Ingeniería de Software: Propiedades al vuelo, una forma de optimizar colecciones

Los objetos tienen la facilidad de poder crear propiedades una vez instanciados. Por lo general, esta característica se utiliza como una mala práctica. Sin embargo, puede emplearse como técnica de optimización de forma tal que permita una independencia modular absoluta.

Cuando se trabaja con objetos en estado puro, una colección puede terminar consumiendo un sinfín de recursos que para lo que se necesita controlar, suelen ser innecesarios.

Lamentablemente, en la mayoría de los casos los programadores terminamos optando por crear métodos que terminan manipulando datos dejando a un lado todos los principios de la programación orientada a objetos y junto a ellos, la gran cantidad de beneficios que dichos principios nos facilitan.

Estos casos que como bien dije antes nos llevan a lamentar la decisión, necesitan -sin lugar a dudas- de una solución no solo viable y que resuelva realmente el problema, sino que además, no se aparte de los principios de la programación orientada a objetos más pura.

El Problema: propiedades colectoras

Por lo general, el mayor problema con las **colecciones** de objetos, se da en aquellas cuyos compositores se encuentra a la vez, compuestos de colecciones. Es decir, en aquellos casos donde **los compositores poseen propiedades colectoras**. Sin lugar a dudas es lo peor que nos puede pasar al necesitar manipular colecciones de objetos en estado puro, sobre todo si lo que se pretende es efectuar un manejo básico.

En al menos el 98% de las aplicaciones orientadas a objetos, todo, absolutamente todo el diseño de objetos genera en este sentido, un cuello de botella a partir del objeto Usuario, del cual, más temprano que tarde, todos los objetos terminarán dependiendo, ya sea por cuestiones de herencia o por composición directa.

Y esto comienza a ser un problema visible, a medida que vamos agregando nuevos módulos al sistema. Es simple ver como cada uno de los módulos genera su propia figura de usuario pero sin embargo, en muchas ocasiones (en la mayoría me arriesgo a decir), no existe una verdadera diferencia entre los usuarios de un

módulo y los de otros, excepto por el tipo de objetos que lo componen. Por ejemplo, si en el mismo sistema conviviesen un módulo de *telemedicina* y otro para compartir código fuente (con lo son las aplicaciones *pastebin*, por ejemplo), en el primer caso los usuarios serían médicos mientras que en el segundo, programadores. Ambos tendrían un nombre de usuario y un nivel de acceso. Pero mientras que un médico podrá tener una colección de casos (o de registros médicos electrónicos), el programador, tendrá una colección de códigos fuente. ¿Para qué crear entonces dos tipos de usuarios diferentes? ¿Por qué mejor no reutilizar el mismo usuario y diferenciarlos solo por su rol dentro del sistema?

La solución

Después de mucho investigar, probar, hacer y rehacer, la idea de reutilizar al mismo objeto usuario en cada módulo, sin heredar de él y de hecho, sin siquiera crear nuevas figuras, ha demostrado solucionar no solo el problema aquí planteado, sino además, arrojó grandes beneficios, entre ellos, los principales son:

- Unifica conceptos y sentando un estándar de forma indirecta: todos los módulos utilizarán al mismo tipo de objeto como único tipo de usuario;
- Facilita la independencia y portabilidad de los módulos: al no ser necesaria la modificación del objeto usuario en el mismo objeto usuario, un mismo módulo podrá ser portado a cualquier aplicación que cuente con un objeto de tipo usuario que al menos posea una propiedad ID;
- Hace más rápido el entendimiento del código ya que el programador solo se centrará en los objetos troncales.

Las propiedades colectoras se crearían al vuelo y en tiempo de ejecución, en cada uno de los módulos.

De esta forma, el objeto usuario permanecería intacto y con un conjunto de propiedades estándar:

```
class Usuario {

    public function __construct() {
           $this->usuario_id = 0;
           $this->denominacion = '';
           $this->nivel = 0;
    }

}
```

Y los módulos de *telemedicina* y *pastebin* se encargarían de agregar sus colecciones respectivas:

```
# Recurso del controlador del modelo CasoClinico del módulo de telemedicina
# Lista los casos clínicos del usuario que solicita el recurso
# Antes de invocar al método get() del usuario, crea la propiedad colectora que almacenará
# dichos casos clínicos
public function listar() {
    $usuario = new Usuario();
    $usuario->usuario_id = isset($_SESSION['user_id']) ? $_SESSION['user_id'] : 0;
    $usuario->casoclinico_collection = array();
    $usuario->get();
}
```

```
# Lo mismo hará el recurso del controlador del modelo CodigoFuente del módulo pastebin
public function listar() {
    $usuario = new Usuario();
    $usuario->usuario_id = isset($_SESSION['user_id']) ? $_SESSION['user_id'] : 0;
    $usuario->codigofuente_collection = array();
    $usuario->get();
}
```

Como puede verse, es un simple artilugio que respetando los principios de la POO es capaz de optimizar el rendimiento de la aplicación de forma impensada.

Utilizando __call para crear pseudo métodos de agregación al vuelo

Finalmente, habrá que tener en cuenta que por cada propiedad colectora, un método de agregación será necesario e invocado por `get()` cada vez que éste sea llamado. Si en el modelo de usuario tuviésemos que crear tantos métodos de agregación como propiedades colectoras fuesen a crearse al vuelo, se perdería toda portabilidad y no solo volveríamos al principio, estaríamos haciendo un modelo inviable.

El método mágico `__call()` de PHP[1] ha sido diseñado para actuar cuando el método inaccesible de un objeto es invocado (notar que no sirve para cuando el método es invocado como función de clase. En ese caso, debe emplearse `__callStatic()`[2]). Definiendo este método en la clase usuario, su clase madre o en la que sea necesario crear propiedades colectoras al vuelo, estaríamos dando solución al problema.

Una implementación básica de este método, podría verse como la siguiente:

```
    function __call($call, $arg) {
        if(strpos($call, 'add_') === 0 && $arg) {
            $cls = str_replace('add_', '', $call);
            $property = "{$cls}_collection";
            if(!property_exists($this, $property)) $this->$property = array();
            $this->$property = array_merge($this->$property, $arg);
        }
    }
```

Ejemplo copiado literalmente del método `__call()` del objeto `StandardObject`[3] de Europio Engine

El método `__call()` recibe dos parámetros desde el intérprete: el primero, es el nombre del método que se intenta invocar. El segundo, un *array* con cada uno de los argumentos pasados al método inaccesible durante su llamada.

1 http://www.php.net/manual/es/language.oop5.overloading.php#object.call

2 http://www.php.net/manual/es/language.oop5.overloading.php#object.callstatic

3 https://bazaar.launchpad.net/~eugeniabahit/europioexperimental/trunk/view/head:/core/orm_engine/objects/standardobject.php#L60

El condicional del método se encarga de verificar que el nombre del método inaccesible comience por la cadena `add_` (esto le dará la pauta de que se trata de un método de agregación) y que el *array* `$arg` no esté vacío.

```
if(strpos($call, 'add_') === 0 && $arg)
```

Para definir el nombre de la propiedad colectora, simplemente eliminará `add_` del nombre del método y concatenará el resultado a `_collection`:

```
$cls = str_replace('add_', '', $call);
$property = "{$cls}_collection";
```

Finalmente, si la propiedad no existe (por si se olvidó crearla), la creará para luego, combinarla con la actual:

```
if(!property_exists($this, $property)) $this->$property = array();
$this->$property = array_merge($this->$property, $arg);
```