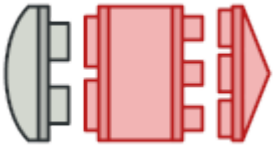

Adapter en PHP

Adapter es un patrón de diseño estructural que permite colaborar a objetos incompatibles.

El patrón Adapter actúa como envoltorio entre dos objetos. Atrapa las llamadas a un objeto y las transforma a un formato y una interfaz reconocible para el segundo objeto.

Aprende más sobre el patrón Adapter →

Navegación

Complejidad: ★☆☆

Popularidad: ★★★

Ejemplos de uso: El patrón Adapter es muy común en el código PHP. Se utiliza muy a menudo en sistemas basados en algún código heredado. En estos casos, los adaptadores crean código heredado con clases modernas.

tipo de clase abstracta/interfaz. Cuando el adaptador recibe una llamada a uno de sus métodos, convierte los parámetros al formato adecuado y después dirige la llamada a uno o varios métodos del objeto envuelto.

Ejemplo conceptual

Este ejemplo ilustra la estructura del patrón de diseño **Adapter** y se centra en las siguientes preguntas:

- ¿De qué clases se compone?
- ¿Qué papeles juegan esas clases?
- ¿De qué forma se relacionan los elementos del patrón?

Después de conocer la estructura del patrón, será más fácil comprender el siguiente ejemplo basado en un caso de uso real de PHP.

index.php: Ejemplo conceptual

```php
<?php

namespace RefactoringGuru\Adapter\Conceptual;

/**
 * The Target defines the domain-specific interface used by the client code.
 */
class Target
{
    public function request(): string
    {
        return "Target: The default target's behavior.";
    }
}

/**
 * The Adaptee contains some useful behavior, but its interface is incompatible
 * with the existing client code. The Adaptee needs some adaptation before the
 * client code can use it.
 */
class Adaptee
{
    public function specificRequest(): string
    {
        return ".eetpadA eht fo roivaheb laicepS";
```

```php
/**
 * The Adapter makes the Adaptee's interface compatible with the Target's
 * interface.
 */
class Adapter extends Target
{
    private $adaptee;

    public function __construct(Adaptee $adaptee)
    {
        $this->adaptee = $adaptee;
    }

    public function request(): string
    {
        return "Adapter: (TRANSLATED) " . strrev($this->adaptee->specificRequest());
    }
}

/**
 * The client code supports all classes that follow the Target interface.
 */
function clientCode(Target $target)
{
    echo $target->request();
}

echo "Client: I can work just fine with the Target objects:\n";
$target = new Target();
clientCode($target);
echo "\n\n";

$adaptee = new Adaptee();
echo "Client: The Adaptee class has a weird interface. See, I don't understand it:\n";
echo "Adaptee: " . $adaptee->specificRequest();
echo "\n\n";

echo "Client: But I can work with it via the Adapter:\n";
$adapter = new Adapter($adaptee);
clientCode($adapter);
```

Output.txt: Resultado de la ejecución

```
Client: I can work just fine with the Target objects:
Target: The default target's behavior.
```

```
Client: But I can work with it via the Adapter:
Adapter: (TRANSLATED) Special behavior of the Adaptee.
```

Ejemplo del mundo real

El patrón **Adapter** te permite utilizar clases de terceros o heredadas incluso aunque sean incompatibles con el grueso de tu código. Por ejemplo, en lugar de reescribir la interfaz de notificación de tu aplicación para que soporte todos los servicios de terceros, como Slack, Facebook, SMS u otros, puedes crear un grupo de envoltorios especiales que adaptan las llamadas desde tu aplicación a una interfaz y formato requerido por cada una de las clases de terceros.

index.php: Ejemplo del mundo real

```php
<?php

namespace RefactoringGuru\Adapter\RealWorld;

/**
 * The Target interface represents the interface that your application's classes
 * already follow.
 */
interface Notification
{
    public function send(string $title, string $message);
}

/**
 * Here's an example of the existing class that follows the Target interface.
 *
 * The truth is that many real apps may not have this interface clearly defined.
 * If you're in that boat, your best bet would be to extend the Adapter from one
 * of your application's existing classes. If that's awkward (for instance,
 * SlackNotification doesn't feel like a subclass of EmailNotification), then
 * extracting an interface should be your first step.
 */
class EmailNotification implements Notification
{
    private $adminEmail;

    public function __construct(string $adminEmail)
    {
        $this->adminEmail = $adminEmail;
```

```
    public function send(string $title, string $message): void
    {
        mail($this->adminEmail, $title, $message);
        echo "Sent email with title '$title' to '{$this->adminEmail}' that says '$message
    }
}

/**
 * The Adaptee is some useful class, incompatible with the Target interface. You
 * can't just go in and change the code of the class to follow the Target
 * interface, since the code might be provided by a 3rd-party library.
 */
class SlackApi
{
    private $login;
    private $apiKey;

    public function __construct(string $login, string $apiKey)
    {
        $this->login = $login;
        $this->apiKey = $apiKey;
    }

    public function logIn(): void
    {
        // Send authentication request to Slack web service.
        echo "Logged in to a slack account '{$this->login}'.\n";
    }

    public function sendMessage(string $chatId, string $message): void
    {
        // Send message post request to Slack web service.
        echo "Posted following message into the '$chatId' chat: '$message'.\n";
    }
}

/**
 * The Adapter is a class that links the Target interface and the Adaptee class.
 * In this case, it allows the application to send notifications using Slack
 * API.
 */
class SlackNotification implements Notification
{
    private $slack;
    private $chatId;

    public function __construct(SlackApi $slack, string $chatId)
    {
        $this->slack = $slack;
        $this->chatId = $chatId;
```

```
    /**
     * An Adapter is not only capable of adapting interfaces, but it can also
     * convert incoming data to the format required by the Adaptee.
     */
    public function send(string $title, string $message): void
    {
        $slackMessage = "#" . $title . "# " . strip_tags($message);
        $this->slack->logIn();
        $this->slack->sendMessage($this->chatId, $slackMessage);
    }
}

/**
 * The client code can work with any class that follows the Target interface.
 */
function clientCode(Notification $notification)
{
    // ...

    echo $notification->send("Website is down!",
        "<strong style='color:red;font-size: 50px;'>Alert!</strong> " .
        "Our website is not responding. Call admins and bring it up!");

    // ...
}

echo "Client code is designed correctly and works with email notifications:\n";
$notification = new EmailNotification("developers@example.com");
clientCode($notification);
echo "\n\n";


echo "The same client code can work with other classes via adapter:\n";
$slackApi = new SlackApi("example.com", "XXXXXXXX");
$notification = new SlackNotification($slackApi, "Example.com Developers");
clientCode($notification);
```

Output.txt: Resultado de la ejecución

```
Client code is designed correctly and works with email notifications:
Sent email with title 'Website is down!' to 'developers@example.com' that says '<strong s

The same client code can work with other classes via adapter:
```

Ejemplo conceptual | Ejemplo del mundo real

Ukrainian office:
FOP Olga Skobeleva
Abolmasova 7
Kyiv, Ukraine, 02002
Email: support@refactoring.guru

Spanish office:
Oleksandr Shvets
Avda Pamplona 63, 4b
Pamplona, Spain, 31010
Email: spain@refactoring.guru