the original Hacker

creado por EUGENIA BAHIT

Jugando con la inteligencia

número 8

Woman Eyes creado por Mourad Mokrane - Silueta de Mujer creado por Leonardo B. Cunha

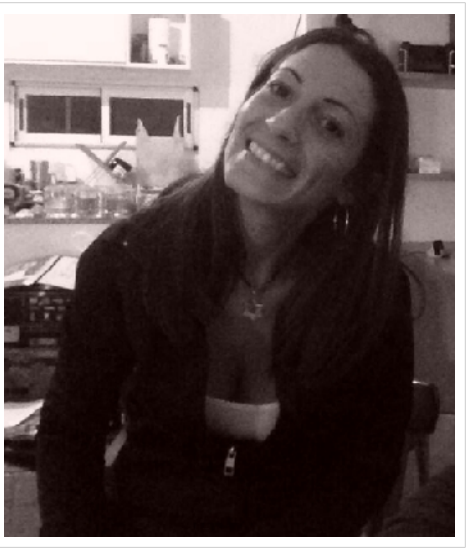
AUTHORS AGAINST DRM

SIN DRM

GNU run free run GNU

THE ORIGINAL HACKER
SOFTWARE LIBRE, HACKING y PROGRAMACIÓN, EN UN PROYECTO DE

EUGENIA BAHIT



@eugeniabahit

GLAMP HACKER Y PROGRAMADORA EXTREMA

Hacker especializada en Programación eXtrema e Ingeniería Inversa de código sobre GNU/Linux, Apache, MySQL, Python y PHP. EUGENIABAHIT.COM

Docente e instructora de tecnologías GLAMP CURSOS.EUGENIABAHIT.COM
CURSOSDEPROGRAMACIONADISTANCIA.COM

Miembro de la Free Software Foundation FSF.ORG y The Linux Foundation LINUXFOUNDATION.ORG.

Creadora de python-printr, Europio Engine, JackTheStripper. Vim contributor. Fundadora de Hackers n' Developers Magazine y responsable editorial hasta octubre '13.

Buenos Aires,
Julio/Agosto 2014

ÍNDICE DE LA EDICIÓN NRO8

PHP y el manejo de objetos en memoria

*— Eugenia Bahit agradece a **Hugo** (@huguidugui) por la **revisión ortográfica** de este artículo*

El presente es una transcripción adaptada del intercambio de ideas sobre objetos y referencias en la estructura foreach, surgidas en la clase de PHP avanzado con Xabi Pico Urisabel.

El tratamiento que PHP hace de los objetos a bajo a nivel, es realmente sorprendente y necesita sin dudas un apartado especial.

Si bien en la documentación oficial de PHP se habla sobre el manejo que el lenguaje hace de los objetos en la memoria aclarando que éstos en realidad, no son pasados por referencia [ver: http://php.net/manual/es/language.oop5.references.php], considero necesario dar una explicación más a fondo con ejemplos concretos puesto que no es algo fácil de relacionar.

De hecho, fue lo que nos sucedió no hace mucho, a **Xabi Pico Urisabel (@xabipic)** y a mí en medio de una clase de PHP y que nos llevó a hacer una investigación más profunda sobre los objetos basándonos en ejemplos concretos.

Cuando se crea un objeto -asignándolo a una variable de tipo `$a = new A();`- la variable $a se almacena en memoria pero el valor de $a no es el objeto en sí mismo, sino un puntero hacia el objeto A.

Gráficamente estaría sucediendo algo como esto:

```
--------+-----------
   D1   |     D2
--------+-----------
  obj A |  $a = D1
--------+-----------
```

(la línea de arriba representa una dirección de memoria ficticia y la de abajo, el dato almacenado en ella)

No interesa si los objetos creados se asignan solo a variables o éstas, además, se introducen en un *array*; los elementos del *array* continuarán apuntando a la dirección en memoria del objeto, ya que la variable lo hace:

```
$c = array($a);
```

```
--------+-----------+-------------
   D1   |     D2    |      D3
--------+-----------+-------------
  obj A |  $a = D1  | $c[0] = D1
--------+-----------+-------------
```

Entonces, si yo creo un alias de `$c[0]` (como el alias generado en la estructura `foreach` mediante el constructor `as`), ese alias también estaría apuntando a la dirección de memoria del objeto. Porque si bien el `as` crea variables al vuelo, no les otorga un valor a las mismas en caso de ser objetos sino que como valor, le otorga un puntero hacia la dirección de memoria del objeto.

Por este motivo, **los objetos pueden modificarse mediante `foreach` sin necesidad de ser pasados por referencia.**

Esto podría explicar además, que si bien lo siguiente agrega la propiedad `foo` al objeto A:

```
php > class A { }
php > $a = new A();
php > $b = new A();
php > $c = array($a, $b);
php > $c[0]->foo = 'bar';
php > print_r($a);
A Object
(
    [foo] => bar
)
```

Al destruir la variable `$a` igual se conserve el objeto:

```
php > unset($a);
php > print isset($a) ? 'True' : 'False';
False
php > print_r($c);
Array
(
    [0] => A Object
        (
            [foo] => bar
        )

    [1] => A Object
        (
        )

)
```

Porque en memoria, estaría pasando esto:

```
--------+-----------+-------------
   D1   |     D2    |      D3
--------+-----------+-------------
  obj A | LIBERADA  | $c[0] = D1
--------+-----------+-------------
```

Por lo tanto, siempre que se esté haciendo referencia al objeto, será éste quien sea modificado y no la variable que lo contiene:

```
php > class A { }
php > $a = new A();
php > $a->foo = 'bar';
php > print_r($a);
A Object
(
    [foo] => bar
)
php > $b = array($a);
php > foreach($b as $obj) $obj->foo = strtoupper($obj->foo);
php > print_r($a);
A Object
(
    [foo] => BAR
)
```

Por ejemplo, en el caso anterior, la estructura `foreach` itera sobre un *array*, construyendo el alias sobre el objeto propiamente dicho. Por lo tanto, éste será un puntero hacia la dirección en memoria del objeto y por ese motivo, es que el `print_r()` final lo muestra modificado.

Sin embargo, si en vez de iterar sobre el *array*, se iterase directamente sobre el objeto, el alias se estaría construyendo sobre las propiedades del objeto y no sobre el objeto mismo. Eso explicaría porqué en ese caso, solo podría modificarse el objeto si el valor de la propiedad es pasado por referencia:

```
php > # sin pasar el valor de la propiedad por referencia
php > foreach($a as $property=>$value) $value = strtolower($value);
php > print_r($a);
A Object
(
    [foo] => BAR
)


php > # ahora, pasando el valor de la propiedad por referencia
php > foreach($a as $property=>&$value) $value = strtolower($value);
php > print_r($a);
A Object
(
    [foo] => bar
)
```

Es también de hacer notar, que **el ámbito de los objetos es independiente al ámbito de las variables**,

puesto que el mismo, jamás deja de ser puntero hacia la memoria **incluso aún dentro de una función**. Es decir, si creo por ejemplo un *array* `$c` con dos objetos `$a` y `$b` y los paso a una función `foo()` como parámetro `$d`, si `$a` se encuentra en *D2* apuntando a *D1*, el primer elemento de `$c`, se encontraría en *D3* apuntando a *D1* y si lo paso como `$d`, `$d` estaría en *D4* apuntando también a *D1*. Esto supone poder modificar *«al array»* indirectamente si lo que se modifica es el objeto, sin necesidad de ser pasado por referencia.

```
php > class A { }
php > $a = new A();
php > $b = new A();
php > $c = array($a, $b);
php >
php > function foo($d) { $d[0]->bar = 123; }
php > foo($c);
php >
php > print_r($a);
A Object
(
    [bar] => 123
)
```

Para ampliar la información sobre las referencias, recomiendo leer la siguiente sección del manual de PHP: http://php.net/manual/es/language.references.php

PHP: Hacking, Debugging o simplemente diversión

*— Eugenia Bahit agradece a **Hugo** (@huguidugui) por la **revisión ortográfica** de este artículo*

El presente paper refleja diversos mecanismos implementados sobre scripts de PHP para analizar el uso de memoria de forma experimental y por lo tanto solo deben ser tomados como un curioso experimento.

Si te gusta experimentar, analizar y dejarte llevar por un espíritu de curiosidad extrema, jugar a *debuggear* y analizar internamente lo que sucede cuando ejecutas un *script* PHP, puede ser muy divertido.

Si bien PHP es un lenguaje interpretado y de alto nivel que no permite efectuar un verdadero *debug* desde el propio código (por ejemplo, no puedes saber en qué dirección de memoria se está escribiendo determinado valor), con apenas conocer el ID del proceso del *script* puedes comenzar a divertirte.

ADVERTENCIA:** el contenido de este paper solo es **de interés para el disfrute y satisfacción personal** de quien se deja invadir por la curiosidad. Fuera de ello, **carece de toda utilidad.

Introducción

Mediante el siguiente comando se puede obtener una lista de procesos relacionados con cualquier búsqueda, como por ejemplo, php:

```
ps aux | grep php
```

Sin embargo, podría tratarse de una lista extensa que se debería filtrar manualmente o mediante expresiones regulares. Por ejemplo: si estoy ejecutando un *script* llamado `file1.php` mediante el usuario *eugenia*, podría filtrar la búsqueda de forma mucho más precisa mediante lo siguiente:

```
ps aux | grep -E "eugenia\s* [0-9]* .*php\ file1.php"
```

Con los comandos anteriores obtendríamos solo la línea correspondiente al proceso de nuestro *script*:

```
eugenia  28608  0.6  0.3  42600  7700 pts/8    S+   13:30   0:00 php file1.php
```

Si quisiéramos «hilar más fino» aún, podríamos obtener solo el ID del proceso si partimos la cadena en palabras. Para ello, guardaremos la ejecución del comando en una variable que utilizaremos luego para extraer el ID:

```
p=(`ps aux | grep -E "eugenia\s* [0-9]* .*php\ file1.php"`) ; echo ${p[1]}
```

PHP Y EL USO DE MEMORIA

El ID del proceso puede ser útil para conocer el uso de memoria de un *script*.

```
eugenia@co...:$ p=(`ps aux | grep -E "eugenia\s* [0-9]* .*php\ file1.php"`); echo ${p[1]}
29043
eugenia@co...:$ cat /proc/29043/status
Name:      php
State:     S (sleeping)    -- estado del proceso (1)
Tgid:      29043
Pid:29043
PPid:      28263
TracerPid: 0
Uid:1000   1000   1000   1000
Gid:1000   1000   1000   1000
FDSize:    256
Groups:    4 24 27 30 46 104 109 124 1000
VmPeak:       43144 kB     -- tamaño máximo de memoria virtual
VmSize:       42600 kB     -- tamaño total de memoria virtual en uso
VmLck:            0 kB
VmPin:            0 kB
VmHWM:         7704 kB
VmRSS:         7700 kB     -- uso real en memoria física
VmData:       11816 kB     -- tamaño del segmento de datos en memoria virtual
VmStk:          136 kB     -- tamaño de la pila en memoria virtual
VmExe:         7680 kB     -- tamaño del texto en memoria virtual
VmLib:        18180 kB
VmPTE:           84 kB
VmSwap:           0 kB
Threads:   1
...
```

(1) Los posibles estados de un proceso pueden ser:

```
R = running
S = sleeping
D = sleeping (pero no puede interrumpirse)
Z = zombie
T = traced or stopped
```

Para **medir el estado del proceso**, se puede mantener «abierto» el *script* de varias formas:

1. invocando a la función `sleep(N)` para retrasar *N* segundos la ejecución
2. invocando a la función `readline()` para dejar el *script* abierto a la espera de una entrada del usuario

Alternativamente, se puede efectuar el análisis desde el propio *script* obteniendo la ID del proceso con la función `getmypid()`.
Luego, se invocará a `shell_exec()` para correr los comandos que nos permitan visualizar el consumo de memoria.

El siguiente ejemplo, compara el uso de memoria que se hace al crear un *array* extenso con diferentes métodos.

Para un resultado objetivo recomiendo ejecutar los código mediante PHP-CLI iniciando una nueva interfaz en cada ejecución.

```
$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = array(); $i = 0; while($i <= 1024*1024) $a[] = $i++;
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));

/*
    VmRSS:     7924 kB
    VmRSS:   118556 kB
*/


$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = array(); for($i=0; $i <= 1024*1024; $a[]=$i++);
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));

/*
    VmRSS:     7924 kB
    VmRSS:   122556 kB
*/


$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = range(0, 1024*1024);
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));

/*
    VmRSS:     7924 kB
    VmRSS:   122540 kB
*/
```

Para tener una idea aproximada de **cuánta memoria consume el código propiamente dicho** y cuánta es la «memoria inicial» consumida por PHP, te propongo realizar el siguiente experimento paso a paso:

1) Crea un archivo llamado `file.php` con el siguiente código:

```
<?php
$pid = getmypid();
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
$a = array(); $i = 0; while($i <= 1024*1024) $a[] = $i++;
print shell_exec(sprintf("grep -i vmrss /proc/%s/status", $pid));
sleep(30);
?>
```

2) Corre el archivo ejecutando:

```
php -f file.php
```

3) Rápidamente, antes de transcurridos los 30 segundos del `sleep`, ejecuta el siguiente comando para conocer el uso de memoria hecho por el *script*:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; grep -i vmrss /proc/
$proid/status
```

4) Ahora, abre el archivo nuevamente y borra todo el código excepto la línea del `sleep()` de forma tal que el archivo se vea así:

```
<?php
sleep(30);
?>
```

5) Repite los pasos 2 y 3 y compara el uso de memoria.

El mismo experimento puede realizarse con diferentes algoritmos y así efectuar un análisis exhaustivo que permita extraer conclusiones confiables. Por ejemplo **¿cuánta memoria consume en realidad la creación de variables?** Puedes contrastar el último análisis contra el resultado del siguiente:

```
<?php
$a = 1234567890123456789012345678901234567890;
sleep(30);
?>
```

Y ahora, para hacerlo más interesante, agrégale un `unset()` a la variable:

```
<?php
$a = 12345678901234567890123456789012345678901234567890;
unset($a);
sleep(30);
?>
```

¡Vaya sorpresa! Sucede que **`unset()` se limita a destruir la variable a nivel de *script* pero no libera la memoria de forma inmediata.** No encontré una explicación oficial de lo que ocurre verdaderamente, pero la no oficial sería que el recolector de basura de PHP liberará esa memoria en el momento que la necesite a fin. Esto podría estar justificado por el hecho de que si la liberación fuese inmediata, debería trabajar la CPU para hacerlo.

Se puede efectuar una innumerable cantidad de pruebas incluso utilizando herramientas mucho más completas que un simple `grep`. Por ejemplo, si eres de esas personas curiosas capaces de pasar horas y hasta días *debuggeando*/analizando frente al ordenador, con `strace` podrás divertirte viendo qué es lo que va haciendo PHP mientras se ejecuta tu *script* (personalmente me encanta ver el trazo cuando llega al momento de dormir el *script* xD):

```
strace php file.php
```

También puedes utilizar:

```
strace -dC php file.php
```

(mi preferida) para ver un *debug* propio de `strace` y al final, un resumen de las diferentes llamadas al sistema que PHP va haciendo. Incluso, puedes filtrar las llamadas mediante:

```
strace -e read,write php file.php
```

`ltrace`, quien se utiliza del mismo modo que `strace`, también puede resultarte tan enloquecedor como entretenido y si bien a la mayoría resulta menos útil que `strace`, a mi me agrada mucho más:

```
ltrace php file.php
```

Otra alternativa es utilizar `gdb` aunque a pesar del enorme cariño que siento por `gdb`, para un *script* (de PHP) no suele ser demasiado útil:

```
gdb php
```

y luego:

```
(gdb) run file.php

...
^C
Program received signal SIGINT, Interrupt.
0xb7fdd424 in __kernel_vsyscall ()
(gdb) bt
#0  0xb7fdd424 in __kernel_vsyscall ()
#1  0xb78f7d06 in nanosleep () at ../sysdeps/unix/syscall-template.S:82
#2  0xb78f7aff in __sleep (seconds=0) at ../sysdeps/unix/sysv/linux/sleep.c:138
...
```

Luego, con `pmap` se puede ver el mapeo de memoria que hace el proceso de PHP:

Corriendo el *script* por un lado con `php -f file.php`, luego ejecutar:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; pmap -x $proid
```

Otros recursos útiles, simpáticos y/o curiosos

top del *script*: Correr el *script* y en paralelo ejecutar:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; top -p $proid
...
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 8045 eugenia   20   0  153m 119m 5372 S    0  6.3   0:00.72 php
```

ps del *script*: Correr el *script* y en paralelo ejecutar:

```
p=(`ps aux | grep -E ".*php\ -f\ file.php"`); proid=${p[1]}; ps -p $proid -o %cpu,%mem,cmd
%CPU %MEM CMD
14.8  6.2 php -f file.php
```

El porcentaje de CPU usado varía de acuerdo al estado del *Script*. Cuánto más tiempo transcurre y el *script* duerme con el `sleep()` menos uso de CPU habrá mientras que el uso de memoria se mantiene impoluto. Si se utilizase `readline()` en lugar de `sleep` podría ponerse en ejecución el `script` mientras se corre el comando reiteradas veces a fin de ver como el consumo de CPU va llegando a `0`.

Software Libre y la diferencia con el Software privativo desde una óptica intrínsecamente profesional y no política

*— Eugenia Bahit agradece a **Hugo** (@huguidugui) por la **revisión ortográfica** de este artículo*

Ensayo sobre cómo decidir tu futuro profesional

Comúnmente, las personas confunden los conceptos de «libertad» y «gratuidad» creyéndolos sinónimos. Pero a esta altura, todos deberíamos saber que no lo son.

Sin embargo, no es mi responsabilidad como programadora, revertir esta garrafal confusión y solo es algo que me interesa cuando «estoy de humor» para que me interese. No obstante, **como miembro de la *Free Software Foundation,* docente e instructora de programación es mi obligación educar a los programadores** en esto, haciéndoles entender que **la diferencia entre uno y otro** no es una cuestión de «bandos» sino que **es una cuestión de «desarrollo profesional»**.

> *Para un programador, **la diferencia entre «libre» y «gratis» es «su propio futuro profesional».***

Demás está decir que promuevo el desarrollo de Software Libre y por lo tanto, **no haré «propaganda política».** Esto significa que no te diré porqué debes desarrollar Software Libre ni mucho menos intentaré convencerte de que lo hagas. Sinceramente, **lo que hagas de tu vida y de tu profesión, no es algo que me importe.** Pues es tu vida y solo tú debes decidir sobre ella.

Yo simplemente, te explicaré la diferencia entre «una cosa y la otra» y procuraré darte las herramientas necesarias para que decidas cuál será tu camino, incluso aunque ello implique que no elijas el camino del Software Libre. Pues **si alguien que no entiende el Software Libre desde una óptica profesional intentase desarrollarlo, podría ser contraproducente para quiénes sí lo entendemos**.

Hablando hace poco con uno de mis alumnos, le dije:

> *— Para poder decidir si desarrollarás Software libre o Software privativo debes plantearte primero por cuál actividad cobrarás.*

Desarrolles Software Libre o privativo, si de tu trabajo se trata, cobrar **tendrás que cobrar**. Eso aquí no está en juego. Lo único que **debes decidir** es **por cuál concepto cobrarás** y para ello, las opciones son solo dos, aunque de cada una de ellas, puedan derivar decenas y hasta cientos de alternativas. El límite estará en tu creatividad. Pero **la elección de una de las dos alternativas, estará solo y únicamente en el valor que tú mismo te otorgues como programador.** Y cuando hablo de valor, no hablo de precio. Piensa en valor como en *«cuánto valoras a tus padres»*. Entonces **¿cuánto te valoras como programador?**

Seguramente, **si recién comienzas a programar**, tus conocimientos no sean muchos y todo te cueste demasiado esfuerzo. Tanto esfuerzo que hasta llegues a creer que resolver un ABM se podría comparar a tocar el cielo con las manos. Entonces, estarás creyendo que te valoras. Sin embargo, al escuchar los requerimientos de tu cliente experimentarás una sensación de miedo e incertidumbre e internamente te preguntarás si verdaderamente podrás resolverlo.

Aunque también podrías experimentar una sensación de omnipotencia y a pesar de no saber mucho más que la forma de definir una variable, estarías convencido de que no existe nada que no puedas resolver. Entonces, creerás que te valoras pero sin embargo, en el fondo, experimentarás una especie de miedo a *«meter la pata»*.

Estando en cualquiera de estos grupos, **seguramente tengas muchísima más facilidad, talento y preparación para generar ideas y crear estrategias comerciales.** Pues no existe ser humano en el mundo que sea incapaz de vender. Sin embargo, los hay incapaces de programar. Claramente, en estos casos, **cobrar por tu trabajo como «comerciante» te permitirá ganar más dinero que cobrar por tu trabajo como «programador»**, pues como programador solo lograrás hacer un trabajo mediocre que cualquier otro programador un poco más experimentado podría mejorar.

Puesto en un ejemplo (con números redondos e irreales):

> *Mientras que por tu trabajo como programador podrías ganar no más de 100 rupias, cobrando como*

> *comerciante, podrías ganar 50 con cada persona que use tu Software y, con que tan solo consigas 3 personas que lo utilicen, estarías ganando más dinero que como programador.*

Pero ahora, puede pasar todo lo contrario de lo dicho anteriormente y **tal vez seas de esas personas especiales que valora en serio su profesión**. Esto puede significar que estudies con el único fin de superarte a ti mismo; que intentes perfeccionarte día a día; que necesites estar programando cosas nuevas constantemente; que solo te motives para programar cuando te enfrentas a nuevos desafíos; que tú mismo te propongas desafíos y que te guste llegar al fondo de las cosas por el mero placer de haberlo logrado. Si este es tu caso, entonces, **sabes valorarte como programador y para ti, verdaderamente no existen imposibles**; pues seguramente sabes vencer tus miedos porque **tu amor por la profesión es superior a todo lo demás.**

Si este es tu caso, tu propio ego te obligará a cobrar como programador, pues de lo contrario, sería tu orgullo el que estaría en juego.

> *Pero a diferencia del anterior, tu podrás ganar 1000 rupias por lo que aquel solo llegaría a ganar 100. Entonces, siempre te será incluso más fácil, ganar el dinero por lo que vales como programador.*

En definitiva, debes elegir entre dos opciones:

1) **ser un comerciante que aprendió a programar para poder hacer negocios;**
2) **ser un programador que valora su trabajo.**

Pero no debes hacer un juicio de valor. Esto se trata de ser honesto con uno mismo. Se trata de tener humildad y de no ponerse metas inalcanzables.

> *A veces, **pretender seguir un objetivo obligado por una ideología, puede costarte demasiado caro.***

Entonces, **nadie puede juzgarte y decir si tu decisión es buena o mala**; correcta o incorrecta. Decidirás lo que esté a tu alcance. Y **si decides con honestidad y humildad, nadie podrá juzgarte.**

Ahora, la pregunta final es: **¿desarrollo Software Libre o privativo?**

Si tu decisión ha sido la primera opción, es decir, ser un comerciante que programa, la única forma de

ganar el suficiente dinero para vivir, será crear Software privativo. **Venderás a los usuarios un permiso para utilizar el software que has desarrollado** ya que ello te permitirá ganar dinero suficiente. Incluso, puede que no necesites programar más de uno o dos Software en toda tu vida. De lo contrario, si intentas cobrar solo tu trabajo como programador, puede que el dinero que ganes no te alcance para vivir y entonces, experimentes mucha frustración.

Si en cambio elegiste que se te valore como programador, cobrarás por cada minuto del tiempo que emplees en crear cada uno de los Software que se te encargue y harás valer celosamente cada segundo de tu tiempo. **No te importará quién ni para qué use tu Software, pues cuanta más gente use tu Software, más prestigio alcanzarás** como programador y **cuanto más prestigio tengas, más valdrá un segundo de tu tiempo** y por consiguiente, más elevada será la cotización de tu hora de trabajo. Entonces, el Software Libre te será más conveniente. Pues solo así lograrás llegar a más gente y ganar más prestigio, características necesarias para poder cotizar mejor tu trabajo.

Y, si me quedaba pendiente esclarecer la diferencia entre «libre» y «gratis» podríamos decir que:

> ***Gratis es:*** *el trabajo de programación realizado por un comerciante ya que éste, no cobrará su trabajo de programación, sino que cobrará el uso que finalmente se haga sobre eso que programó.*

En cambio:

> ***Libre es:*** *el trabajo hecho por quien cobra por programar sin hacer restricciones sobre lo que se haga con aquello que programó.*

Elige a conciencia y estarás siguiendo el camino más apropiado.

La Orientación a Objetos y sus enfoques como estilo y paradigma

*— Eugenia Bahit agradece a **Hugo** (@huguidugui) por la **revisión ortográfica** de este artículo*

Comprender desde las bases la radical diferencia que existe entre la Orientación a Objetos vista como paradigma de programación y la que se observa desde el punto de vista arquitectónico, será la diferencia entre un sistema informático perfecto y uno erróneo.

La Orientación a Objetos parece, en ocasiones, ser un gran misterio difícil de dilucidar. Esto se debe a que **mucho se habla** de ella, **pero muy poco se conoce en realidad**.

Lo cierto es que la Orientación a Objetos (OO) no solo es un paradigma de programación. La Orientación a Objetos, de hecho, puede ser vista, analizada y estudiada desde dos **enfoques** absolutamente diferentes:

- como **paradigma** de programación
- como **estilo** arquitectónico

En una gran parte de la bibliografía, la OO se abarca desde ambas ópticas pero sin hacer una clara distinción. Tal es así, que sendos enfoques llegan a confundirse generando en el lector o estudiante, una idea equivocada sobre qué es en realidad la OO.

Este error, es el que más nos cuesta corregir en el público en general, a los docentes y teóricos que nos dedicamos al estudio de la OO desde sus dos enfoques. Por ello, intentaré realizar dicha distinción utilizando ejemplos concretos, recurriendo así al código antes que a las explicaciones teóricas.

Comenzaré entonces, con un ejemplo clásico de la orientación a objetos, basándome en *el objeto Usuario*.

Si recurrimos a la **OO como estilo arquitectónico**, nos podremos encontrar con la siguiente clase:

```
class Usuario(object):

    def __init__(self):
        self.nombre = ''
        self.clave = ''
        self.email = ''
        self.telefono = ''
```

```
    def registrar(self):
        pass
```

En cambio, si el mismo ejemplo, se abarca desde el paradigma, nos encontraríamos con algo similar a:

```
class Usuario(object):

    def __init__(self):
        self.usuario_id = 0
        self.nombre = ''
        self.datoscontacto = []

    def add_datocontacto(self, datocontacto):
        self.datoscontacto.append(composite(DatoContacto, datocontacto))

    def save(self, clave):
        pass

    def get(self):
        pass

    def destroy(self):
        pass


class DatoContacto(object):

    def __init__(self):
        self.datocontacto_id = 0
        self.denominacion = ''
        self.valor = ''
        self.usuario = 0

    # etc...
```

Como puede verse, **la complejidad del paradigma es lejanamente superior a la sencillez del estilo**.

> *El paradigma es complejo, detallista, minucioso, lógico y racional, mientras que **el estilo es pragmático**.*

Originalmente, la OO **desde el punto de vista arquitectónico,** promovía el uso del paradigma para el diseño de todos los componentes del sistema. Sin embargo, no lo hacía de forma explícita, sino que simplemente se limitaba a definir que **«todo componente del sistema debía ser tratado y manipulado como un objeto»**. Es decir, no se hablaba de «paradigma» sino tan solo de «objetos».

Esto, dio origen a que por efecto de los usos y costumbres, el estilo arquitectónico utilizara como valor prioritario, la sintaxis y elementos de la Programación Orientada a Objetos (POO) pero dejando a un lado **el «paradigma»**, el cual define **«un modo único e indiscutible de diseñar los objetos de un sistema pero no necesariamente todos sus componentes»**.

Indiscutible significa «que no tiene lugar a objeciones».
De esta forma, si yo dijese que el objeto persona «tiene dos ojos» cualquiera podría objetar mi afirmación diciendo que «los ojos no los tiene la persona, sino que los posee la cara de la persona». Y esto último, es indiscutible. Pues nadie podría decir que los ojos NO se encuentran ubicados en la cara. Es entonces, **cuando se llega a este nivel de abstracción inequívoca e inobjetable**, que **los objetos están siendo razonados desde un enfoque racional; desde el enfoque del paradigma de programación**.

Así, puede concluirse (en una forma demasiado simplista y poco ortodoxa) en que *«cualquier objeto sujeto a debate es un objeto visto desde un enfoque arquitectónico pero no teórico»*. Siendo inmensamente más ortodoxos, diríamos que:

> *El estilo orientado a objetos es pragmático mientras que el paradigma de programación es racional*

Copiaré ahora el ejemplo según estilo, resaltando aquello que para el paradigma sería incorrecto:

```
class Usuario(object):

    def __init__(self):
        self.nombre = ''
        self.clave = ''
        self.email = ''
        self.telefono = ''

    def registrar(self):
        pass
```

Objeciones según el paradigma
Para el paradigma de programación, se debe razonar de forma purista y ser muy racional al momento de abstraer los contextos.

- Objeción sobre la propiedad clave: Yo, como usuario, no me caracterizo por mi contraseña. De hecho, nadie debería conocer mi contraseña y las cualidades que «nos definen» son aquellas cualidades que «se perciben».
- Objeción sobre las propiedades e-mail y teléfono: yo como usuario no soy «de e-mail» y si dijese que «tengo un teléfono» debería tratar al mismo como un objeto. Sin embargo, no estoy hablando del «aparato telefónico» sino del número «de contacto». De allí la indiscutible afirmación de que tanto el teléfono como el e-mail son datos de contacto que «pertenecen» al usuario.
- Objeción al método registrar: registrar una acción que puede realizar un ser humano sobre el sistema informático. Por eso, a dicha funcionalidad se la denomina «recurso» (herramienta con la que cuenta el usuario para realizar determinada acción). Los recursos de un sistema, son parte de los componentes del sistema pero no forman parte de los actores. Los actores del sistema, según el paradigma, solo pueden efectuar por sí solos, 3 acciones: crearse (para persistir), leerse y destruirse.

Otro aspecto a tener en cuenta, es que **el sistema informático perfecto** no es aquel que elige la OO desde uno u otro enfoque, sino que **es aquel que sabe como utilizar ambos enfoques de forma equilibrada**.

El estilo arquitectónico tratará a cada uno de los componentes del sistema como un objeto. Esto significará que desde un *helper* hasta un formulario será un objeto para el sistema.

> ***Metáfora:*** *como ayuda para entender los sistemas informáticos equilibrando el uso de objetos,* ***pensar en un sistema como en un rodaje de TV*** *en el que existen actores y otros roles. Entonces,* ***todos los*** *componentes que puedan considerarse «****actores*** *principales y secundarios»* ***se definirán según el paradigma*** *y el resto, lo hará solo arquitectónicamente.*

Es el caso concreto de **MVC donde los actores son «los modelos»** mientras que las vistas, controladores, *helpers* y demás componentes del núcleo del sistema, solo poseen un «estilo orientado a objetos».

Debe además tenerse en cuenta que cuando se ha elegido un estilo orientado a objetos para el diseño del sistema, toda información que deba hacerse persistir deberá -previamente- analizar qué tipo de persistencia aplicaría a cada caso. En estos sistemas puede haber dos tipos de **persistencia**:

- la persistencia **de datos**
- la persistencia **de actores**

Todo lo que se considere «actor» será un objeto en el sentido más purista del término, es decir que será tratado bajo el dogma del paradigma sin excepciones. Según la opinión profesional del programador, los **«objetos puros»** (actores) podrán hacerse persistir en una **base de datos** (SQL o no-SQL), en archivos, etc.

Sin embargo, toda persistencia de **«datos fijos»** (no variables) preferentemente deberá hacerse **en archivos de configuración** (*config*) **o inicialización** (*settings*).

> *Tener en cuenta el tipo de persistencia, será lo que otorgue mayor portabilidad y facilidad de mantenimiento y evolución al sistema.*

Cuando uno se enfrente a determinados datos que requieran persistir en una DB, se estará entonces frente a un objeto que no está siendo tratado de la forma adecuada, ya sea porque se lo está tratando de forma arquitectónica cuando es en realidad un actor o bien, porque siendo un objeto de estilo estamos queriendo darle un tratamiento de actor.

DEFECTIVE BY DESIGN.org

HACKTIVISMO ÉTICO

LAS SIGUIENTES COMPAÑÍAS HACEN DAÑO A LA LIBERTAD DE LAS PERSONAS

amazon.com

Mediante sus plataformas Kindle y Prime. **Leer más »**

Apple utiliza el DRM para controlar iOS y a los usuarios de OS X. **Leer más »**

Microsoft

El DRM se incorpora en el corazón de Windows y muchos servicios como Silverlight, imponen el DRM en los usuarios. **Leer más »**

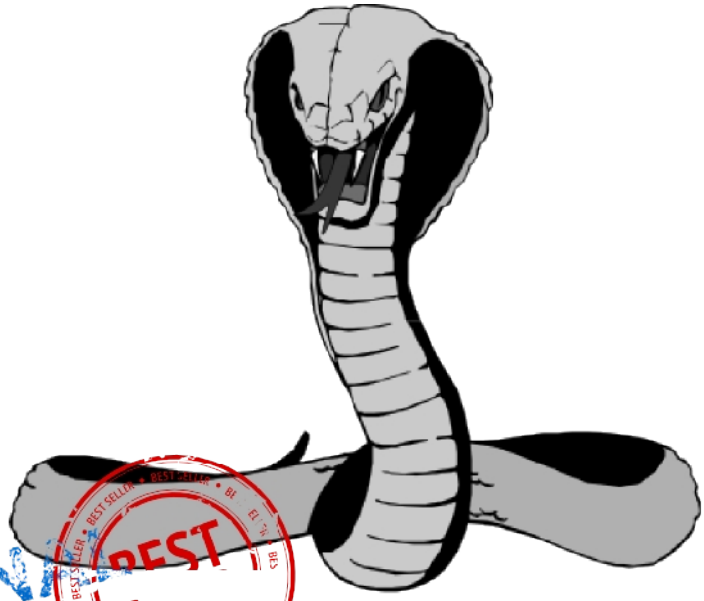
NETFLIX

No satisfecho con el uso de DRM para su servicio de *streaming*, ahora Netflix está tratando de imponer el DRM en la Web. **Leer más »**

SONY

Sony ha utilizado acciones legales para hostigar e intimidar a las personas que han modificado sus sistemas PS3. **Leer más »**

SI DE VERDAD APOYAS EL SOFTWARE LIBRE, ÚSALO, PROMUÉVELO Y ABANDONA LO PRIVATIVO SIN EXCUSAS. EL CAMBIO ESTÁ EN TUS MANOS.



The Original Hacker #_8_