    var\_dump("1 item added to total with the price of: " . $itemPrice );

  }

}

2. Write the adapter class

Now, that the old PayWithPayzilla class has an interface to implement, we can write the **adapter class**. Let's call the **adapter class** PayKalToPayZillaAdapter, and make it implement the PayZilla interface.

1class PayKal2PayZillaAdapter implements PayZilla {}

Except for implementing the interface, the **adapter class** needs to hold a reference to the new PayKal class. For this purpose, we'll inject the new class's object to the constructor, and hold a reference to it in the private $payObj variable.

// The adapter implements the original class

class PayKal2PayZillaAdapter implements PayZilla {

  // The adapter holds a reference to the new class.

  private $payObj;

  // In order to hold a reference, we need to pass the new

  // class's object throught the constructor.

  function \_\_construct($payObj)

  {

    $this -> payObj = $payObj;

  }

}

*The****adapter class****translates the old interface to the new one by giving the methods in the adapter class the names of the old class's methods, while using the new class's methods as the methods' bodies.*

Now, the **adapter class** will translate the old interface to the new one by giving the methods in the adapter class the names of the old class's methods, while using the new class's methods as the methods' bodies. For example, the addItem method from the old PayWithPayzilla class, will wrap the addOneItem method from the new PayKal class.

// The adapter implements the original class

class PayKal2PayZillaAdapter implements PayZilla {

  // The adapter holds a reference to the new class.

  private $payObj;

  // In order to hold a reference, we need to pass the new

  // class's object throught the constructor.

  function \_\_construct($payObj)

  {

    $this -> payObj = $payObj;

  }

  // The name of the methods is that of the old class.

  // The code within the methods uses the code of the new class.

  function addItem($itemName)

  {

    $this -> payObj -> addOneItem($itemName);

  }

}

Let's run the code:

$payKal = new PayKal();

$pay = new PayKal2PayZillaAdapter($payKal);

$customer = new Customer($pay);

$customer -> buy("lollipop", 2);

And find that the adapter performed its job.

string '1 item added: lollipop' (length=22) string '1 item added to total with the price of: 2' (length=42)

The **adapter pattern** essentially wraps the old class's methods with the new class's method's names, and does so by:

1. Implementing the old class's interface for the names of the adapter's methods.
2. Holding a reference to the new class, and using the new class's methods as the adapter's methods bodies.

In conclusion

The **adapter pattern** is one of the simplest and most used patterns. We use it in those cases that we incorporate a new class that essentially performs the same task as an old class, but it has different names for the methods. The adapter pattern suggests that we solve the problem by using an **adapter class**. The methods of the adapter class, are constructed from the new class's methods as bodies wrapped with the names of the old class's methods. By doing so, the adapter translates the new interface to the old interface, and so saves us the trouble of re-writing our existing code.