Programación orientada a objetos en PHP

Fran Iglesias

Programación orientada a objetos en PHP

Fran Iglesias

This book is for sale at http://leanpub.com/programaobjetosphp

This version was published on 2016-02-05

Índice general

El objetivo de este libro es...

Qué son los objetos y cómo se crean

La respuesta corta

En programación, un objeto es una unidad que encapsula estado y comportamiento.

La respuesta larga

Cuando escribes un programa lo que haces es diseñar un modelo formal de una actividad o proceso. En esta actividad pueden manejarse ciertas entidades que, a medida que son sometidas a ciertos procedimientos, van cambiando su estado.

Eso que llamamos estado se concreta en una serie de propiedades o variables. Aunque los objetos modelan entidades o seres del mundo real eso no quiere decir que tengamos en cuenta todas sus propiedades, sino sólo las que son relevantes para los procesos que estamos modelando. Por ejemplo, en un sistema de gestión de alumnado para un colegio representaremos a los alumnos teniendo en cuenta aspectos como su nombre, su dirección postal, su fecha de nacimiento, etc, pero no tomaremos su estatura o peso y otras muchas de sus propiedades. Sin embargo, en un sistema para gestión de una consulta de pediatría sí que necesitamos llevar cuenta de su estatura y peso, entre otras variables.

Los procedimientos en que manipulamos las entidades modeladas determinan el comportamiento de los objetos. Volviendo a los ejemplos anteriores, en el sistema de gestión de alumnado nos interesan comportamientos como realizar un examen, recibir una calificación, llegar tarde, etc. En el sistema médico, por su parte, modelamos procesos como realizar una vacunación, sufrir una enfermedad, asignar y seguir un tratamiento, y un largo etcétera.

La discusión sobre qué entidades modelar, qué propiedades deben tener y qué comportamientos deben ejecutar, consituyen el meollo del diseño del sistema orientado a objetos.

Veamos un ejemplo.

Supongamos una biblioteca. En la biblioteca se gestionan libros, lo que implica adquirirlos, catalogarlos, prestarlos, devolverlos, enviarlos a restaurar, etc.

Cada libro es un objeto y tiene unas propiedades que lo definen, como el título o el autor (entre otras muchas), pero también estarían incluídas otras como si está disponible para préstamo, si está prestado o si ha sido retirado temporalmente para restauración. El conjunto de estas propiedades en un momento dado es el estado.

Por otra parte, con cada libro se dan una serie de procesos. Los libros son adquiridos por la biblioteca y catalogados. Una vez hecho esto están disponibles para el préstamo, así que los libros pueden ser prestados y devueltos. O bien, si el libro está en mal estado, debe ser enviado al restaurador.

En un programa representaríamos los libros con objetos que estarían constituidos de estado y comportamiento. El estado vendría representado por las propiedades del objeto, mientras que el comportamiento vendría definido por los métodos.

Representando el mundo

Para representar un libro podríamos optar por varias estrategias en PHP.

```
<?php

$aBookTitle = 'Don Qujote';
$aBookAuthor = 'Miguel de Cervantes';
$aBookAvailable = true;

echo "El libro se titula ".$aBookTitle." y está escrito por ".$aBookAuthor;
?>
```

Incluso los más novatos en PHP se darán cuenta de que ésta es una forma bastante ineficaz de trabajar. Necesitamos definir tres variables diferentes para poder trabajar con un libro. Tenemos que acordarnos siempre de las tres y no hay manera de obligarnos a mantenerlas juntas. Si volvemos a ver nuestro código después de unos días pasaremos un buen tiempo intentando entender cómo funciona ésto.

Afortundamante PHP ofrece una estructura de datos que podría encajar muy bien aquí: los arryas asociativos.

```
<?php

$aBook = array(
    'title' => 'Don Quijote',
    'author' => 'Miguel de Cervantes',
    'available' => true
);

echo "El libro se titula ".$aBook['title']." y está escrito por ".$aBook['author\
'];
?>
```

Mucho mejor. Ahora sólo tenemos una variable y es fácil entender la organización de datos y sus relaciones. Incluso el acceso a cada propiedad del libro es sencillo.

Pero podemos ir mucho más allá. Con objetos no sólo mantenemos las propiedades juntas y describimos entidades del mundo real como conjuntos unitarios, sino que incluso podemos hacer que actúen, que tengan comportamiento y que hagan cosas.

Objetos y propiedades

Pero vayamos por partes: las propiedades del objeto son como variables definidas dentro del ámbito del objeto y pueden adoptar cualquier tipo de valor soportado por el lenguaje de programación, incluyendo otros objetos.

En principio esto podría definirse en PHP así:

```
1  <?php
2      class Book {
3          var $title;
4          var $author;
5          var $available;
6      }
7  ?>
```

Bien. Este código, como cualquier otro, es discutible, pero debería dejarnos claras algunas cosas y también plantearnos algunas preguntas.

Por ejemplo, ¿a qué viene eso de *class*?

Los objetos no salen de la nada, tienen que definirse en alguna parte. Para ello escribimos clases que son definiciones de la estructura de los objetos. En un programa utilizamos instancias de esa clase que hemos definido.

Para declarar una clase, utilizamos la palabra clave `class` y el nombre del objeto. Veremos más adelante que esta declaración puede tener algunos modificadores interesantes.

El cuerpo del objeto va entre llaves `{}`, como es habitual en la definición de bloques en PHP. Dentro del cuerpo definimos propiedades y definimos métodos.

Las propiedades del libro, que describen su estado, llevan la palabra clave `var` y su nombre. es posible no declarar las propiedades del objeto, ya que PHP es un lenguaje tipado dinámicamente, pero es una mala práctica. Lo mejor es declarar explícitamente las propiedades de la clase.

Cómo se usa un objeto

Para usar un objeto debemos generar una instancia de la clase que deseemos. Así, por ejemplo, si queremos usar un libro en nuestro programa lo haremos así:

```php
<?php

class Book {
    var $title;
    var $author;
    var $available;
}

$aBook = new Book();

?>
```

Este código muestra cómo instanciar un objeto de la clase Book.

Instanciar es crear una variable del tipo de objeto deseado. La clave `new` le dice a PHP que cree un objeto de la clase Book en memoria y lo asocie a la variable `$aBook`.

¿Qué pasaría si añadimos una línea al código anterior y creamos otra instancia de Book?

```php
<?php

class Book {
    var $title;
    var $author;
    var $available;
}

$aBook = new Book();
$otherBook = new Book();

?>
```

$otherBook es un objeto de la clase Book, o sea, de la misma clase que $aBook, pero es un objeto distinto. Ocupa diferente espacio en memoria. Aunque se usasen los mismos dato representan dos libros físicamente distintos, de la misma forma en que una biblioteca puede tener varios ejemplares de un mismo título.

Ahora bien. ¿Cómo asignamos valores a las propiedades del objeto?

Pues lo haríamos así:

```php
<?php

class Book {
    var $title;
    var $author;
    var $available;
}

$aBook = new Book();
$aBook->title = 'Don Quijote';
$aBook->author = 'Miguel de Cervantes';
$aBook->available = true;

echo "El libro se titula ".$aBook->title." y está escrito por ".$aBook->author;

?>
```

Por defecto, las propiedades del objeto declaradas con la palabra clave `var` son públicas, es decir, accesibles desde fuera del propio objeto y podemos asignarles valores y leerlos como se muestra en el código anterior.

¿Y esto de qué sirve?

Bien. Ya sabemos definir una clase con sus propiedades e instanciar un objeto a partir de la clase definida, pero no tenemos comportamientos ni parece que el objeto vaya a ser muy útil tal cual está, aparte de poder almacenar algunos datos.

En realidad, este tipo de objeto incluso tiene un nombre. Se trata de un **Data Transport Object** (Objeto de transporte de datos) o DTO. Resulta muy útil cuando necesitamos mover datos relacionados de una manera cómoda y rápida. Podríamos argumentar que un array asociativo hace lo mismo, pero pronto veremos que los objetos tienen varias ventajas.

Visibilidad de las propiedades

Acabo de mencionar que las propiedades del objeto que acabamos de definir son públicas. Sin embargo, esto debería chirriarnos un poco. La programación orientada a objetos trata en gran medida de *ocultar la información* y la estructura y funcionamiento interno de los objetos. Estos son cajas negras para el resto del programa que sólo debería ver algunos métodos públicos.

Es como cuando contratas los servicios de cualquier profesional. Supongamos que se estropea tu nevera. Llamas al servicio técnico y una persona se desplaza a tu casa y la repara. Tú no tienes ni idea de lo que hace o cómo lo hace, simplemente le pides que repare tu nevera, le explicas los

síntomas en los que notas que no funciona bien y esperas que te de una respuesta, la cual puede ser que tu nevera vuelve a funcionar correctamente o bien que ya no se puede reparar.

Como norma general, las propiedades de los objetos deben declararse privadas.

Esto se hace sustituyendo la declaración `var` por `private`.

```
1  <?php
2
3  class Book {
4      private $title;
5      private $author;
6      private $available;
7  }
8
9  $aBook = new Book();
10
11 // A partir de aquí tendremos error
12
13 $aBook->title = 'Don Quijote';
14 $aBook->author = 'Miguel de Cervantes';
15 $aBook->available = true;
16
17 echo "El libro se titula ".$aBook->title." y está escrito por ".$aBook->author;
18
19 ?>
```

Nuestro nuevo código nos va a dar un error porque estaremos intentando acceder a una propiedad privada del objeto $aBook. Las propiedades privadas sólo están accesibles dentro del objeto en que están definidas. Pero entonces no tenemos modo de asignar ni obtener sus valores. Necesitamos métodos para ello. Los veremos dentro de un momento.

¿Por qué es tan importante ocultar las propiedades de los objetos?

Veamos el caso de los libros. Los libros llegan a una biblioteca una vez publicados, por lo que el título, el autor y otros datos no cambiarán nunca. En nuestro código eso se refleja haciendo que esas propiedades sean *inmutables*, lo que se logra:

- declarándolas privadas, de modo que no sean accesibles al mundo exterior.
- asignándolas en el momento de la instanciación del objeto, a través del llamado método constructor, el cual veremos a continuación.

Por otro lado, examinemos la propiedad available. Esa propiedad tiene que cambiar según sea necesario para indicar si el libro está disponible para préstamo o no. Sin embargo, puede que

necesitemos realizar ciertas comprobaciones antes de cambiar su valor para asegurarnos de que realmente el libro está o no disponible. Además, si la propiedad es pública corremos el riesgo de que otra parte del programa la cambie de manera arbitraria, sin que se realicen las comprobaciones necesarias llevando a nuestro objeto libro a un estado inconsistente, como podría ser tenerlo disponible para préstamo cuando está siendo restaurado, por ejemplo.

El constructor __construct()

El primer método que vamos a escribir es un constructor, que en PHP se declara con el nombre reservado `__construct` y que se invoca automáticamente cuando instanciamos un objeto con `new()`.

Los métodos de una clase se declaran igual que las funciones, con la diferencia de que se hace dentro del bloque de declaración de la clase.

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct() {
    }
}

?>
```

Al igual que las funciones, podemos indicar argumentos en la signatura de la función, los cuales se pueden utilizar dentro del método. En nuestro caso, queremos pasar el título y el autor del libro, por lo que podemos escribir el siguiente código:

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($title, $author) {
        $this->title = $title;
        $this->author = $author;
    }
```

```
12  }
13
14  ?>
```

Lo más llamativo de este método es la partícula `$this->`. Esta partícula indica que nos referimos a la propiedad con ese nombre del objeto. También usaremos `$this` para referirnos a los métodos. `$this` viene a significar "la instancia actual de la clase".

Volviendo al método, simplemente le pasamos los parámetros `$title` y `$author` y asignamos sus valores a las propiedades correspondientes. No hay ninguna razón técnica para que tengan los mismos nombres, pero preferimos hacerlo así por legibilidad. También podrías adopatar otra convención si crees que esta forma resulta ambígua. De paso, veremos cómo instanciar un objeto de la clase Book.

```
1   <?php
2
3   class Book {
4       private $title;
5       private $author;
6       private $available;
7
8       function __construct($aTitle, $anAuthor) {
9           $this->title = $aTitle;
10          $this->author = $anAuthor;
11      }
12  }
13
14  $aBook = new Book('El Quijote', 'Miguel de Cervantes');
15
16  ?>
```

Aparte de cambiar el nombre de los parámetros, hemos utilizado `new` para instanciar el objeto `$aBook`, que es de la clase Book. Para instanciar más objetos, o sea para tener más libros, usaríamos new pasándole los datos de los nuevos libros.

Obetener el valor de propiedades privadas: getters

Ahora que nuestro libro ya puede tener título y autor se nos plantea el problema de acceder a esos valores. Es decir, hemos aprendido a asignar valores a las propiedades durante la construcción del objeto, pero ¿cómo accedo a esos valores si necesito consultarlos más adelante?

Para ello debemos escribir métodos que nos los devuelvan. A este tipo de métodos se les suele llamar *getters*: Veamos un ejemplo:

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthor) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
    }

    public function getTitle() {
        return $this->title;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');
echo $aBook->getTitle();

?>
```

El método `getTitle` nos permite obtener el contenido de la propiedad `$title` del objeto. Como puedes ver las visibilidad del objeto es pública, para que pueda ser usado por nuestro programa y se limita a devolver el valor mediante return.

Obviamente un método puede ser más complejo y realizar operaciones diversas para generar un resultado determinado.

Algunos IDE permiten generar automáticamente métodos get* para todas las propiedades de la clase. Sin embargo, hay muy buenas razones para no hacerlo. En realidad, sólo deberíamos crear métodos get* para aquellas propiedades que queremos que se puedan consultar externamente.

En nuestro caso, puede que nos interese poder acceder al título para generar listados de los libros utilizados por un lector determinado. Podría ser incluso que nunca necesitemos un método que nos devuelva sólo el autor, sino una cadena que combine título y autor. Todo esto depende, obviamente, de los casos de uso que queremos cubrir con nuestra aplicación. Veamos un ejemplo:

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthor) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
    }

    public function getTitle() {
        return $this->title;
    }

    public function getAsReference() {
        return $this->author.', '.$this->title;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');
echo $aBook->getAsReference();

?>
```

Una nota sobre los nombres de los métodos: deberían revelar las intenciones, o sea, el nombre del método debería indicar qué hace el método y qué podemos esperar de él. El nombre del método no debe reflejar el cómo lo hace.

Propiedades que cambian

Hemos dejado sin tocar la propiedad `$available` para poder centrarnos en ella un momento. Con éste código podemos "construir" libros que tienen título y autor. Además, como no hay otros métodos que nos permitan acceder a esas propiedades, estos libros son "inmutables" al respecto de las mismas. Sin embargo, `$available` no queda definida (ahora mismo contiene `null`) por lo que tenemos un problema. ¿No sería mejor establecerla también en el constructor? La respuesta es un rotundo sí.

El estado inicial del libro debería quedar correctamente establecido en el constructor. Esto es, al ejecutar el constructor, el objeto tiene que tener un estado válido, quedando todas las propiedades relevantes iniciadas a valores que sean válidos y con sentido para los propósitos de ese tipo de objetos. Podría haber propiedades no incializadas si eso tiene un significado o un sentido en la vida del objeto.

Por ejemplo: la fecha del último préstamo, si consideramos que la clase Book debe gestionarla de algún modo, podría quedar definida aquí como `null` porque el hecho de que no esté definida indica que no ha sido prestada y no tiene sentido definirla hasta que el libro sea prestado.

Pero la propiedad `$available` sí necesita estar definida. Cuando un libro se añade al catálogo de la biblioteca, ¿debe estar ya disponible o todavía no? Esta es una decisión que hay que tomar para el caso que estamos trabajando: puede que tenga que estar disponible ya, puede que no porque hay que realizar otras operaciones o puede que haya que decidirlo en el momento de añadir el libro al catálogo. Veamos eso en código:

En este ejemplo se ha decidido que todos los libros están disponibles nada más darlos de alta:

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthor) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
        $this->available = true;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');

?>
```

Por otra parte, aquí se ha decidido lo contrario, y los libros no están disponibles hasta una decisión posterior:

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthor) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
```

```
        $this->available = false;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');

?>
```

Por último, en este código la decisión se toma en el momento de dar de alta el libro:

```
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthor, $isAvailable) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
        $this->available = $isAvailable;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');

?>
```

Como ya supondrás, la propiedad –$available– cambiará durante la vida del libro a medida que este sea prestado y devuelto. Tendremos que escribir un método para eso.

Asignar valor a propiedades: setters.

Una primera forma de afrontar la cuestión es crear métodos que nos permitan asignar valores a propiedades. A los métodos cuyo propósito es asignar valores a una propiedad específica los llamamos *setters*. Para asignar valores a `$available`, podríamos hacer lo siguiente:

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthore) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
        $this->available = false;
    }

    public function setAvailable($available) {
        $this->available = $available;
    }

    public function getAvailable() {
        return $this->available;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');
$aBook->setAvailable(true);
?>
```

Bien, ya tenemos un método para asignar un valor a –$available– y, de paso, hemos creado un método getter para poder consultarlo. Ahora bien, ¿qué ventaja presenta esto sobre hacer pública la propiedad `$available`? Pues la verdad es que ninguna. Estamos dejando que sea un factor externo el que controle el valor de la propiedad y hacerlo mediante getters y setters o mediante propiedades públicas es más o menos lo mismo.

Este tipo de planteamiento se suele conocer como **Anemic Domain Model**, o Modelo de Dominio Anémico (Fowler): un objeto sin comportamiento significativo que sólo tiene métodos para asignar o leer valores.

Planteémoslo de otra forma: ¿qué es lo que hace que un libro esté disponible o no? Pues un libro estará disponible siempre que:

- no esté prestado.
- no se haya retirado por restauracion.

Por lo tanto, al respecto de la propiedad –$available–, ¿qué acciones son importantes para nuestro libro?

- préstamo
- devolución
- restauración
- reposición

En el mundo real, el hecho de prestar un libro hace que no se encuentre disponible para nuevos préstamos ni, de hecho, para ninguna otra acción (salvo tal vez una reserva de préstamo, pero no nos vamos a ocupar de eso ahora). Cuando nos devuelven el libro, éste vuelve a estar disponible. Podemos reflejarlo así:

```php
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthor) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
        $this->available = false;
    }

    public function lend() {
        if ($this->isAvailable()) {
            $this->available = false;
        }
    }

    public function getBack() {
        $this->available = true;
    }

    private function isAvailable() {
        return $this->available;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');
$aBook->lend();
if($aBook->isAvailable()) {
    echo $aBook->title.' está disponible.'}
```

```
else {
    echo $aBook->title.' está prestado o en restauración.'
}
?>
```

Hemos escrito métodos para prestar, devolver y conocer el estado de disponibilidad de un libro. Estos métodos trabajan todos con la propiedad `$available`, pero ahora es el propio objeto libro el responsable de ella. Los agentes externos usan el libro con acciones significativas, como lend (prestar), getBack (devolver) o isAvailable (preguntar si está disponible).

Aunque en el fondo actúan como setters y getters, su nombre nos indica "algo más". Nos indica qué hace el método de una manera significativa para nuestra aplicación.

Analicemos ahora cada método, antes de realizar unos cambios que los hagan aún mejores.

El método lend

El método lend hace primero una comprobación: si el libro está disponible, entonces lo puede prestar. Al prestarlo tiene que poner la proviedad $available a false para indicar que ya no está disponible. Otra parte del programa puede ocuparse de recoger qué usuario se lleva el libro y cuál es el plazo de devolución. De momento, lo que necesitamos es manipular su disponiblidad.

Si el libro no está disponible no hace nada. Esto es insuficiente porque nosotros querríamos que el bibliotecario nos diga que no nos puede prestar el libro porque no está disponible, no queremos que simplemente no nos diga nada.

Una forma de hacer esto es lanzar una excepción. Hablaremos sobre las excepciones más adelante, pero de momento, si no sabes lo que son te bastará con saber que son errores que podemos "capturar" en un bloque try-catch para decidir cómo actuar en caso de que se produzcan.

```
<?php

class Book {
    private $title;
    private $author;
    private $available;

    function __construct($aTitle, $anAuthore) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
        $this->available = false;
    }

    public function lend() {
```

```
15          if (!$this->isAvailable()) {
16              throw new Exception ('El libro '.$this->title.' está prestado o en r\
17  estauración');
18          }
19          $this->available = false;
20      }
21
22      public function getBack() {
23          $this->available = true;
24      }
25
26      public function isAvailable() {
27          return $this->available;
28      }
29  }
30
31  $aBook = new Book('El Quijote', 'Miguel de Cervantes');
32  try {
33      $aBook->lend();
34      echo $aBook->title.' acaba de ser prestado.';
35  } catch(Exception $e) {
36      echo $e->getMessage();
37  }
38  ?>
```

¿Qué está ocurriendo aquí? Una vez instanciado el libro lo intentamos prestar. Como en nuestro sistema el libro no está disponible nada más ser ingresado (ver el método __construct), el método lend arrojará una excepción porque el libro no está disponible. La excepción será capturada por el bloque catch, que muestra el mensaje de error de la misma.

El método getBack

El método getBack se limita a poner en true la propiedad $available.

El método isAvailable

El método isAvailable devuelve el estado de disponibilidad del libro, para lo cual simplemente devuelve el valor de la propiedad `$available`. Podríamos argumentar que no es muy diferente de getAvailable, pero no es así. Para empezar, el propio nombre del método es mucho más explícito:

- getAvailable significa “dame el valor de la propiedad $available”, y ya lo interpretaré yo,

- isAvailable significa "¿está este libro disponible?". No necesito interpreatar nada.

Otra parte de la cuestión es que el método isAvailable puede hacer más comprobaciones si fuese necesario, sin tener que cambiar su interfaz pública. Supongamos que la clase Book mantiene una propiedad ($refurbish) que nos indica si el libro está siendo restaurado:

```
<?php

class Book {
    private $title;
    private $author;
    private $available;
    private $refurb;

    function __construct($aTitle, $anAuthore) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
        $this->available = false;
        $this->refurb = false;
    }

    public function lend() {
        if (!$this->isAvailable()) {
            throw new Exception ('El libro '.$this->title.' está prestado o en r\
estauración');
        }
        $this->available = false;
    }

    public function getBack() {
        $this->available = true;
    }

    private function isAvailable() {
        return $this->available && !$this->refurb;
    }
}

$aBook = new Book('El Quijote', 'Miguel de Cervantes');
try {
    $aBook->lend();
    echo $aBook->title.' acaba de ser prestado.';
```

```
} catch(Exception $e) {
    echo $e->getMessage();
}
?>
```

Si observas el código anterior verás que no hemos tenido que tocar nada en lo que respecta a usar el objeto $aBook, todos los cambios han sido dentro de la definición de la clase y, específicamente, dentro del método isAvailable que ahora toma en consideración el estado de $refurbish para responder.

Pero el código "cliente", el código que usa la clase Book no ha tenido que cambiarse para nada, es exactamente el mismo que antes. Ese es un ejemplo de cómo utilizar la encapsulación para nuestro beneficio, ya que la clase Book puede evolucionar sin necesidad de tocar el código que ya la utiliza.

Te habrás fijado que `isAvailable()` está definido como método privado. De momento, al escribir este código pensamos que ningún agente externo va a preguntar directamente si el libro está disponible, sencillamente solicitará el préstamo y se le responderá a eso. Sin embargo, en otro contexto podría ser necesario hacer público ese método. De nuevo, la visibilidad de los métodos es una cuestión de necesidades en el contexto de tu aplicación específica.

En muchos casos es una buena práctica comenzar con métodos privados y hacerlos visibles sólo si se descubre que es realmente necesario.

Añadamos un par de métodos para manejar la posibilidad de enviar un libro a restaurar.

```
<?php

class Book {
    private $title;
    private $author;
    private $available;
    private $refurb;

    function __construct($aTitle, $anAuthore) {
        $this->title = $aTitle;
        $this->author = $anAuthor;
        $this->available = false;
        $this->refurb = false;
    }

    public function lend() {
        if (!$this->isAvailable()) {
            throw new Exception ('El libro '.$this->title.' está prestado o en r\
estauración');
        }
```

```
21          $this->available = false;
22      }
23
24      public function refurb() {
25          if (!$this->isAvailable()){
26              throw new Exception ('El libro '.$this->title.' está prestado o en \
27  restauración');
28          }
29          $this->refurb = true;
30      }
31
32      public function getBack() {
33          $this->available = true;
34      }
35
36      public function getBackAfterRefurb() {
37          $this->refurb = false;
38      }
39
40      public function isAvailable() {
41          return $this->available && !$this->refurb;
42      }
43  }
44
45  $aBook = new Book('El Quijote', 'Miguel de Cervantes');
46  $aBook->refurb();
47  try {
48      $aBook->lend();
49      echo $aBook->title.' acaba de ser prestado.';
50  } catch(Exception $e) {
51      echo $e->getMessage();
52  }
53  ?>
```

Como puedes comprobar, de nuevo hemos podido hacer evolucionar la clase sin modificar el código cliente existente, salvo en el hecho de haber añadido un paso para utilizar la nueva funcionalidad. Todo gracias a diseñar los métodos de nuestra clase a partir de comportamientos útiles para nuestro sistema.

Polimorfismo y extensibilidad de objetos

En Programación Orientada a Objetos el **polimorfismo** es una característica que nos permite enviar el mismo mensaje, desde el punto de vista sintáctico, a distintos objetos para que cada uno de ellos lo realice con un comportamiento específico.

En la práctica, eso signfica que si disponemos de varios objetos que tienen un mismo método con la misma signatura podemos usar el mismo código para invocarlos, sin necesidad de preguntarles por su clase previamente.

El ejemplo clásico es el de un programa de dibujo en el que se manejan distintos tipos de formas. Cada forma se define mediante una clase con un método para dibujarla (por ejemplo, `draw`) que cada una ejecuta de distinta manera. Se recorre la lista de objetos activos y se va pidiendo a cada uno que se dibuje sin que tengamos que preocuparnos de saber de antemando de qué clase es.

Otro ejemplo: todos los empleados de una empresa son de la clase "Emppleados", pero pueden desempeñar distintos puestos, que definen responsabilidades, salarios, beneficions, etc. Cuando les pedimos que "trabajen" cada uno hace lo que le corresponde.

Algunos autores sobre Programación Orientada a Objetos dicen que cuando hacemos una llamada a un método de un objeto le enviamos un mensaje para que se ponga a ejecutar alguna tarea, de ahí nuestra primera definición. Al decir que es el mismo mensaje desde el punto de vista sintáctico queremos decir que es una llamada a métodos con el mismo nombre que cada objeto puede interpretar de manera más o menos diferente (punto de vista semántico).

Siguiendo con nuestro ejemplo de la biblioteca, supongamos que tenemos las clases Book y Review y que ambas definen el método `lend()` para indicar que son prestadas. Entonces es posible escribir un código similar a éste:

```
1  <?php
2
3  $objects  =
4      array(
5          new Book('Programar en PHP', 'Pepe Pérez'),
6          new Review('PHP monthly review', 'Springer', 'May 2014')
7      );
8
9  foreach($objects as $theObject) {
10     $theObject->lend();
11 }
```

```
12
13 ?>
```

Sencillamente, tenemos una lista de objetos que quire retirar un lector. Vamos recorriendo la colección uno por uno invocando el método `lend`, y no tenemos que comprobar su tipo, simplemente les decimos que "sean prestados". Aunque los objetos son distintos ambos pueden responder al mismo mensaje.

Hasta cierto punto, para los programadores en PHP esto no debería resultar muy sorprendente, pero los programadores de otros lenguajes se removerían inquietos en sus asientos, así que hay que hacer un intermedio para hablar un momento acerca de los tipos de datos en los lenguajes.

Tipado estricto vs tipado débil

El sistema de tipado de PHP permite que podamos reutilizar una misma variable para distintos tipos de datos. Es decir, comenzamos usando una variable de tipo integer, pero posteriormente podríamos introducir un dato de tipo string. El intérprete se encarga de, sobre la marcha, decidir el tipo de dato que puede manejar y, si es necesario, lo convierte. PHP es lo que se conoce como un lenguaje de **tipado débil**.

Otros lenguajes tienen **tipado estricto**. Esto quiere decir que las variables deben declararse con su tipo para que el intérprete o compilador reserve el espacio de memoria adecuado. Además, una vez definida una variable, ésta sólo puede usarse para almacenar valores de ese tipo o que hayan sido explícitamente convertidos a ese tipo. De este modo, si intentas guardar un string en una variable integer, saltará un error.

Por esa razón, el polimorfismo en los lenguajes de tipado estricto resulta especialmente llamativo, ya que con el tipado estricto no podríamos declarar una variable de tipo Book para empezar y luego asignarle un valor de tipo Review como se hace en el código anterior. El polimorfismo, por tanto, implica que el sistema de tipos se relaja.

Por otro lado, en los lenguajes de tipado estricto también deben declararse los tipos de los argumentos de las funciones y, consecuentemente, los tipos de los argumentos de los métodos. Esto es muy bueno, porque nos proporciona un primer medio de control de que los datos que se pasan son válidos. Si intentamos pasar un argumento de un tipo no indicado para la función el intérprete o compilador nos lanzará un error.

Precisamente, PHP ha incorporado esa caracterísica, denominada *Type Hinting* en las llamadas a funciones y métodos pero sólo para objetos y, gracias a ello, podemos declarar el tipo de objeto que pasamos como argumento a una función o método. Su uso no es obligatorio, pero como beneficio directo tenemos esa primera línea de validación de datos que, más adelante, veremos como explotar a fondo.

Sin embargo, ¿cómo es posible que el tipado fuerte o, en su caso, el type hinting permitan el polimorfismo?

Pues a través de mecanismos que permiten que una clase pueda responder como si, de hecho, fuese de varios tipos simultáneamente. Estos mecanismos son la herencia y las interfaces.

A través de la **herencia** podemos derivar unas clases a partir de otras, de modo que las clases "hijas" también son del mismo tipo que sus clases padres y, de hecho, que todos sus ascendientes.

Las **interfaces** nos permiten que las clases "se comprometan" a cumplir ciertas condiciones asumiendo ser del tipo definido por la interfaz. Es habitual referirse a esto como un contrato. El cumplimiento del mismo es asegurado por el intérprete de PHP que lanza un error si la clase no implementa correctamente la interfaz.

En ambos casos nos queda garantizado que los objetos instanciados serán capaces de responder a ciertos métodos, que es lo que nos interesa desde el punto de vista del código cliente, aparte de respetar la integridad del sistema de tipado.

En el caso de la herencia, se consigue por el hecho de que las clases bases ya ofrecen esos métodos incluso aunque no estén definidos en las clases hijas. En el caso de las interfaces, el contrato obliga a las clases a implementar esos métodos. El resultado es que el código cliente puede confiar en que puede esperar ciertos comportamientos de los objetos que utiliza.

Herencia

La herencia es un mecanismo para crear clases extendiendo otras clases base. La nueva clase hereda los métodos y propiedades de la clase base a la que extiende, así que ya cuenta con una funcionalidad de serie.

Normalmente utilizaremos la herencia para crear especializaciones de una clase más general. Es decir, no extendemos una clase para conseguir que la nueva contenga métodos de la clase extendida, sino que lo hacemos porque la nueva clase es semánticamente equivalente a la clase base pero más específica o concreta. Por ejemplo, podemos tener una clase que gestiona la conexión genérica a una base de datos y extenderla con varias clases hijas para manejar drivers específicos (MySQL, SQLite, Postgre, MongoDB, etc.).

La nueva clase puede tener nuevos métodos y propiedades, o bien puede sobreescribir los existentes, sesgún sea necsario. Como resultado tenemos una clase nueva, que además de por su propio tipo, puede ser identificada por el mismo que su "clase madre" aunque se comporte de manera diferente.

Retomamos la reflexión sobre el sistema de biblioteca. Resulta que ella no sólo hay libros, sino revistas, discos, películas y otros medios disponibles para los usuarios. Podríamos crear clases para cada tipo, pero ¿cómo asegurárnos de que todas van a poder responder a los métodos necesarios?. Examinando la cuestión nos damos cuenta de que todas ellas podrían ser "hijas" de una clase más general, una "superclase", que vamos a llamar "Media". De esta manera los libros serían Media, así como las revistas, los discos o las películas.

La clase Media podría aportar algunos métodos y propiedades comunes, y cada clase descendiente realizar algunas cosas de manera un poco diferente. Por ejemplo, los libros se suelen identificar con autor y título, las revistas por título y número, las películas por título, etc.

Un ejemplo de otro ámbito es el de los vehículos. La clase de los vehículos engloba motocicletas, turismos, furgonetas, camiones, autobuses, etc. Existen ciertas propiedades comunes que pueden adoptar valores diferentes. Todos los vehículos tienen ruedas: algunos dos, otros cuatro, otros más. Todos tienen motor, que puede ser de diferentes cilindradas y potencias. Todos tienen un número de plazas además del conductor o tienen una capacidad de transporte de mercancía. Incluso dentro de cada subtipo de vehículo podríamos establecer más especializaciones, creando una jerarquía de clases cuya base es Vehículo y, de hecho, todas las subclases siguen siendo igualmente Vehículos. O lo que es lo mismo, cada clase es, a la vez, esa clase y todos su ancestros hasta la clase base.

Con todo, las jerarquías de clases no deberían ser muy largas o profundas.

Por otro lado, las jerarquías de clases deberían cumplir el principio de sustitución de Liskov, que es uno de los principios SOLID, y que dice que las clases de una jerarquía deberían ser intercambiables.

Interfaces

La herencia como forma de extender objetos tiene varios problemas y limitaciones. Al prinicpio parece una prestación espectacular, pero intentar resolver todas las necesidades de polimorfismo con Herencia nos lleva a meternos en muchos líos.

Por ejemplo, podemos empezar a desear que una clase pueda heredar de varias otras simultáneamente. O también podríamos querer que una clase descienda de otra determinada sólo para poder pasarla a un método y que supere el type hinting. Este uso de la herencia acaba violando varios de los principios SOLID, en especial el de Liskov y el de segregación de interfaces.

Para lograr eso respetando los principios SOLID tenemos las interfaces.

Las interfaces son declaraciones similares a las clases que sólo contienen signaturas de métodos públicos, sin ningún código ni propiedades, aunque pueden definir constantes. Las clases que implementan la interfaz se comprometen a implementar esos métodos respetando la signatura definida. En otras palabras: es un contrato de obligado cumplimiento.

Una clase puede implementar varias interfaces, lo que respondería a esa necesidad de la herencia múltiple. Y, por otra parte, dos o más clases pueden implementar la misma interfaz aunque no tengan ninguna otra relación semántica entre ellas.

Por ejemplo, podríamos tener una interfaz para clases que puedan escribir en un log.

```php
<?php

interface Loggable {
        public function log($message);
        public function write();
}

class Emailer implements Loggable {

        private $messages;

        public function log($message)
        {
                $this->messages[] = $message;
        }

        public function write()
        {
                foreach ($this->messages as $message) {
                        // Write message to email.log file
                }
                $this->messages = array();
        }
}


class DataBase implements Loggable {

        private $messages;

        public function log($message)
        {
                $this->messages[] = $message;
        }

        public function write()
        {
                foreach ($this->messages as $message) {
                        // Write message to database.log file
                }
                $this->messages = array();
        }
```

```
43
44  }
45
46
47  class Logger
48  {
49          private $objects;
50
51          public function register(Loggable $object)
52          {
53                  $this->objects[] = $object;
54          }
55
56          public function write()
57          {
58                  foreach ($this->objects as $object) {
59                          $object->write();
60                  }
61          }
62
63  }
64
65  ?>
```

Todas las clases que quieran ser `Loggable` deberán implementar un método `log` que espera un parámetro `$message` y un método `write` que escribirá los mensajes existentes en un archivo. El archivo de log concreto en que se escribe es cosa de cada clase, pero al definirlo así las clases están obligados a implementar los métodos cada una según sus necesidades. A la vez, podemos ver que no hay otra vinculación entre las clases, que pueden evolucionar de forma completamente independiente.

Es más, en unos casos el objeto podría escribir el mensaje a un archivo, otro objeto podría enviar un email y otro objeto escribir en una base de datos. Sencillamente la interfaz declara que los objetos de tipo `Loggable` deben poder tomar nota de un mensaje (método `log`) y deben poder "escribirlo" posteriormente de acuerdo a sus propios propósitos o necesidades (método `write`).

Para indicar que una clase implementa una interfaz la declaramos asi:

```
1  class Example implemente Interface {}
2
3  class Emailer implements Loggable {}
```

Si se implementan varias interfaces ser haría así, por ejemplo, para que la clase Emailer implemente dos hipotéticas interfaces Loggable y Service:

```
class Emailer implements Loggable, Service {}
```

Por otra parte, en el código de ejemplo tenemos una clase `Logger` que puede registrar objetos de tipo `Loggable` y que se encargará de actualizar los logs en una sola pasada. Para ello, sabe que los objetos registrados tendrán un método `write()` para escribir sus mensajes, sin preocuparse para nada de cuántos mensajes o de qué archivo concreto debe ser modificado. De este modo, podemos registrar tantos objetos `Loggable` como nuestra aplicación necesite.

Puesto que `Logger` no tiene que preocuparse de la manera concreta en que los objetos `Loggable` registrados realizan el método `write` resulta que en una única llamada podemos tener escrituras en diferentes archivos de logs, a la vez que otros envían un email a un administrador y otros lo registran en una tabla de una base de datos.

En el ejemplo se puede ver que hay una duplicación de código entre las clases que implementan Loggable (es casi el mismo código) y eso podría hacernos plantear la pregunta: ¿y no podemos hacer lo mismo con herencia?. Sí y no. Bien, si estas clases se utilizasen en un proyecto real no sería difícil darse cuenta que tienen poco que ver una con la otra: una de ellas sería una clase para enviar emails y la otra para conectarse a una base de datos. Seguramente nos interesa que ambas tengan la capacidad de escribir logs para poder hacer un seguimiento de su actividad y detectar fallos, pero es no es motivo para que ambas hereden de una superclase “Loggable”. Por eso usamos interfaces, aunque supongan escribir el código aparentemente común.

Las últimas versiones de PHP dan soporte a traits, que son una forma de reutilizar código entre diferentes clases. Un trait nos permite definir métodos y propiedades que luego podríamos utilizar en diferentes clases. Sin embargo, no debemos confundir eso con la idea de las interfaces. EL hecho de que dos clases, como en el ejemplo, tengan algún método idéntico nos permitiría utilizar un trait para no duplicar código (el método `log` es un claro candidato en el ejemplo), pero ambas siguen teniendo que implementar la misma interfaz para beneficiarnos del polimorfismo.

Los principios SOLID

Los principios SOLID son cinco principios en el diseño orientado a objetos compilados por Robert C. Martin.

Se trata de cinco principios que nos proporcionan una guía para crear código que sea reutilizable y se pueda mantener con facilidad. Como tales principios, nos aportan unos criterios con los que evaluar nuestras decisiones de diseño y una guía para avanzar en el trabajo. El código que los respeta es más fácil de reutilizar y mantener que el código que los ignora. Obviamente, no tienen una forma completa de implementarse y, en cada caso, pueden tener una interpretación algo diferente. No son recetas concretas para aplicar ciegamente, sino que requieren de nosotros una reflexión profunda sobre el código que estamos escribiendo.

Los cinco principios son:

- Principio de única responsabilidad (Single Responsibility Principle)
- Principio abierto/cerrado (Open/Closed Principle)
- Principio de sustitución de Liskov (Liskov Substitution Principle)
- Principio de segragación de interfaces (Interface Segregation Principle)
- Principio de inversión de dependencias (Dependency Inversion Principle)

Si eres un desarrollador único o todo-en-uno en tu empresa o proyecto puedes pensar "para qué necesito que mi código sea mantenible o reutilizable si voy a ser yo quién se ocupe de él en el futuro y quién lleva ocupandose durante estos años y que nadie más va a verlo". La respuesta es sencilla: dentro de unos pocos meses vas a ser un programador distinto del que eres ahora. Cuando después de varias semanas trabajando en una parte de tu código te dirijas a otra parte a hacer una modificación tal vez acabes preguntándote qué estúpida razón te llevó a escribir esa función o a crear esa clase y no tendrás respuesta. Sólo por tu salud mental y laboral deberías escribir un código limpio, legible y mantenible. Seguramente tendrás por ahí secciones completas de código que no te atreves a tocar porque no sabes qué puede pasar si lo haces, o partes que se duplican en varios sitios, sabes que tienes código que apesta y que no te atreverías a enseñar. También por eso, por celo profesional, deberías aplicar los principios SOLID.

Los principios SOLID están muy relacionados entre sí, por lo que suele ocurrir que al intentar cumplir uno de ellos estás contribuyendo a cumplir otros.

Vamos a examinarlos en detalle.

Principio de única responsabilidad

Este principio dice que las clases deberían tener una única razón para cambiar.

Otra forma de decirlo es que las clases deberían hacer una sola cosa. El problema de esta definición, aparentemente más clara, es que puede estrechar demasiado tu visión: no se trata de que las clases tengan sólo un método o algo por el estilo.

En cualquier organización hay secciones o departamentos. Cada uno de ellos puede tener necesidades diferentes con respecto a un asunto. En un colegio, por ejemplo, el profesorado se preocupa de las calificaciones y asistencia de los alumnos, mientras que la administración le interesa saber qué servicios consume el alumno y por cuales debe facturarle, al servicio de comedor o cantina le interesa saber si el alumno va a comer ese día o no, y al servicio de transporte le interesa saber dónde vive. Al diseñar una aplicación para gestionar un colegio nos encontraremos con estas visiones y peticiones desde los distintos servicios.

Si más de un agente puede solicitar cambios en una de nuestras clases, eso es que la clase tiene muchas razones para cambiar y, por lo tanto, mantiene muchas responsabilidades. En consecuencia, será necesario repartir esas responsabilidades entre clases.

Principio abierto/cerrado

Este principio dice que una clase debe estar cerrada para modificación y abierta para extensión.

La programación orientada a objetos persigue la reutilización del código. No siempre es posible reutilizar el código directamente, porque las necesidades o las tecnologías subyacentes van cambiando, y nos vemos tentados a modificar ese código para adaptarlo a la nueva situación.

El principio abierto/cerrado nos dice que debemos evitar justamente eso y no tocar el código de las clases que ya está terminado. La razón es que si esas clases están siendo usadas en otra parte (del mismo proyecto o de otros) estaremos alterando su comportamiento y provocando efectos indeseados. En lugar de eso, usaríamos mecanismos de extensión, como la herencia o la composición, para utilizar esas clases a la vez que modificamos su comportamiento.

Cuando creamos nuevas clases es importante tener en cuenta este principio para facilitar su extensión en un futuro.

Principio de sustitución de Liskov

En una jerarquía de clases, las clases base y las subclases deben poder intercambiarse sin tener que alterar el código que las utiliza.

Esto no quiere decir que tengan que hacer exactamente lo mismo, sino que han de poder reemplazarse.

El reverso de este principio es que no debemos extender clases mediante herencia por el hecho de aprovechar código de las clases bases o por conseguir forzar que una clase sea una "hija de" y superar un *type hinting* si no existe una relación que justifique la herencia (ser clases con el mismo tipo de comportamiento, pero que lo realizan de manera diferente). En ese caso, es preferible basar el polimorfismo en una interfaz (ver el Principio de segregación de interfaces).

Principio de segregación de interfaces

El principio de segregación de interfaces puede definirse diciendo que una clase no debería verse obligada a depender de métodos o propiedades que no necesita.

Supongamos una clase que debe extender otra clase base. En realidad, nuestra clase sólo está interesada en dos métodos de la clase base, mientras que el resto de métodos no los necesita para nada. Si extendemos la clase base mediante herencia arrastramos un montón de métodos que nuestra clase no debería tener.

Otra forma de verlo es decir que las interfaces se han de definir a partir de las necesidades de la clase cliente.

Principio de inversión de dependencia

La inversión de dependencias no es sólo lo que tú piensas[1]

El principio de inversión de dependencia dice que:

- Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de detalles, son los detalles los que deben depender de abstracciones.

Las abstracciones definen conceptos que son estables en el tiempo, mientras que los detalles de implementación pueden cambiar con frecuencia. Una interfaz es una abstracción, pero una clase que la implemente de forma concreta es un detalle. Por tanto, cuando una clase necesita usar otra, debemos establecer la dependencia de una interfaz, o lo que es lo mismo, el type hinting indica una interfaz no una clase concreta. De ese modo, podremos cambiar la implementación (el detalle) cuando sea necesario sin tener que tocar la clase usuaria lo que, por cierto, contribuye a cumplir el principio abierto/cerrado.

[1] http://blog.koalite.com/2015/04/la-inversion-de-dependencias-no-es-solo-lo-que-tu-piensas/

Dependencias y acoplamiento

En programación decimos que se establece una **dependencia** cuando un módulo de software utiliza otro para realizar su trabajo. Si hablamos de clases, decimos que una clase (cliente) tiene un dependencia de otra (servicio) cuando cliente usa servicio para llevar a cabo sus propias responsabilidades. La dependencia se manifiesta porque la clase Cliente no puede funcionar sin la clase Servicio.

Las dependencias de software no son malas en sí mismas (tal y como se habla de ellas en algunos artículos parecería que sí). El problema de las dependencias es de grado. Es decir, hay dependencias muy fuertes y dependencias ligeras. La clave es cómo las gestionamos para que sean lo más flojas posibles.

De hecho, la existencia de dependencias es un buen indicio ya que podría indicar que estamos respetando el principio de Responsabilidad Única haciendo que unas clases deleguen en otras las tareas que no les corresponden.

Al grado de dependencia entre dos unidades de software la llamamos **acoplamiento**. Decimos que hay un alto acoplamiento (thigh coupling) cuando tenemos que reescribir la clase Cliente si quisiéramos cambiar la clase Servicio por otra. Esto es una violación del principio Abierto/Cerrado. Por el contrario, decimos que hay un bajo acoplamiento (loose coupling) cuando podemos cambiar la clase Servico por otra, sin tener que tocar a Cliente.

¿Cómo podemos hacer eso? Pues utilizando tres herramientas:

- El patrón de Inyección de dependencias
- La D en SOLID: el principio de inversión de dependencias.
- El patrón Adaptador

Dependencias ocultas

Comencemos con la peor situación posible: la clase Cliente utiliza a la clase Servicio sin que nadie lo pueda saber. Veamos un ejemplo:

```php
<?php

class Client {
        private $service;

        public function __construct() {
                $this->service = new Service();
        }
        public function doSomething() {
                $this->service->doTask();
        }
}

class Service {
        public function doTask() {
                echo 'Performed by Service';
        }
}


$client = new Client();
$client->doSomething();

?>
```

Para saber que Cliente utiliza servicio tendríamos que examinar su código fuente porque desde su interfaz pública no podemos ver nada.

Aquí se produce la violación del principio SOLID Abierto/Cerrado. Al estar así construída, la clase cliente está abierta a la modificación y para cambiar su comportamiento tenemos que reescribirla.

En este caso el acoplamiento es máximo y en el momento en que tuviésemos que tocar Servicio por algún motivo, la funcionalidad de Cliente podría romperse. Por ejemplo, supón que Servicio es una clase de un paquete o biblioteca y los autores deciden actualizarla y cambian la interfaz de los métodos que usa cliente. Por mucho que en PHP tengas acceso al código te puedes imaginar la pesadilla de mantenimiento y los riesgos que supone. De hecho, tendrías que quedarte en una versión fija de Servicio y olvidarte de las actualizaciones.

Para lidiar con este problema tienes una solución fácil y que es aplicar el patrón de inyección de dependencia:

```php
<?php

class Client {
        private $service;

        public function __construct(Service $service) {
                $this->service = $service;
        }
        public function doSomething() {
                $this->service->doTask();
        }
}

class Service {
        public function doTask() {
                echo 'Performed by Service';
        }
}


$client = new Client(new Service());
$client->doSomething();

?>
```

Así de simple. Se trata de cargar la dependencia a través del constructor (o de un Setter). Ahora la dependencia es visible. Todavía hay un alto acoplamiento, pero ya empezamos a tener más libertad pues sabemos cómo se relacionan ambas clases.

Inversión de dependencias

La inversión de dependencias es el camino que debemos seguir para reducir al máximo el acomplamiento entre dos clases o módulos. El principio de Inversión de Dependencias nos dice que:

- Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de detalles, son los detalles los que deben depender de abstracciones.

En resumen: cualquier dependencia debe ocurrir sobre abastracciones, no sobre implementraciones concretas.

En nuestro ejemplo, la dependencia es ahora explícita, lo que es bueno, pero Cliente sigue dependiendo de una implementación concreta de Servicio, lo que es malo.

Para invertir la dependencia debemos hacer lo siguiente:

Cliente no debe esperar una instancia concreta de Servicio, sino que debe esperar una clase que cumpla ciertas condiciones, o lo que es lo mismo, que respete un contrato. Y, como hemos visto, un contrato en programación es una interfaz. Y una interfaz es lo más abstracto de lo que podemos disponer en software.

Servicio, por su parte, debe respetar la interfaz para poder ser usada por Cliente, o sea, también debe depender de esa abstracción.

Así que necesitamos crear una interfaz y hacer que Cliente espere cualquier clase que la implemente. ¿Cómo puedo definir la interfaz? Pues a partir de las necesidades o intereses de Cliente, Servicio tendrá que adaptarse.

```php
<?php

interface ServiceInterface {
        public function doTheThing();
}

class Client {
        private $service;

        public function __construct(ServiceInterface $service) {
                $this->service = $service;
        }
        public function doSomething() {
                $this->service->doTheThing();
        }
}

class Service {
        public function doTask() {
                echo 'Performed by Service';
        }
}


$client = new Client(new Service());
$client->doSomething();

?>
```

El código mostrado no funcionará todavía: Service no implementa ServiceInterface por lo que Cliente no lo aceptará.

¿Por qué he cambiado el modo en que Cliente utiliza Servicio? Es decir, ¿por qué he cambiado el método que Cliente llama? Pues simplemente para ilustrar la necesidad de que la interfaz se escriba según las necesidades del cliente y también para mostrar cómo podemos hacer para Servicio cumpla la interfaz sin tener que tocar su código.

En el listado 3, Client depende de ServiceInterface, lo que significa que espera una clase que implementa un método `doTheThing()`. Sin embargo, Service no tiene ese método. Para resolverlo debemos o bien modificar la clase Service para implementar ServiceInterface o bien aplicar un patrón Adaptador, para utilizar la clase Service respetando ServiceInterface.

Un adaptador es una clase que implementa una interfaz usando otra clase, así que añadimos el adaptador a nuestro código:

```
1  <?php
2
3  interface ServiceInterface {
4          public function doTheThing();
5  }
6
7  class Client {
8          private $service;
9
10         public function __construct(ServiceInterface $service) {
11                 $this->service = $service;
12         }
13         public function doSomething() {
14                 $this->service->doTheThing();
15         }
16 }
17
18 // We don't touch Service
19
20 class Service {
21         public function doTask() {
22                 echo 'Performed by Service';
23         }
24 }
25
26 // We create an adpater
27
28 class ServiceAdapter implements ServiceInterface {
```

```
29          private $service;
30
31          public function __construct(Service $service)
32          {
33                  $this->service = $service;
34          }
35
36          public function doTheThing()
37          {
38                  $this->service->doTask();
39          }
40  }
41
42
43  $client = new Client(new ServiceAdapter(new Service()));
44  $client->doSomething();
45
46  ?>
```

La llamada se ha complicado un poco, pero los beneficios son enormes ya que hemos reducido el acoplamiento al mínimo posible:

A partir de ahora, las clases `Client` y `Service` pueden cambiar independientemente una de la otra con la condición de que la interfaz no cambie (y las interfaces están pensadas para ser estables en el tiempo salvo motivos muy justificados). Si fuese necesario tendríamos que modificar `ServiceAdapter` en el caso de que `Service` cambiase su interfaz.

En este ejemplo, imaginamos que Service ha sufrido un cambio que rompe la compatibilidad hacia atrás:

```
1   <?php
2
3   interface ServiceInterface {
4           public function doTheThing();
5   }
6
7   class Client {
8           private $service;
9
10          public function __construct(ServiceInterface $service) {
11                  $this->service = $service;
12          }
13          public function doSomething() {
```

```php
                $this->service->doTheThing();
        }
}


// Service has change its public interface

class Service {
        public function doService() {
                echo 'Performed by Service';
        }
}

// We change Adapter according to the changes in Service

class ServiceAdapter implements ServiceInterface {
        private $service;

        public function __construct(Service $service)
        {
                $this->service = $service;
        }

        // We need to change the way the adapter uses Service

        public function doTheThing()
        {
                $this->service->doService();
        }
}


$client = new Client(new ServiceAdapter(new Service()));
$client->doSomething();

?>
```

Es más, podríamos sustituir `Service` por cualquier otra clase que o bien cumpla la interfaz `ServiceInterface` por sí misma o bien lo haga a través de un Adaptador. De este modo, podemos modificar el comportamiento de `Client` sin tocar su código, respetando así el principio Abierto/Cerrado.

Ahora añadimos una nueva clase que pueda sustituir a Service:

```php
<?php

interface ServiceInterface {
        public function doTheThing();
}

class Client {
        private $service;

        public function __construct(ServiceInterface $service) {
                $this->service = $service;
        }
        public function doSomething() {
                $this->service->doTheThing();
        }
}

class Service {
        public function doService() {
                echo 'Performed by Service';
        }
}

class ServiceAdapter implements ServiceInterface {
        private $service;

        public function __construct(Service $service)
        {
                $this->service = $service;
        }

        public function doTheThing()
        {
                $this->service->doService();
        }
}

class NewService {
        public function theMethod()
        {
                echo 'Performed by New Service';
        }
```

```
43  }
44
45  class NewServiceAdapter implements ServiceInterface {
46          private $service;
47
48          public function __construct(NewService $service)
49          {
50                  $this->service = $service;
51          }
52
53          public function doTheThing()
54          {
55                  $this->service->theMethod();
56          }
57
58  }
59
60  $client = new Client(new ServiceAdapter(new Service()));
61  $client->doSomething();
62  echo chr(10);
63
64  $client2 = new Client(new NewServiceAdapter(new NewService()));
65  $client2->doSomething();
66
67  ?>
```

Sobre los Adaptadores

Entre los Adaptadores y las clases adaptadas existe una dependecia o acoplamiento muy estrechos. Es obvio que no podemos desacoplarlos. Este acoplamiento no es problema ya que para los efectos de nuestro Cliente, el Adaptador *es* el Servicio y no le preocupa cómo está implementado con tal de que respete la interfaz. Además, el Adaptador es una clase con un código trivial, se limita a *traducir* los mensajes entre el Cliente y el Servicio.

Esto nos podría llevar a pensar en ocultar la dependencia y hacela implícita, como forma de ahorrar un poco de código al instanciar el adaptador, pero no es muy buena idea. Todas las dependencias deberían ser explícitas.

Cuando la Inyección de dependencias se complica

Como se puede ver en los ejemplos anteriores, el desacoplamiento aumenta un poco la complejidad en el momento de instanciar la clase Cliente. Si ésta tiene varias dependencias, las cuales pueden

utilizar Adaptadores, la inyección de dependencias se hace tediosa y prolija aunque no especialmente complicada. Sin embargo, eso quiere decir que hay que repetir un buen pedazo de código en diversas partes de una aplicación para conseguir instanciar un cierto objeto.

La solución para eso es utilizar algunos de los diferentes patrones de construcción de objetos, como factorías, constructores, pools o prototipos que automaticen esa instanciación.

Otra solución es lo que se llama un Contenedor de Inyección de Dependencias (DIC o Dependency Injection Container) que es, ni más ni menos, una clase encargada de proporcionarnos objetos de una clase completamente construídos. Se puede decir que los DIC hacen uso de diversos patrones de construcción de objetos para que nosotros podamos registrar la forma en que se deben instanciar.

No siempre hay que desacoplar

El acoplamiento debemos considerarlo en el contexto del comportamiento de la clase Cliente. Ésta utiliza un comportamiento de la clase Servidor para poder llevar a cabo su propio comportamiento.

Esto no se cumple en ciertos casos. Por ejemplo, cuando hablamos de Value Objects, éstos no contribuyen al comportamiento de la clase usuaria del mismo modo. Los Value Objects se utilizan como si fuesen tipos primitivos del lenguaje y su comportamiento está destinado proporcionar servicios a la clase protegiendo sus propias invariantes. Los Value Objects son inmutables y además no tienen dependencias externas o, si las tienen, son de otros ValueObjects. Por lo tanto son objetos que pueden ser instanciados sin más con new o con un constructor estático con estático si lo hemos diseñado así.

Así que podemos distinguir entre objetos "newables" y objetos "inyectables". Miško Hevery lo explica muy bien[2]. En resumen:

Los objetos newables son aquellos que necesitan algún parámetro variable en el momento de su creación. De hecho, necesitaremos un objeto nuevo cada vez. Imagina una clase Email que represente una dirección de email o incluo un mensaje (ojo, no lo envía, sólo lo representa, pero también se encarga de validarlo y asegurarse de que está bien construido). Necesitamos un objeto distinto por cada dirección de email o mensaje. Veamos un ejemplo:

<<(code/dependencies-newable.php)

Los objetos newables no se pueden inyectar porque el contenedor de inyección de dependencias no puede saber qué parámetros necesita cada instancia concreta. Podríamos tener una factoría a la que pasarle el parámetro y que nos devuelva el objeto construido, pero ¿para qué? `new`, en este caso, es una factoría tan buena como cualquiera. Y en realidad es la mejor.

Los newables no necesitan de interfaces explícitas porque no tienes que tener un abanico de opciones. En cualquier caso un objeto newable podría extender otro por herencia y, en ese caso, la clase base sería "la interfaz" si respetamos el principio de sustitución de Liskov, de modo que ambas clases sean intercambiables.

[2] http://misko.hevery.com/2008/09/30/to-new-or-not-to-new/

Los objetos inyectables, por su parte, son objetos que se pueden construir mediante un DIC porque no necseita parámetros que cambien en cada instanciación. Normalmente tienen interfaces porque han de ser sutituibles, es decir, queremos que las clases usuarias tengan un bajo acoplamiento con ellos.

Los inyectables y los newables no se deben mezclar. Es decir, un inyectable no puede construirse con algún newable, y tampoco un newable puede construirse pasándole un injectable. Eso no quiere decir que unos no puedan usar otros, pero han de pasarse como parámetros en métodos.

Objetos inmutables

Los objetos inmutables son objetos que se instancian con unos valores que no cambian a lo largo de su ciclo de uso.

DTO: Data Transfer Objects

Se trata de objetos simples que se utilizan para mover datos entre procesos agrupando varias llamadas en una sola (Fowler[3]). Otra definición[4] los considera como contenedores ligeros y serializables para distribuir datos desde una capa de servicio.

Podrías considerarlos como el equivalente de una interfaz pero para datos. Suponen un contrato entre las partes acerca de la forma en que comparten información.

Los DTO no suelen tener comportamiento, aparte del necesario para inicialización y para serialización. En cualquier caso no lleva lógica de negocio.

En su versión más simple un DTO es un objeto con propiedades públicas y un constructor para instanciarlo y asignar los valores. Esto no es muy diferente de lo que podríamos construir con un array asociativo y es frecuente encontrar opiniones a favor de los arrays[5] basadas en su sencillez y velocidad. Sin embargo, la gran ventaja de los DTO viene de la mano del *Type Hinting*, lo cual nos permite usar los DTO como auténticos contratos sobre la transferencia de datos entre clases o partes de una aplicación, sumado a otros muchos beneficios derivados del uso de objetos.

Por ejemplo, el siguiente bug puede pasar fácilmente desapercibido, asumiendo que el array tiene una clave 'name':

```
1  $name = $data['nmae'];
```

Mientras que el código equivalente con un DTO simple nunca funcionaría:

```
1  $name = data->nmae;
```

Sin embargo, el hecho de un objeto tenga propiedades que son accesibles públicamente tiene el riesgo de que podrían modificarse, de modo que una forma más explícita sería hacer privadas las propiedades del DTO y añadirle getters para recuperar cada una de ellas. Es un poco más engorroso pero de esta forma nos aseguramos de que los objetos son realmente inmutables y los valores se mantienen sea cual sea el proceso por el que tengan que pasar.

[3] http://martinfowler.com/eaaCatalog/dataTransferObject.html

[4] http://neverstopbuilding.com/data-transfer-object

[5] http://stackoverflow.com/questions/2056931/value-objects-vs-associative-arrays-in-php

Cuando usar DTO

Patrón objeto-parámetro

Cuando un método o función necesita muchos argumentos es posible simplificar su signatura agrupando esos argumentos en uno o más objetos simples, que bien pueden ser DTO.

Observa el siguiente código:

```php
<?php

class DBConnector {

        private $host;
        private $port;
        private $user;
        private $password;
        private $database;

        public function connect($host, $port, $user, $password, $database = null)
        {
                // Validation stuff
                $this->host = $host;
                $this->port = $port;
                $this->user = $user;
                $this->password = $password;
                $this->database = $database;

                // Connection stuff
        }
}

?>
```

En este ejemplo, la clase DBConnector tiene que llevar la cuenta de los detalles de la conexión, lo que lleva como consecuencia que, entre otras cosas, debe ocuparse de validarlos. Pero esa no debería ser su tarea (Principio de Responsabilidad Única), sino que los datos de conexión deberían venir validados, pero: ¿dónde y cuándo sucedería esa validación?

Ahora compara con este otro código:

```php
<?php

class DBSettings {

        private $host;
        private $port;
        private $user;
        private $password;
        private $database;

        public function __construct($host, $port, $user, $password, $database = null)
        {
                // Validation stuff
                $this->host = $host;
                $this->port = $port;
                $this->user = $user;
                $this->password = $password;
                $this->database = $database;
        }

        public function getHost()
        {
                return $this->host;
        }

        public function getPort()
        {
                return $this->port;
        }

        public function getUser()
        {
                return $this->user;
        }

        public function getPassword()
        {
                return $this->password;
        }

        public function getDatabase()
        {
```

```
43                  return $this->database;
44          }
45
46  }
47
48  class DBConnector
49  {
50          private $settings;
51
52          public function connect(DBSettings $settings)
53          {
54                  $this->settings = $settings;
55                  // Connection stuff
56          }
57  }
58
59  ?>
```

En este ejemplo usamos un DTO para contener los datos de conexión y hacemos que el DTO contenga el código para validarlos (aunque no lo hemos escrito en el ejemplo). Gracias a eso, siempre que inicializamos un objeto de tipo DBSettings será válido, por lo que DBConnector puede aceptarlo sin tener que hacer nada más.

Devolución de datos múltiples de un método

PHP, como otros lenguajes, sólo permite un único valor de vuelta de un método. Si necesitamos devolver más, podemos componer un DTO. Volvemos a lo mismo: podría hacerse con un array asociativo, pero el DTO nos permite forzar restricciones que se controlan por el propio intérprete de PHP y lanzan errores o excepciones.

Value Objects

El concepto viene del DDD y se refiere a objetos que representan valores importantes para el dominio, su igualdad viene dada por la igualdad de sus propiedades.

Los Value Objects tienen comportamientos. No tienen identidad ni un ciclo de vida.

El ejemplo clásico de Value Object es el dinero (Money), que usamos en precios, salarios, etc. Habitualmente modelamos los valores del dinero con números de coma flotante, pero el valor no es suficiente. El dinero es el valor y la moneda que se esté utilizando: no es lo mismo 10 dólares que 10 euros. Puede argumentarse que en muchas aplicaciones no es necesario tener en cuenta la moneda, pero si algún día en el futuro eso cambiase, el impacto en el código podría ser enorme.

Dado que en OOP pretendemos encapsular lo que cambia junto, tiene sentido crear una clase de objetos para representar dinero que tenga dos propiedades: valor y monedo.

```php
<?php

class Money {
        private $amout;
        private $currency;

        public function __construct($amount, $currency) {
                $this->amount = $amount;
                $this->currency = $currency;
        }

        public function getAmount() {
                return $this->amount;
        }
}
?>
```

Aunque un objeto sea inmutable, puede tener métodos que se basen en sus propiedades para generar nuevos valores. Dicho de otra forma, deben devolver instancias de nuevos objetos con los nuevos valores, en lugar de reasignar las propiedades del objeto valor.

```php
<?php

class Money {
        private $amout;
        private $currency;

        public function __construct($amount, $currency) {
                $this->amount = $amount;
                $this->currency = $currency;
        }

        public function getAmount() {
                return $this->amount;
        }

        public function incrementByPercentage($pct) {
                return new Money($this->amount*(1+$pct), $this->currency);
        }
}

$price = new Money(100, 'EUR');
```

```php
$newPrice = $price->incrementByPercentage(.10);
echo $newPrice->getAmount();

?>
```

Usamos Value Objects cuando necesitamos representar valores:

- No se pueden representar con un tipo básico del sistema.
- Son complejos y agrupan varios datos de tipo simple que van junos.
- Requieren una validación específica.

Por ejemplo, el dinero (y con él, precios, salarios, etc) puede representarse con un valor de coma flotante, pero en cuanto necesitamos gestionar la moneda se introduce un segundo dato. Encapsulamos ambos para manejarlos de manera conjunta.

Otro ejemplo típico es una dirección postal, se trata de varios datos string que van juntos para componer una dirección. Encapsulados en un Value Object son más fáciles de manejar.

Un ejemplo más sutil es el email. Una dirección de email puede representarse con un string, pero encapsulándolo en un Value Object podemos introducir las reglas de validación (email bien formado) en el constructor, asegurándonos de que todos los objetos email que manejemos sean válidos. Eso quizá no nos asegura que los emails sean reales, pero sí nos garantiza que están bien formados.

```php
<?php


class Email {
        private $email;

        public function __construct($email)
        {
                if (filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
                        throw new InvalidArgumentException("$email is not a valid email");
                }
                $this->email = $email;
        }

        public function getValue()
        {
                return $this->email;
        }
}

```

```
21 $emailsToTry = array(
22         'franiglesias@mac.com',
23         'franiglesias@',
24         '@example.com',
25         'user@example'
26 );
27
28 foreach ($emailsToTry as $email) {
29         try {
30                 $emailObject = new Email($email);
31         } catch (Exception $e) {
32                 echo $e->getMessage().chr(10);
33         }
34
35 }
36
37 ?>
```

Patrones de diseño

Los patrones de diseño fueron introducidos en el libro de Gamma, Helm, Johnson y Vlissides: Design Patterns: Elements of Reusable OBject-Oriented Software, publicado en 1994. Se considera uno de los libros fundamentales para el moderno desarrollo de software.

Los patrones describen problemas de diseño y soluciones adecuadas para los mismos. No se trata de recetas que se puedan copiar y pegar. Más bien se trata de directrices para identificar el tipo de problema que se pretende solucionar y un modelo de clases que resuelve ese problema de una manera probada. Es tarea de cada desarrollador llevar a cabo la implementación del patrón adecuada a su proyecto.

Estos patrones, a su vez, ayudan a componer un lenguaje que permite la comunicación entre programdores, incluso cuando no están utilizando los mismos lenguajes de programación, ya que los patrones son completamente independientes de los mismos.

Por lo general, los elementos del patrón son:

- La situación o abanico de situaciones que se trata de afrontar.
- El modelo de clases que resuelve el problema y los elemntos que intervienen: clases, interfaces, subclases, etc.

El patrón Fachada (Facade)

El patrón Fachada es un patrón de tipo estructural que persigue encapsular operaciones complejas o repetitivas en una interfaz simplificada. El patrón Fachada ayuda a lograr DRY y KISS.

Supongamos que queremos o tenemos que utilizar una librería de código, ya sea procedural u orientada a objetos, la cual resulta compleja de manejar o que requiere muchas instrucciones repetitivas y/o preparación de datos y otros objetos cada vez que queremos realizar alguna tarea. La solución sería construir una clase “fachada” con métodos que utilicen esa librería. Nuestra aplicación se relaciona con la Fachada y no tiene que saber nada acerca de la librería.

En cierto modo, el patrón Fachada es la base de otros patrones como el Adaptador o el Decorador pues, al igual que éstos, envuelve o encapsula un código para que sea accesible de un modo diferente y más conveniente para la aplicación.

Las diferencias serían:

- La Fachada busca simplificar una interfaz compleja para hacerla manejable.

- El Adaptador busca convertir la interfaz "externa" en una interfaz esperada por las clases cliente, que suele estar definida explícitamente, y persigue habilitar el polimorfismo.
- El decorador añade funcionalidad a un objeto en tiempo de ejecución envolviéndolo, pero no busca cambiar la interfaz.

Un caso típico en PHP es el acceso a la base de datos, para lo que es habitual crear una clase que simplifique los pasos de configuración, conexión, petición de datos, etc. Otro ejemplo puede ser el uso de la librería gráfica GD.

Los elementos del patrón sería:

- Fachada: la clase que encapsula. Utilza el Sistema encapsulado para realizar su tarea.
- Sistema encapsulado: una librería, una clase compleja, una colección de clases relacionadas, etc. No sabe nada de la fachada.
- Usuarios: el código usuario de la Fachada y que quiere usar el sistema encapsulado. Conoce la interfaz de Fachada, y no sabe nada del sistema encapsulado.

Las Fachadas nos ayudan en varios aspectos:

- Reducen la complejidad del código al delegar en un objeto especializado llamadas complejas al sistema encapsulado.
- Mejoran la mantenibilidad al centralizar en un sólo lugar código que puede ser utilzado muchas veces.
- Reduce la complejidad de las dependencias, al confinarlas a una clase, lo que, a su vez, permite cambiarlas fácilmente reescribiendo la fachada (que es la base del patrón Adaptador). Aunque la fachada está fuertemente acoplada al sistema encapsulado, pero de ese modo desacopla el resto de la aplicación.
- Nos permite crear puntos de entrada entre capas o subsistemas.

El patrón repositorio

El patrón repositorio se utiliza en Diseño Dirigido por el Dominio para proporcionar persistencia en la capa del dominio sin acoplarla con una implementación concreta.

El patrón especificación (Specification)

El patrón especificación encapsula las reglas de negocio para seleccionar objetos.

Al usar el patrón repositorio podemos encontrarnos el problema de tener que crear métodos para hacer selecciones específicas de objetos. Cada vez que necsitamos una nueva selección con nuevos

criterios tendríamos que crear un método nuevo que aplique los correspondientes criterios. El patrón especificación soluciona ese problema encapsulando las condiciones en un objeto sencillo con un único método `isSatisfiedBy` al que se le pasa un objeto del dominio y que devuelve un valor boolean.

Esto nos indica si los objetos individuales satisfacen la especificación por lo que el repositorio tendría que obtener antes todos los objetos almacenados y examinarlos uno por uno hasta encontrar todos los que la cumplen. Por supuesto, en muchos sistemas esta tarea consume excesivos recursos por lo que debemos encontrar un método más económico. Por ejemplo, un sistema que sea capaz de cargar una selección manejable de objetos y que aplique luego la especificación en ellos, o bien, una traducción de la especificación al lenguaje del almacenamiento concreto.

En este último caso, puede ser interesante que la Especificación contenga un método que devuelva las reglas de negocio de una manera que pueda ser usada por el repositorio para hacer la consulta al medio de almacenamiento. Por ejemplo, si es un repositorio basado en SQL, podríamos tener un método `asSql()` que simplemente nos devuelva la petición tal como la necesitaríamos para realizar la consulta en la base de datos. Esto difumina un poco las fronteras entre capas, pero simplifica el código del Repositorio y elimina la necesidad de un intermediario que traduzca la especificación. Pero, al fin y a la postre, el Repositorio mismo es un espacio de frontera y, en la práctica, la Especificación sigue perteneciendo al dominio.

Otro uso de la Especificación podría ser la validación. En efecto, `isSatisfiedBy` es un método de validación, ya que comprueba que el objeto que se le pasa cumple con ciertas condiciones.

La primera ventaja es que el Repositorio no necesita conocer tanto de la Entidad o del Agregado que maneja, sino que esta tarea queda en manos de las diferentes especificaciones.

Si necesitamos usar nuevos conjuntos de criterios no tendríamos más que crear una nueva Especificación y utilizarla.

http://culttt.com/2014/08/25/implementing-specification-pattern/

Es una manera de encapsular reglas de negocio para devolver un valor boolean. Nos permite crear clases que tengan una sola responsabilidad y que se puedan combinar con otros objetos del mismo tipo (especificaciones) para gestionar requisitos complejos.

La Especificación tiene un método público isSatisfiedBy que devuelve un boolean.

Este patrón funciona bien para validar objetos, pero no tanto para seleccionarlos.

En este caso habría que usar DoubleDispatch crear un método satisfyingElementsFrom en la especficación que se corresponda con un método de selección en el repositorio. El repositorio podría utilizar una petición para obtener un conjunto de datos próximo y usar la especificacion para filtrarlos.

http://stackoverflow.com/questions/33331007/implement-specification-pattern

Especificaciones compuestas

Nos permiten crear especificaciones complejas combinando otras más simples. Aquí hay una explicación clara:

http://marcaube.ca/2015/05/specifications/

Especificaciones compuestas