Capítulo 4
261

1. Funciones

1.1 Introducción

Al igual que en otros lenguajes de programación, PHP ofrece la posibilidad de definir sus propias funciones (llamadas funciones del "usuario") con todas las ventajas asociadas (modularidad, uso de mayúsculas...). Una función es un conjunto de instrucciones identificadas por un nombre, cuya ejecución devuelve un valor y cuya llamada se puede utilizar como operando en una expresión. Un procedimiento es un conjunto de instrucciones identificadas por un nombre que puede ser llamado como una instrucción.

1.2 Declaración y llamada

La palabra clave `función` permite introducir la definición de una función.

Sintaxis

```
función nombre_función([parámetro]) [: tipo]{
  instrucciones;
}
```

`nombre_función`	Nombre de la función (debe respetar las reglas de denominación presentes en el capítulo Introducción a PHP - Estructura básica de una página PHP). En este nombre no se diferencian mayúsculas y minúsculas (para PHP, las funciones `unafunción` y `UnaFunción` son las mismas).
`parámetro`	Parámetros posibles de la función expresados como una lista de variables (véase la sección Parámetros): `$parámetro1, $parámetro2, ...`
`tipo`	Declaración del tipo de datos devuelto por la función. Valores posibles: `int`, `float`, `string`, `bool`, `array`, `callable`, `iterable`, `object`, `mixed`, `void`, un nombre de clase o de interfaz (véase en este capítulo la sección Clases) o una unión de tipos. El nombre del tipo se puede preceder por un punto de interrogación (`?`) que indica que la función puede devolver un valor `NULL`. En el capítulo Introducción a PHP puede encontrar la definición de los tipos de datos (apartado Las bases del lenguaje PHP - Tipos de datos).
`instrucciones`	Conjunto de instrucciones que componen la función.

El nombre de la función no debe ser una palabra reservada de PHP (nombre de función nativa, de instrucción) ni ser igual al nombre de otra función definida de antemano.

Una función de usuario se puede llamar como una función nativa de PHP: en una asignación, en una comparación, etc.

Si la función devuelve un valor, es posible utilizar la instrucción `return` para definir el valor de retorno de la función.

Sintaxis

```
return expresión;
```

`expresión`	Expresión cuyo resultado constituye el valor de retorno de la función (`NULL` por defecto).

El resultado de una función puede ser de cualquier tipo (cadena, número, matriz, etc.).

La instrucción `return` detiene la ejecución de la función y devuelve el resultado de `expresión` a quien realiza la llamada. Si hay varias instrucciones `return` en la función, la primera que se encuentre en el desarrollo de las instrucciones es la que define el valor de retorno y provoca la interrupción de la función. Si la función no incluye ninguna instrucción `return` (o si no se ha ejecutado ninguna instrucción `return`), el valor de retorno de la función es `NULL`.

© Editions ENI - All rights reserved

Ejemplo

```
<?php
// Función sin parámetro que muestra "¡Hola!"
// Sin valor de retorno.
function mostrar_hola() {
  echo '¡Hola!<br />';
}
// Función con dos parámetros que devuelve el producto
// de los dos parámetros.
function producto($valor1,$valor2) {
  return $valor1 * $valor2;
}
// Llamada de la función mostrar_hola.
mostrar_hola();
// Usos de la función producto:
// - en una asignación
$resultado = producto(2,4);
echo "2 x 4 = $resultado<br />";
// - en una comparación
if (producto(10,12) > 100) {
  echo '10 x 12 es superior a 100.<br />';
}
?>
```

Resultado

```
¡Hola!
2 x 4 = 8
10 x 12 es superior a 100.
```

Observación

En el lenguaje PHP, no existe un procedimiento real. Para definir algo equivalente a un procedimiento, basta con definir una función que no devuelva ningún valor y llamar a la función como si se tratara de una instrucción (como la función `mostrar_hola`, por ejemplo). Una función que no devuelve nada, se puede declarar de manera explícita con el tipo de retorno `void`.

Como ya mencionamos anteriormente, el contenido de una matriz se puede transformar en una lista de parámetros dentro de una llamada de función gracias al operador `...` (tres puntos suspensivos).

Ejemplo

```
<?php
// Función con tres parámetros que devuelve la suma
// de los tres parámetros.
function suma($valor1,$valor2,$valor3) {
  return $valor1 + $valor2 + $valor3;
}
// Transformación del contenido de una matriz en
// lista de parámetros.
$valores = [1,2,3];
echo '1 + 2 + 3 = ',suma(...$valores),'<br />';
// Lo mismo para una parte solamente de los parámetros
// con una matriz definida directamente en la llamada.
echo '1 + 2 + 4 = ',suma(1,...[2,4]),'<br />';
?>
```

Resultado

```
1 + 2 + 3 = 6
1 + 2 + 4 = 7
```

Cuando una función devuelve una matriz, es posible acceder directamente a un elemento de la matriz llamando a la función con una sintaxis de tipo `función(...)[clave]`.

Ejemplo

```
<?php
// Definición de una función que devuelve una matriz.
function quien() {
  return ['Olivier','Heurtel'];
}
// Llamada a la función y recuperación directa del nombre almacenado
// en el índice 0 de la matriz devuelta.
$nombre = quien()[0];
echo "quien()[0] = $nombre<br />";
?>
```

Resultado

```
quien()[0] = Olivier
```

Esta técnica funciona también cuando la función devuelve una matriz multidimensional con una sintaxis de tipo `función(...)[clave1][clave2]`.

Es posible utilizar una función antes de definirla.

© Editions ENI - All rights reserved

Ejemplo

```
<?php
// Utilización de la función producto.
echo producto(5,5);
// Definición de la función producto.
function producto($valor1,$valor2) {
  return $valor1 * $valor2;
}
?>
```

Resultado

```
25
```

Por lo tanto, no hay ningún problema para definir funciones que se llamen entre ellas.

Observación

Una función se puede utilizar solo en el script donde se define. Para utilizarla en varios scripts, es necesario o bien copiar su definición en los diferentes scripts (se pierde el interés por definir una función) o bien definirla en un archivo incluido donde la función sea necesaria.

Ejemplo

– Archivo `funciones.inc` que contiene la definición de funciones:

```
<?php
// Definición de la función producto.
function producto($valor1,$valor2) {
  return $valor1 * $valor2;
}
?>
```

– Script que utiliza las funciones definidas en `funciones.inc`:

```
<?php
// Inclusión del archivo contenedor de la definición de las funciones.
include('funciones.inc');
// Utilización de la función producto.
echo producto(5,5);
?>
```

Declaración del tipo de retorno

Es posible definir el tipo de datos devuelto por una función.

Cuando es el caso, en el modo de funcionamiento por defecto (a diferencia de modo estricto que se presenta a continuación), PHP realiza si es necesario una conversión automática del valor devuelto al tipo de datos declarado.

Ejemplo

```
<?php
// Declaración de dos funciones que devuelven el producto
// de dos parámetros, el segundo especifica un tipo
// de datos "entero" para el valor de retorno.
function producto1($valor1,$valor2) {
  return $valor1 * $valor2;
}
function producto2($valor1,$valor2) : int {
  return $valor1 * $valor2;
}
// Llamada de dos funciones con los mismos parámetros
echo 'producto1(20,1/7) => ',var_dump(producto1(20,1/7)),'<br />';
echo 'producto2(20,1/7) => <b>',var_dump(producto2(20,1/7)),'</b><br />';
?>
```

Resultado

```
producto1(20,1/7) => float(2.8571428571428568)
producto2(20,1/7) => int(2)
```

En este ejemplo, podemos ver claramente que PHP ha convertido el valor devuelto por la segunda función a un valor entero (con las reglas de conversión mencionadas en el capítulo Introducción a PHP - Las bases del lenguaje PHP - Tipos de datos).

Si PHP no es capaz de realizar la conversión (tipos de datos no convertibles entre ellos), se produce una excepción `TypeError`. Esta excepción detiene el script si no se maneja (véase en este capítulo la sección Clases - Excepciones).

Ejemplo

```
<?php
// Declaración y llamada de una función que debe devolver una
// matriz pero que devuelve una cadena de caracteres.
function quien() : array {
  return 'Olivier Heurtel';
}
echo 'quien()[0] = ',quien()[0];
?>
```

Resultado

```
quien()[0] =
Fatal error: Uncaught TypeError: Return value of quien() must be of the
type array, string returned in /app/scripts/index.php:5 Stack trace: #0
/ app/scripts/index.php(7): quien() #1 {main} thrown in/app/scripts/
index.php on
line 5
```

© Editions ENI - All rights reserved

Una función declarada con un tipo de retorno diferente de `void`, debe devolver un valor no `NULL`. Si este no es el caso, se devuelve un error, diferente según el caso:

Ausencia de instrucción `return`

```
Fatal error: Uncaught TypeError: Return value of MiFuncion() must be of
the type int, none returned in ...
```

Instrucción `return vacía`

```
Fatal error: A function with return type must return a value in ...
```

Instrucción `return NULL`

```
Fatal error: Uncaught TypeError: Return value of MiFuncion() must be of
type int, null returned in ...
```

Para autorizar una función para que devuelva un valor `NULL`, hay que utilizar un punto de interrogación (`?`), delante del nombre del tipo (diferente de `void`).

```
<?php
// Declaración y llamada de una función que especifica un
// tipo de datos de retorno que puede ser NULL
function cubo($valor) : ?int {
  if (is_null($valor)) {
    return NULL;
  } else {
    return $valor ** 3 ;
  }
}
echo 'cubo(2) => <b>',var_dump(cubo(2)),'</b><br />';
echo 'cubo(NULL) => <b>',var_dump(cubo(NULL)),'</b><br />';
?>
```

Resultado

```
cubo(2) => int(8)
cubo(NULL) => NULL
```

Incluso con esta opción, la función debe tener una instrucción `return` no vacía. Si no es el caso, se devuelve un error, diferente según el caso:

Ausencia de instrucción `return`

```
Fatal error: Uncaught TypeError: Return value of MiFuncion() must be of
type int, none returned in ...
```

Instrucción `return` vacía

```
Fatal error: A function with return type must return a value (did you
mean "return null;" instead of "return;"?) in ...
```

Capítulo 12
El patrón de diseño Composite

1. Descripción

El objetivo del patrón de diseño `Composite` es materializar composiciones de objetos en forma de estructura de árbol. Estas composiciones están concebidas de tal forma que un cliente tratará indiferentemente un componente y un compuesto.

2. Ejemplo

En nuestro sistema de venta de vehículos, queremos representar las empresas cliente, en especial para conocer el número de vehículos de los que disponen y proporcionarles ofertas de mantenimiento para su parque de vehículos.

Las empresas que posean filiales solicitan ofertas de mantenimiento que tengan en cuenta el parque de vehículos de sus filiales.

Una solución inmediata consiste en procesar de forma diferente las empresas sin filiales y las que posean filiales. No obstante esta diferencia en el procesado entre ambos tipos de empresa haría que la aplicación fuera más compleja y totalmente dependiente de la composición interna de las empresas cliente.

El patrón de diseño `Composite` resuelve este problema unificando ambos tipos de empresa y utilizando la composición recursiva. Esta composición recursiva es necesaria puesto que una empresa puede tener filiales que posean, ellas mismas, otras filiales. Se trata de una composición en árbol tal y como se ilustra en la figura 12.1 donde las empresas madre se sitúan sobre sus filiales. Por razones evidentes de simplificación, suponemos que no hay filiales comunes a dos empresas.

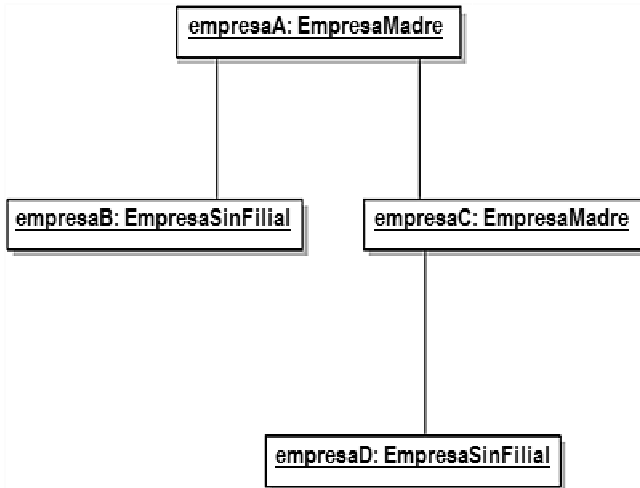

Figura 12.1 - Árbol de empresas madres y de sus filiales

La figura 12.2 muestra el diagrama de clases correspondiente. La clase abstracta `Empresa` contiene la interfaz destinada a los clientes. Posee dos subclases concretas, a saber `EmpresaSinFilial` y `EmpresaMadre`, esta última guarda una relación de agregación con la clase `Empresa` representando los enlaces con sus filiales.

© Editions ENI - All rights reserved

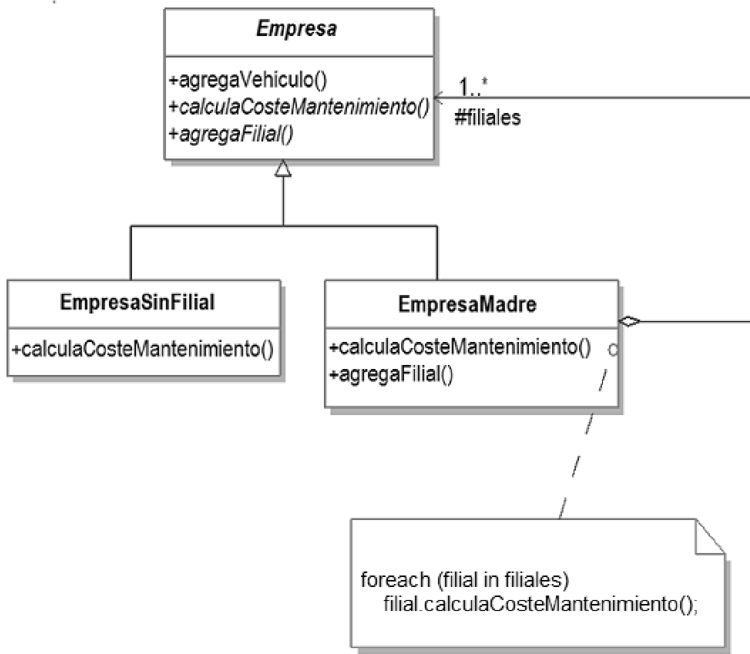

Figura 12.2 - El patrón de diseño `Composite` *aplicado a la representación de empresas y sus filiales*

La clase `Empresa` posee tres métodos públicos de los cuales dos son abstractos. El método concreto es el método `agregaVehiculo` que no depende de la composición en filiales de la empresa. En cuanto a los otros dos métodos, se implementan en las subclases concretas (`agregaFilial` no tiene implementación en `EmpresaSinFilial`, por eso no se representa en el diagrama de clases).

3. Estructura

3.1 Diagrama de clases

La figura 12.3 detalla la estructura genérica del patrón de diseño `Composite`.

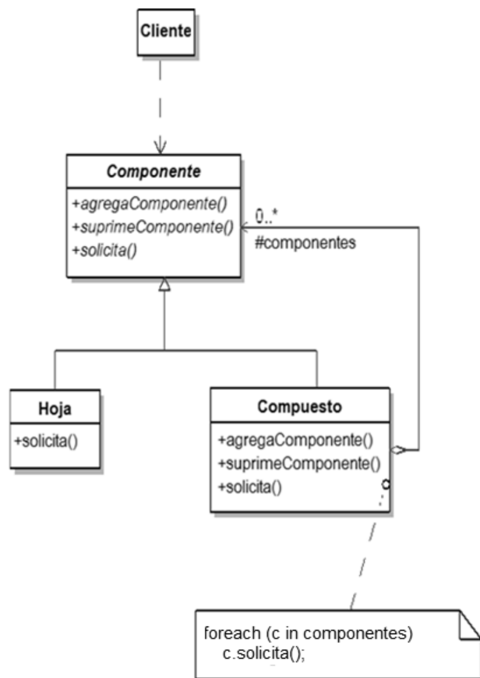

Figura 12.3 - Estructura del patrón de diseño `Composite`

3.2 Participantes

Los participantes del patrón de diseño `Composite` son los siguientes:

– `Componente` (`AbstractEmpresa`) es la clase abstracta que describe la interfaz de los objetos de la composición, implementa los métodos comunes e introduce la firma de los métodos que gestionan la composición agregando o suprimiendo componentes.

© Editions ENI - All rights reserved

– `Hoja` (`EmpresasSinFilial`) es la clase concreta que representa las hojas de la composición (en una estructura de árbol, una hoja no posee componentes, es un nudo terminal).
– `Compuesto` (`EmpresaMadre`) es la clase concreta que representa los objetos compuestos de la jerarquía. Esta clase posee una asociación de agregación con la clase `Componente`.
– `Cliente` es la clase de los objetos que acceden a los objetos de la composición y los manipulan.

3.3 Colaboraciones

Los clientes envían sus peticiones a los componentes a través de la interfaz de la clase `Componente`.

Cuando un componente recibe una petición, reacciona en función de su clase. Si el componente es una hoja, procesa la petición tal y como se ilustra en la figura 12.4.

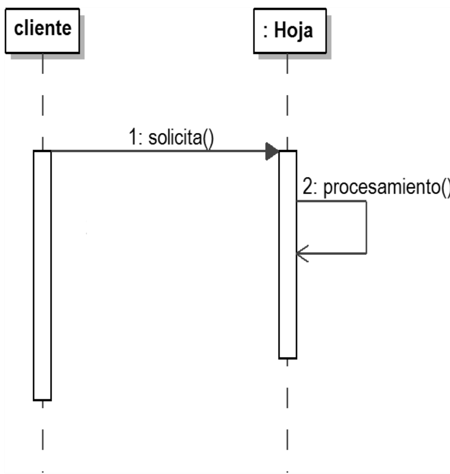

Figura 12.4 - Procesado de un mensaje por parte de una hoja

Si el componente es una instancia de la clase `Compuesto`, realiza un procesado previo, luego generalmente envía un mensaje a cada uno de sus componentes y realiza un procesamiento posterior. La figura 12.5 ilustra este comportamiento de llamada recursiva a otros componentes que van a procesar, a su vez, esta petición bien como hoja o bien como compuesto.

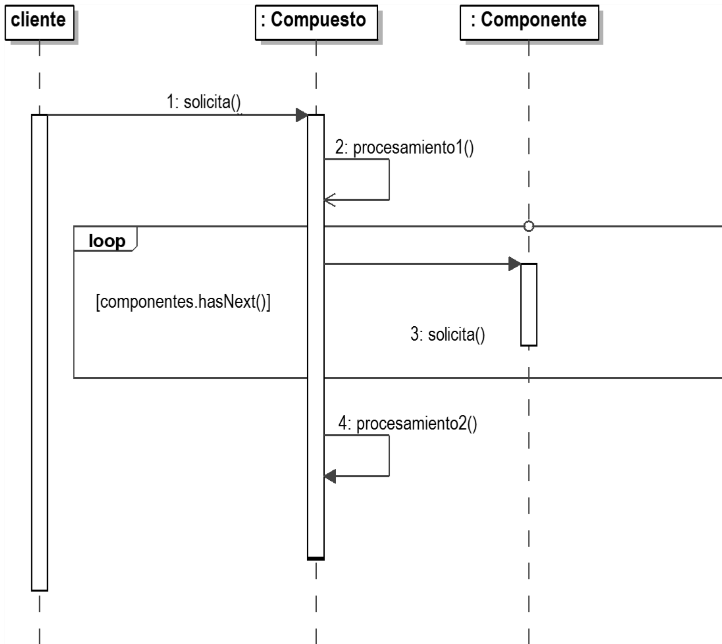

Figura 12.5 - Procesado de un mensaje por parte de un compuesto

4. Dominios de aplicación

El patrón de diseño Composite se utiliza en los siguientes casos:

– Es necesario representar jerarquías de composición en un sistema.

– Los clientes de una composición deben ignorar si se comunican con objetos compuestos o no.

© Editions ENI - All rights reserved

5. Ejemplo en PHP

Retomemos el ejemplo de las empresas y la gestión de su parque de vehículos.

El código fuente en PHP de la clase abstracta `AbstractEmpresa` se describe a continuación. Conviene observar que el método `agregaFilial` devuelve un resultado booleano que indica si ha sido posible realizar o no la agregación.

```
<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\Composite;

abstract class AbstractEmpresa
{
   protected static float $costeUnitarioVehiculo = 5;

   protected int $numVehiculos = 0;

   public function agregaVehiculo(): void
   {
       $this->numVehiculos++;
   }

   abstract public function calculaCosteMantenimiento(): float;

   abstract public function agregaFilial(AbstractEmpresa
$filial): bool;
}
?>
```

El código fuente de la clase `EmpresaSinFilial` aparece a continuación. Lógicamente, las instancias de esta clase no pueden agregar filiales.

```
<?php

declare(strict_types=1);

namespace ENI\DesignPatterns\Composite;

class EmpresaSinFilial extends AbstractEmpresa
{
   public function agregaFilial(AbstractEmpresa $filial): bool
```