the original Hacker

creado por EUGENIA BAHIT

Jugando con la inteligencia

número 2

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

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, The Linux Foundation linuxfoundation.org e integrante del equipo de Debian Hackers debianhackers.net.

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

MATERIAL DE LIBRE DISTRIBUCIÓN
HECHO EN LA REPÚBLICA ARGENTINA

Buenos Aires, 3 de Diciembre de 2013

ÍNDICE DE LA EDICIÓN NRO2

Desarrollar Software Aplicando la Ingeniería Inversa: el arte de la ciencia

Desarrollar Software puede convertirse en un verdadero "arte científico" si se es capaz de combinar paralelamente, lógica y pensamiento lateral. Y eso, es lo que la ingeniería inversa nos propone. En esta edición de The Original Hacker, veremos en qué consiste la ingeniería inversa de Software aplicada durante su desarrollo.

Si alguna vez te viste envuelto en una maraña de ideas y procedimientos al momento de desarrollar una determinada funcionalidad y terminaste enredándote tanto que decidiste "dibujar" lo que necesitabas alcanzar y desde allí, fuiste sustituyendo tu "dibujo" por código hasta terminarlo, lo que hiciste, fue aplicar un procedimiento de ingeniería inversa para resolver ese requerimiento que tanto te enredaba.

A la ingeniería inversa muchas veces se le suele tener miedo y esto sucede -creo yo- por la inmensa cantidad de mitos alrededor de este maravilloso concepto pero por sobre todo -y en definitiva-, por la gran ignorancia y desconocimiento sobre su existencia.

> *La ingeniería inversa plantea una forma lateral de alcanzar objetivos, partiendo del resultado -el "qué"- para obtener el modo de alcanzarlo -el "cómo"-*

La ingeniería inversa es amplia. Es amplia como ciencia y como técnica pero también como concepto, ya que si bien posee un único significado, sus aplicaciones son tan variadas y disímiles que es allí donde se produce la distorsión conceptual que tantos miedos genera.

Al proponer el inicio de los procesos partiendo del objeto final, es que comúnmente se la relaciona con una de sus aplicaciones más frecuentes:

- Ingeniería Inversa de Hardware: aquella que, partiendo de un componente físico (hardware) cuyo

código es privativo (no libre, no abierto), intenta obtener un software que permita reemplazar al controlador del fabricante.

Sin embargo, no es únicamente en esta área donde la ingeniería inversa se aplica. La ingeniería inversa tiene muchísimas más aplicaciones, entre ellas:

- Ingeniería Inversa de código: aquella que va "siguiendo la pista" del código fuente de una aplicación ya sea tanto para entender la forma en la que ésta funciona como para detectar fehacientemente vulnerabilidades que no podrían ser halladas mediante herramientas para pruebas de intrusión automatizadas (que tan de moda se han puesto en los últimos tiempos pero que sin embargo, no han sido capaces hasta el momento, de sustituir ni reemplazar la inteligencia humana);

- Ingeniería Inversa de Software: ya sea para descomposición de aplicaciones (generalmente, aplicada para desentrañar un Software cerrado y privativo) o para el caso contrario, la composición de Software, a la cual nos dedicaremos en este artículo.

TDD: El ejemplo perfecto de Ingeniería Inversa de Software aplicada al desarrollo

La técnica extrema TDD *(Test-Driven Development)*, que en español se traduce como **Desarrollo guiado por pruebas**, es un claro ejemplo de cómo la Ingeniería Inversa aplicada al desarrollo de Software logra lógicas realmente simples de comprender y prácticamente unívocas e inequívocas.

Muy frecuentemente, confundimos Unit Testing (pruebas unitarias) con Test-Driven Development y Test-First Programming. Estos errores de concepto pueden generar que jamás se logre entender con exactitud, qué es la Ingeniería Inversa, cómo se aplica y para qué es necesario implementarla. Por ello, me gustaría tratar de explicar los tres conceptos con la esperanza de que por fin, queden claramente diferenciados entre sí.

En principio, podemos decir que Unit Testing, Test-First Programming y TDD, son tres técnicas donde "de menor a mayor" una es requerida por la otra. Es decir que TDD necesita de Test-First Programming y éste, de Unit Testing. Es por ello, que comenzaremos definiéndolos de "menor a mayor".

Unit Testing

Las pruebas unitarias simplemente son funciones destinadas a evaluar una única acción del código fuente. Es así que si en nuestra aplicación tenemos una función destinada a eliminar etiquetas HTML de una cadena de texto, podremos tener, en paralelo 5 métodos por ejemplo, que pasen 5 cadenas de texto diferentes a nuestra función y comprueben que el valor de retorno sea efectivamente el esperado.

Las pruebas unitarias como tales, no requieren de Ingeniería Inversa y de hecho, hasta pueden ser generadas de forma automatizada, por lo que de forma aislada, no pueden considerarse como una técnica extrema. Sin ir más lejos, las pruebas unitarias podrían escribirse una vez que la aplicación se encontrase ya desarrollada.

Test-First Programming

Esta es una técnica que consiste en escribir primero las pruebas unitarias y luego el código de la aplicación.

Esta técnica ya es un poco más compleja que el mero hecho de escribir pruebas unitarias pero sin embargo, no alcanza a ser una técnica de Ingeniería Inversa, puesto que podría facilitarse tranquilamente, el diseño de lo que se quiere lograr (es decir, "el cómo") antes de contar exactamente con el objeto (es decir, "el qué").

TDD – Test-Driven Development

El desarrollo guiado por pruebas, es pura y exclusivamente una técnica extrema de Ingeniería Inversa que plantea pensar primero en lo que se quiere lograr (es decir "el qué") y partiendo de pseudo-prototipos (que a veces no son más que el resultado *hardcodeado* de un ejemplo de lo que se desea lograr) ir implementando el mínimo código necesario para que dicho pseudo-prototipo deje de serlo. Y para escribir el mínimo código necesario, propone utilizar Test-First Programming cómo técnica para la implementación de las pruebas unitarias.

¿Por qué decimos que la técnica de TDD es Ingeniería Inversa pura?

Porque para que ésta esté bien implementada, se debe partir el desarrollo desde lo que se quiere lograr para alcanzar de a poco el cómo lograrlo.

Comprendiendo "El qué": comenzar por el resultado

A lo largo de los años he notado que lo que más le cuesta entender a todas las personas es qué es exactamente a lo que nos referimos cuando hablamos de "resultado", es decir, "qué es exactamente el qué". Y de verdad, que de tan simple que me resulta, a veces creo que esa obviedad es la que lo hace tan difícil de entender como quizás también de explicar. **El resultado es exactamente lo que esperas ver cuando hayas terminado tu aplicación y la utilices por primera vez.**

Paso a paso, un ejemplo muy sencillo y de fácil comprensión sobre cómo se implementaría la Ingeniería Inversa en el desarrollo de Software, podría ser el siguiente:

1. **Se identifica lo que se quiere alcanzar:**
   por ejemplo, un generador de formularios de contacto;

2. **Se identifica aquello que producirá nuestro Software terminado:**
   por ejemplo, un formulario HTML;

3. **Se crea un prototipo del producto de nuestra aplicación:**
   es decir, se crea un archivo HTML con un formulario de contacto a modo de ejemplo;

4. **Se implementa el mínimo código necesario para que ese formulario sea retornado:**
   por ejemplo, desde el lenguaje de programación en el que se vaya a desarrollar la aplicación, se procedería a crear una función que lea el contenido del archivo HTML y lo retorne. A continuación, una instrucción, llamaría a dicha función imprimiendo en pantalla el valor de retorno;

5. **Se procedería a repetir el paso 4, de a un ítem por vez:**
   esto se hará, hasta lograr que el formulario HTML sea generado de forma 100% dinámica;

Cuando además, se implementa **TDD** como técnica para lograr el paso 4, **la diferencia con la guía anterior, es que tras el paso 3, se aplica *Test-First Programming*** para lograr el paso 4.

Troubleshooting: Ingeniería Inversa

Este subtítulo fue puesto ex profeso a fin de que a través del humor - considerando a la Ingeniería Inversa "un problema a resolver" - se puedan plantear formas amenas y "humanas" de llevar a cabo tal técnica.

Problema: se confunde el qué con el cómo

Descripción del problema:
Generalmente, el primer problema al que uno se enfrenta es saber identificar el qué sin pensar en el cómo. Y esto, se soluciona solo y únicamente con un cambio radical de actitud.

Objetivo:

- Perderle el miedo a lo desconocido;
- Aprender a convivir con la incertidumbre y aceptar que el cómo llegará a su debido momento;
- Aceptar que lo simple es preferible a lo complejo y lo complejo es preferible a lo complicado;
- Hacerse a la idea de que todo es posible y que alcanzar objetivos no es un problema: el problema es identificarlos;

Solución:

1. Definir lo que se quiere lograr a modo de Historia de Usuario: como [usuario] puedo [acción];
2. Pensar en la acción de la Historia de Usuario como en una aplicación completa (un todo);
3. Hacerse la pregunta ¿qué voy a obtener cuando la funcionalidad esté operativa? Y responder a ella con una sola frase corta utilizando la menor cantidad de calificativos y de verbos posibles. Lo ideal es centrarse en los sustantivos.

Problema: se tiende a pensar en "el cómo" como un todo y en vez de comenzar el desarrollo por el final se comienza por el principio

Descripción del problema:
Otro de los problemas más frecuentes es que una vez hallado "el qué" no se logra implementar un desarrollo inverso y por el contrario, se comienza a escribir el código de la manera que se lo haría tradicionalmente. Esto genera un estancamiento en el desarrollo y consecuente falta de prolijidad en el código generado.

Objetivo:

- Aprender a pensar lateralmente;
- Implementar la lógica y el pensamiento lateral en paralelo para aplicar la Ingeniería Inversa de forma correcta.

Solución:

- Mientras que en pasos anteriores se definió de forma lineal lo que se pretendía lograr, ahora es momento de razonar de manera lógica para obtener el siguiente paso (como si se tratase de una secuencia a la que se quiere dar continuidad). Es decir, el siguiente paso consiste en escribir el mínimo código necesario para sustituir "algo". Ese "algo" es el que debe estar desencadenado por una secuencia lógica;
- Para encontrar dicha secuencia, una gran parte de las veces, puede ser necesario mirar "el problema" desde diversos ángulos, es decir, analizar lo que se tiene de forma lateral;
- Hallada la secuencia, se debe elegir "uno y solo un" punto de partida y concentrarse solo y únicamente en escribir el código que resuelva ese -y no otro- ítem, evitando pensar en otros ítem o en lo que se necesitará resolver más adelante.

Es importante comprender que la Ingeniería Inversa demandará hacer un uso continuo del Refactoring y tomar la no-necesidad de refactorización del código como un error de diseño prácticamente asegurado.

Python Scripting: Manipular Archivos de Configuración Mediante ConfigParser

El módulo ConfigParser de Python resultará de una gran ayuda para desarrollar scripts que nos permitan automatizar tareas generalmente relacionadas con la instalación de aplicaciones en las cuales, tanto la configuración como la optimización y aseguramiento de las mismas, cumpla un papel prácticamente protagónico.

Desde herramientas como fail2ban hasta aplicaciones como MySQL y PHP, utilizan archivos de configuración basados en secciones y, nada más usual, que la necesidad de crear *scripts* automatizados que ayuden a los administradores de sistemas o especialistas en seguridad (o Hackers Éticos si eres de os que disfruta los eufemismos *marketineros*) a configurar de forma óptima estos archivos.

Generalmente, como programadores, resolvemos esta necesidad haciendo "malabares", utilizando expresiones regulares, reemplazando archivos y un sinfín de alternativas que si bien solucionan el problema, claramente nos dan hartos dolores de cabeza.

Sin embargo, desde hace muchísimo tiempo, Python nos provee en su librería estándar un módulo cuya responsabilidad, es justamente, la de permitirnos la manipulación de archivos de configuración basados en secciones. Se trata del módulo `ConfigParser` (renombrado a `configparser` en Python 3).

El módulo ConfigParser permite tanto la lectura como edición de archivos de configuración basados en secciones. Como bien comenté al inicio, ejemplo de este tipo de archivos, podrían ser `fail2ban.conf` de fail2ban, `my.cnf` de MySQL o `php.ini` de PHP, entre otros tantos.

No obstante, `ConfigParser` no será únicamente un módulo destinado solo a un público relacionado a la administración del sistema; también puede ser una buena alternativa, para que los programadores implementen sistemas de configuración a sus propias aplicaciones. Incluso hasta puede ser sumamente útil para implementar en aplicaciones Web en las cuáles se requieran interfaces en múltiples idiomas.

En estos casos, los archivos de configuración podrían servir para definir textos estáticos en los diversos idiomas que la aplicación requiera y toda alternativa de personalización que se le quiera permitir al usuario, podría realizarse de forma sencilla, evitando el consumo de recursos producido por la implementación de bases de datos. Por ejemplo, suponiendo una aplicación Web en español, inglés y catalán, podrían tenerse tres archivos:

```
textos.es
textos.ca
textos.en
```

Y cada uno de ellos, contar con las mismas secciones y variables (reconocidas por `ConfigParser` como "opciones"), pero por supuesto, diferentes valores:

```
; Archivo textos.es

[modulos]
clientes = Clientes
proveedores = Proveedores

[acciones]
administrar = Administrar
editar = Modificar
eliminar = Eliminar


; Archivo textos.en

[modulos]
clientes = Clients
proveedores = Suppliers

[acciones]
administrar = Manage
editar = Edit
eliminar = Delete
```

Como se puede ver, la finalidad de `ConfigParser` es directamente proporcional a la cantidad de usos que se le puede dar a los archivos de configuración basados en secciones y el único límite, está en la imaginación y creatividad del programador.

Crear archivos de configuración con ConfigParser

Cuando se trata de efectuar tareas de administración y/o de seguridad, difícilmente se necesiten crear los archivos de configuración de las aplicaciones que se están optimizando. Generalmente, **la creación de estos archivos, será útil cuando se los requiera para aplicaciones propias.**

> ***La no inclusión de archivos de configuración*** *por defecto en aplicaciones propias, puede utilizarse como* ***parámetro condicional***

Por supuesto, que estos archivos podrían incluirse por defecto en la aplicación, pero la no inclusión, podría utilizarse como parámetro deductivo en la lógica de instalación: **si el archivo no existe, la instalación no se ha concluido**.

ConfigParser fue renombrado a configparser desde Python 3. La herramienta 2to3 efectúa el cambio de nombre automático en la importación del módulo, al momento de convertir el código fuente.

Un ejemplo básico de cómo crear un archivo de configuración desde `ConfigParser` aplicando una lógica deductiva, podría verse como el siguiente:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from ConfigParser import RawConfigParser
from os import path


# Si el archivo de configuración no existe
if path.exists('myapp.conf') is False:
     # Pregunto si se lo desea crear
     print 'La aplicación aún no se ha configurado correctamente'
     continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
     # Creo el archivo de configuración
     if continuar.lower() == 'y':
         cparser = RawConfigParser()  # Se crea el objeto ConfigParser
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)    # Se escribe el archivo de configuración
     else:
        exit()  # Finalizo la aplicación si el usuario eligió no concluir la instalación

print 'Continúo con la ejecución...'
```

`RawConfigParser` es el objeto de configuración básico que utiliza `ConfigParser`. Sin embargo, el objeto **`SafeConfigParser`** también se encuentra disponible:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from ConfigParser import SafeConfigParser

#...

     if continuar.lower() == 'y':
         cparser = SafeConfigParser()
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)
```

A simple vista, no se puede notar ninguna **diferencia entre `RawConfigParser` y `SafeConfigParser`**, sin embargo, este último cuenta con una característica que lo convierte en "mágico": permite la interpolación de variables en los archivos de configuración.

> *A diferencia de `RawConfigParser`,* ***`SafeConfigParser`*** ***permite la interpolación de variables*** *en los archivos de configuración*

```
; Archivo de configuración
; Ejemplo de interpolación de variables

[miseccion]
carga_maxima_permitida = 100 %(unidad_de_medida)s
unidad_de_medida = MB

; el valor de carga_maxima_permitida será 100 concatenado al valor de unidad_de_medida
; es decir: 100 MB
```

Se debe tener en cuenta que la interpolación anterior, solo será posible si se utiliza `SafeConfigParser`.

En el ejemplo anterior, nos limitamos a crear un archivo de configuración en blanco, lo cual claramente, no tendría mucho sentido. Sin embargo, completar el archivo, no es nada difícil. Basta con **agregar las secciones** con **`add_section`**`('nombre_de_la_seccion')` e **incluir variables** y valores con **`set`**`('seccion', 'variable', 'valor')`. A continuación, un sencillo ejemplo que lo aclarará todo:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from ConfigParser import SafeConfigParser
from os import path


if path.exists('myapp.conf') is False:
     print 'La aplicación aún no se ha configurado correctamente'
     continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
     if continuar.lower() == 'y':
         cparser = SafeConfigParser()
         cparser.add_section('Accesos')
         cparser.set('Accesos', 'servidor', '123.456.78.90')
         cparser.set('Accesos', 'usuario', 'pepegrillo')
         cparser.set('Accesos', 'puerto', '1001')
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)
     else:
        exit()

# ...
```

Lo anterior, generará el siguiente archivo de configuración:

```
[Accesos]
servidor = 123.456.78.90
usuario = pepegrillo
puerto = 1001
```

Vale aclarar que una forma de crear el mismo archivo de forma personalizada, podría haber sido la utilización de `raw_input` para asignar los valores de cada variable:

```
        cparser.add_section('Accesos')
        cparser.set('Accesos', 'servidor', raw_input('Servidor: '))
        cparser.set('Accesos', 'usuario', raw_input('Usuario: '))
        cparser.set('Accesos', 'puerto', raw_input('Puerto: '))
```

Obteniendo valores

Supongo que a esta altura, existiendo un método `set()`, la obviedad de que seguramente se dispone de un método `get()`, no será necesaria mencionarla. Pues como siempre sucede en Python, la lógica, el sentido común, la intuición y por sobre todo, la prolijidad y la coherencia, van de la mano.

Antes de poder obtener un valor concreto es necesario leer el archivo con el método read

Lo único que se necesita ANTES de invocar al método **`get('sección', 'variable')`** (ya sea de `RawConfigParser` como de `SafeConfigParser`) es "leer" el archivo de configuración. Y para ello, disponemos de un método **`read()`**:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


from ConfigParser import SafeConfigParser
from os import path


cparser = SafeConfigParser()

if path.exists('myapp.conf') is False:
     print 'La aplicación aún no se ha configurado correctamente'
     continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
     if continuar.lower() == 'y':
         cparser.add_section('Accesos')
         cparser.set('Accesos', 'servidor', '123.456.78.90')
         cparser.set('Accesos', 'usuario', 'pepegrillo')
         cparser.set('Accesos', 'puerto', '1001')
         with open('myapp.conf', 'wb') as archivo:
            cparser.write(archivo)
     else:
        exit()
else:
```

```
    cparser.read('myapp.conf')
    print cparser.get('Accesos', 'usuario')
```

Lo anterior (siguiendo los mismos ejemplos que al crear el archivo), arrojará `pepegrillo` como salida en caso que el archivo de configuración ya exista.

Otras opciones para obtener valores mediante `ConfigParser` son los métodos `sections()` y `options()` que retornan la lista de secciones y variables (opciones) del archivo de configuración leído, respectivamente:

```
from ConfigParser import SafeConfigParser

cparser = SafeConfigParser()
cparser.read('myapp.conf')
    for seccion in cparser.sections():
        print seccion
        print cparser.options(seccion)
```

Lo anterior, retornaría:

```
Accesos
['servidor', 'usuario', 'puerto']
```

Sin embargo, es muy probable que se desee no solo obtener las variables `(options)` sino también sus valores. Una forma de lograrlo, es mediante el método `items()`, similar a `options()` pero que agrega los valores al elemento de retorno, en forma de tuplas:

```
from ConfigParser import SafeConfigParser

cparser = SafeConfigParser()
cparser.read('myapp.conf')
    for seccion in cparser.sections():
        print seccion
        print cparser.items(seccion)


# La salida de lo anterior será:
# Accesos
# [('servidor', '123.45.678.90'), ('usuario', 'pepegrillo'), ('puerto', '1001')]
```

Finalmente, es posible que se necesite verificar las variables obteniendo errores si no se cumple con el tipo de datos esperado. Para ello, 3 métodos se encuentran disponibles: `getint()`, `getfloat()` y `getboolean()`, aunque es importante entender que **el uso de estos tres métodos, solo se justificará si se desea obtener error en caso de que el elemento solicitado no se corresponda al tipo de datos pedido**:

```
cparser.getboolean('Accesos', 'puerto')  # ValueError: Not a boolean: 1001
```

Si bien los métodos generan error cuando el tipo de datos no es el esperado, no fallan cuando la conversión previa es posible:

```
cparser.getfloat('Accesos', 'puerto')  # 1001.0
```

Agregar secciones y/u opciones

Cuando se desea agregar una o más secciones a un archivo de configuración, se realiza exactamente de la misma forma en la que se ha efectuado el agregado al momento de la creación del archivo. Esta regla aplica también, como es de esperarse, cuando lo que se desea agregar son opciones a las secciones que se están incorporando.

Sin embargo, si se quieren agregar opciones a secciones existentes, la invocación al método `read()` se hará necesaria:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


from ConfigParser import SafeConfigParser
from os import path


cparser = SafeConfigParser()

if path.exists('myapp.conf') is False:
    print 'La aplicación aún no se ha configurado correctamente'
    continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
    if continuar.lower() == 'y':
        # Se agrega nueva sección y nuevas opciones
        cparser.add_section('Accesos')
        cparser.set('Accesos', 'servidor', '123.456.78.90')
        cparser.set('Accesos', 'usuario', 'pepegrillo')
        cparser.set('Accesos', 'puerto', '1001')
    else:
        exit()
else:
    # Para agregar nuevas opciones a secciones existentes
    # primero se lee el archivo y luego se realiza la agregación
    cparser.read('myapp.conf')
    cparser.set('Accesos', 'rsa_file_path', raw_input('Ruta llave RSA: '))

with open('myapp.conf', 'wb') as archivo:
    cparser.write(archivo)
```

Analizar o evaluar el contenido de un archivo de configuración

Algo tan simple como verificar si una determinada sección u opción se encuentra presente en un archivo de configuración antes de ejecutar una instrucción concreta, es posible gracias a los métodos **has_section** y **has_option**:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


from ConfigParser import SafeConfigParser
from os import path


cparser = SafeConfigParser()

if path.exists('myapp.conf') is False:
    print 'La aplicación aún no se ha configurado correctamente'
    continuar = raw_input('¿Desea configurarla ahora? (Y/n) ')
    if continuar.lower() == 'y':
        cparser.add_section('Accesos')
        cparser.set('Accesos', 'servidor', '123.456.78.90')
        cparser.set('Accesos', 'usuario', 'pepegrillo')
        cparser.set('Accesos', 'puerto', '1001')
    else:
        exit()
else:
    cparser.read('myapp.conf')
    # Antes de agregar una nueva opción, verifica si la sección existe:
    if cparser.has_section('Accesos'):
    # Y que la opción a agregar no se encuentre ya presente
        if not cparser.has_option('Accesos', 'rsa_file_path'):
            cparser.set('Accesos', 'rsa_file_path', raw_input('Ruta llave RSA: '))

with open('myapp.conf', 'wb') as archivo:
    cparser.write(archivo)
```

Modificar valores de opciones existentes

En este caso, con tan solo "mezclar un poco de `read()` con algo de `set()`" será suficiente. Siempre que se desee modificar un valor puntual, debe primero leerse el archivo y recurrir al método `set()` para modificar el valor de una variable. Esto puede hacernos llegar a la conclusión que:

> **al invocar al método set si la opción pasada por parámetro existe, modifica su valor; en caso contrario, la crea.**

En el siguiente ejemplo, si la sección "Accesos" existe, se modificará el valor de la variable `rsa_file_path` en caso que exista o se creará la opción, en caso contrario:

```
#...
    if cparser.has_section('Accesos'):
        cparser.set('Accesos', 'rsa_file_path', raw_input('Ruta llave RSA: '))
#...
```

Eliminar secciones u opciones

Finalmente, lo que nos faltaba para poder manipular archivos de configuración de forma completa, era una

forma de eliminar secciones y/u opciones.

Para lograrlo, primero necesitaremos leer el archivo de configuración, luego, invocar a los métodos **`remove_section`**`('sección')` o **`remove_option`**`('sección', 'variable')` para eliminar una sección u opción respectivamente y por último, guardar el archivo:

```
# Elimina toda la sección Accesos
cparser.remove_section('Accesos')

# Elimina solo la opción rsa_file_path de la sección Accesos
cparser.remove_option('Accesos', 'rsa_file_path')
```

Es muy importante saber que si la sección u opción pasada a remove_section o remove_option no existe, ninguno de los dos métodos arrojará un error.

Resumen de ConfigParser

```
# Importación del módulo en Python 2.x
from ConfigParser import SafeConfigParser   # Para soporte de interpolación
from ConfigParser import RawConfigParser    # Sin soporte de interpolación

# Importación del módulo en Python 3.x
from configparser import SafeConfigParser   # Para soporte de interpolación
from configparser import RawConfigParser    # Sin soporte de interpolación


# Inicializar el parser
cparser = SafeConfigParser()  # Con soporte de interpolación
cparser = RawConfigParser()   # Sin soporte de interpolación

# Leer el archivo EXCEPTO si se lo va a crear
cparser.read('/ruta/al/archivo')

# Agregar secciones
cparser.add_section('Sección')

# Agregar o modificar opciones
cparser.set('Sección', 'Variable', 'Valor')


# Obtener valores:
#   Obtener una opción:
cparser.get('Sección', 'Variable')

#   Obtener una opción especificando el tipo de datos para ser convertido o
#   fallando en caso de error
cparser.getint('Seccion', 'variable')
cparser.getfloat('Seccion', 'variable')
cparser.getboolean('Seccion', 'variable')

#   Obtener una lista con todas las secciones
cparser.sections()
```

```
#   Obtener una lista con todas las opciones de una sección
cparser.options('Sección')

#   Obtener una lista con las opciones y valores de una sección, dentro de tuplas
cparser.items('Sección')

# Saber si una sección existe
cparser.has_section('Sección')

# Saber si una opción existe
cparser.has_option('Sección', 'variable')

# Guardar un archivo de configuración
with open('/ruta/al/archivo', 'wb') as archivo:
    cparser.write(archivo)

# Eliminar una sección
cparser.remove_section('Sección')

# Eliminar una opción
cparser.remove_option('Sección', 'variable')
```

Formato de los archivos de configuración

```
; comentarios iniciados por punto y coma
# también son ignoradas líneas iniciadas con el signo numeral

[Nombre de la Sección]
opción = valor tipo string
entero = 1000
flotante = 19.5
booleano = True

; opciones interpoladas
; requieren SafeConfigParser
variable = Se hicieron %(horas)i horas extras
horas = 12
; el valor de variable será: Se hicieron 12 horas extras
```

Ingeniería de Software: agregado de archivos cron y ejecución periódica de procesos en los paquetes .deb

Esta serie de papers que comenzó en la edición nro. 11 de la revista "Hackers & Developers Magazine" explicando cómo crear paquetes Debian y continuó en la edición anterior de The Original Hacker hablando acerca de cómo desarrollar archivos pre y post instalación propios, comienza a llegar a su fin en esta entrega, en la que me concentraré en la necesidad de nuestras aplicaciones de incluir la ejecución periódica de procesos que para correr, requieran almacenarse en los directorios /etc/cron.*

Cuando desarrollamos aplicaciones, frecuentemente incluimos instrucciones, sugerencias y hasta incluso directivas de mantenimiento en archivos README o en la documentación de nuestro Software. Sin embargo, incluso aunque el público objetivo de nuestros desarrollos sean profesionales de los sectores IT, toda instrucción, directiva o sugerencia que requiera ser ejecutada post-instalación, suele ser ignorada, sobre todo, si la misma necesita ser llevada a cabo en más de una oportunidad.

Tanto Debian como sus *distros* derivadas, son capaces de ejecutar tareas específicas de forma periódica, que van **más allá de** la programación de tareas mediante **`crontab`**.

Dichas tareas son ejecutadas por el propio Sistema Operativo, cada hora, diaria, semanal o mensualmente y las mimas, se encuentran especificadas en **guiones de *shell* ejecutables**, alojados en los directorios `/etc/cron.hourly`, `/etc/cron.daily`, `/etc/cron.weekly` y `/etc/cron.monthly`.

De allí que, por ejemplo, tareas de limpieza, depuración y mantenimiento de la base de datos de una aplicación *(como la que vimos en el artículo de emulación de tokens temporales[1] en la edición pasada)*, tranquilamente podrían efectuarse mediante un `cron.daily` - agregado al paquete `.deb` -, liberando así al usuario de tener que programar dicha tarea manualmente.

1 Artículo **"Emulación de tokens de seguridad temporales para el registro de usuarios"** (The Original Hacker Nº 1): http://library.originalhacker.org/biblioteca/articulo/ver/115

Los archivos cron.* y su formato

Los archivos `cron.*`, como comenté unos párrafos más arriba, son guiones de *shell* ejecutables. Estos archivos se alojan en las diversas carpetas `/etc/cron.*` bajo el nombre de la aplicación.

Es así, que por ejemplo, en la carpeta `/etc/cron.daily` encontraremos archivos cuyos nombres serán `apache2`, `dpkg`, `sendmail` o `apt`, entre otros.

Cada carpeta `cron.*` está destinada a almacenar guiones que se ejecutarán bajo un mismo período de tiempo. Muy intuitivamente, la relación carpeta-período es la siguiente:

```
/etc/cron.hourly        cada hora
/etc/cron.daily         diariamente
/etc/cron.weekly        semanalmente
/etc/cron.monthly       mensualmente
```

Haciendo un `cat` a cualquiera de los archivos alojados dentro de alguna de estas carpetas `cron.*`, podremos comprobar que efectivamente se trata de guiones de *shell*:

```
eugenia@cococha-gnucita:/etc/cron.daily$ cat passwd
#!/bin/sh

cd /var/backups || exit 0

for FILE in passwd group shadow gshadow; do
        test -f /etc/$FILE              || continue
        cmp -s $FILE.bak /etc/$FILE     && continue
        cp -p /etc/$FILE $FILE.bak && chmod 600 $FILE.bak
done
eugenia@cococha-gnucita:/etc/cron.daily$
```

> *Cualquier guión de shell es válido para ser agregado como archivo cron.**

Excepciones cron.d

La única **excepción** a todo lo anterior, son los archivos cuyo destino será el directorio **`/etc/cron.d.`** Los archivos alojados en este directorio, tendrán un formato diferente y además, solo se deben utilizar cuando el período de ejecución difiera de cualquiera de los anteriores.

Un claro **ejemplo de uso correcto de `cron.d`, es el archivo php5 encargado de eliminar las sesiones inactivas**. ¿A que no te lo esperabas? Sin buscarlo, de pronto encuentras la respuesta a **¿cómo hace PHP para eliminar las sesiones inactivas tras un período de tiempo determinado?**. Y la respuesta

justamente, es que **PHP agrega un archivo php5 a `/etc/cron.d`** indicando como período de ejecución (formato `crontab`) el indicado en la directiva `session.gc_maxlifetime` del archivo `php.ini` y procediendo a ejecutar la instrucción encargada de destruir toda sesión cuyo período de inactividad, supere el mencionado.

```
eugenia@cococha-gnucita:/etc/cron.d$ cat php5
# /etc/cron.d/php5: crontab fragment for php5
#  This purges session files older than X, where X is defined in seconds
#  as the largest value of session.gc_maxlifetime from all your php.ini
#  files, or 24 minutes if not defined.  See /usr/lib/php5/maxlifetime

# Look for and purge old sessions every 30 minutes
09,39 *     * * *     root   [ -x /usr/lib/php5/maxlifetime ] && [ -d /var/lib/php5 ] &&
find /var/lib/php5/ -depth -mindepth 1 -maxdepth 1 -type f -cmin +$
(/usr/lib/php5/maxlifetime) ! -execdir fuser -s {} 2>/dev/null \; -delete
```

> *Los archivos* `cron.d` *deben tener el formato requerido por* `crontab` *para poder ser ejecutados por el sistema de forma satisfactoria*

Especificaciones Debian

Como especificaciones puntuales para los paquetes Debian (`.deb`), se deben tener en cuenta sus normas. Los archivos cuyo destino sean las carpetas `/etc/cron.*` seguirán estas pautas:

- El **formato del archivo** debe ser un guión de *shell* para todos los casos excepto para `cron.d` en el que debe tener el formato sugerido por crontab;
- Los archivos deben tener **permisos de ejecución** (se debe hacer un `chmod +x` a cada archivo `.cron.*`);
- Los archivos **deben guardarse en la carpeta DEBIAN** del paquete;
- El **nombre de archivo** debe tener el formato: *`nombre_del_paquete`*`.cron.*` dónde:
  `nombre_del_paquete` será el de la aplicación. Por ejemplo, `apache2`
  `*` será `hourly`, `daily`, `weekly`, `monthly` o `d` según corresponda.

Por ejemplo, un archivo **`DEBIAN/apache2.cron.daily`** se alojará en `/etc/cron.daily/apache2` y será ejecutado diariamente.

Europio Engine Lab: ¿cómo se hicieron los ABM de la Web "The Original Hacker Library"?

En la edición anterior, vimos que crear formularios y listados con Europio Engine, era sumamente rápido y sencillo si se utilizaban los complementos disponibles para estos fines. En esta entrega, vamos a completar el laboratorio viendo como crear un ABM, puede convertirse en una tarea que se pueda ejecutar "con los ojos cerrados".

Cuando lancé la Web **The Original Hacker Library**[2], uno de los comentarios que más recibí, fue referido a la gran cantidad de trabajo que el desarrollo de la plataforma me debía haber consumido. Sin embargo, **solo tuve que invertir unos cuantos minutos**. Pero el poco trabajo, no fue producto de las "magias" de una herramienta, sino de algo tan sencillo como tener un *core* (núcleo) bien resuelto.

> ***Mientras que el núcleo de una aplicación sea** lo suficientemente **genérico** para adaptarse a cualquier requerimiento y lo necesariamente **flexible** para que lo genérico no se transforme en mágico, **toda la ciencia de un ABM es desmitificada por la propia arquitectura** de la aplicación y el diseño del Sistema.*

Para explicar como crear un sencillo ABM, tomaré como ejemplo de objeto a `Revista`, por ser un objeto muy completo: se compone de una colección de objetos `Articulo` que a la vez, se componen de un objeto `Autor`. Pasemos a explicarlo.

2 http://library.originalhacker.org

De las historias de usuario a los objetos, sus relaciones y datos anexos...

- *Como administradora puedo agregar una nueva revista*
- *Como administradora puedo agregar artículos a una revista*

De los criterios de aceptación de las Historias de Usuario, surgieron todos los objetos (aparecen subrayados y en letra cursiva):

- **Como administradora puedo agregar artículos a una *revista***
  - ....
  - Un mismo *artículo* no puede agregarse a más de una revista
  - Cada artículo puede contener un solo *autor*
  - ....

El análisis de la **relación entre los objetos**, fue sumamente simple:

- La revista tiene una colección de artículos
  - El artículo tiene un autor

Con una relación de objetos pura, ya estaba en condiciones de armar mi diagrama de clases y de paso, pensar en qué datos (más allá de las propiedades de cada objeto), podría necesitar. Y así, surgió el siguiente híbrido objetos-datos:

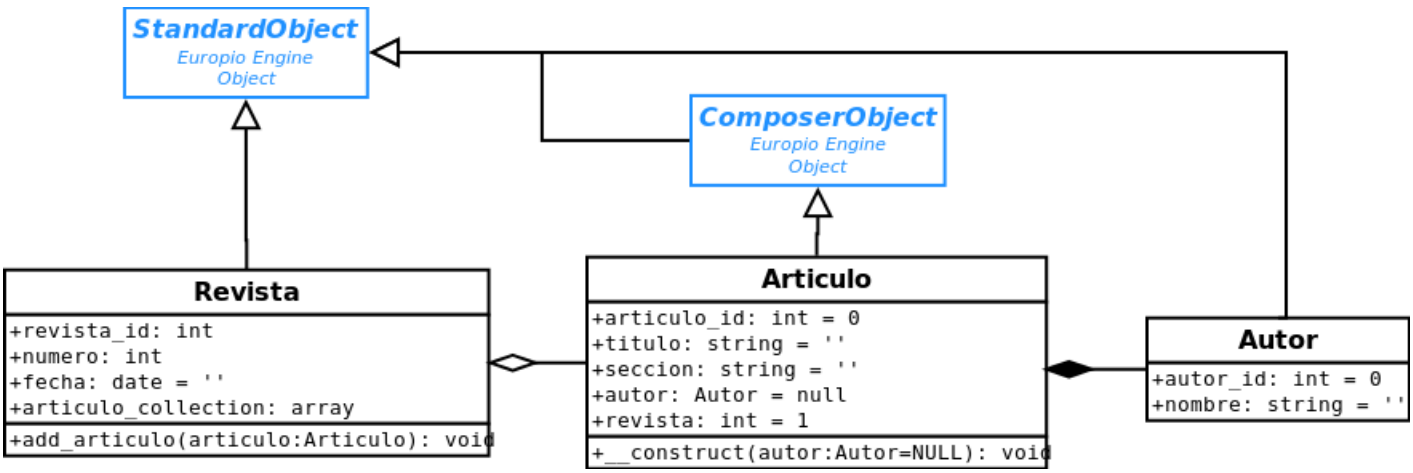


Diagrama UML realizado con **Dia** https://projects.gnome.org/dia/ (Software Libre)

`StandardObject y ComposerObject son dos objetos provistos por Europio Engine`

`StandardObject`, como su nombre lo indica, es un objeto estándar del cual heredarán la mayoría de los objetos a no ser que tengan características especiales como es el caso de los objetos de pertenencia (o compositores exclusivos), que deben heredar de `ComposerObject`.

¿Dónde obtener mayor información sobre `StandardObject` y `ComposerObject`?

- Para mayor información de **`StandardObject`** puedes leer la documentación oficial en www.europio.org/docs/standardobject.

- Lo relativo a **`ComposerObject`**, puedes encontrarlo en www.europio.org/docs/composerobject y www.europio.org/docs/many2one.

- Si necesitas obtener **información teórico-conceptual** sobre los objetos estándar y compositor, en el libro **Teoría sintáctico-gramatical de objetos**, doy una explicación detallada con ejemplos tanto en PHP como en Python. Para obtener el libro, puedes ingresar en http://www.bubok.es/libros/219288/Teoria-sintacticogramatical-de-objetos.

Creación de archivos para los modelos, las vistas y los controladores

A fin de evitar los típicos errores humanos, creé estos archivos con la CLI *(Command Line Interface)* de Europio Engine. Para ello, navegué hasta la carpeta `core/cli` y creé primero, las carpetas para el módulo al cuál llamé `biblioteca`:

```
eugenia@cococha-gnucita:~/webprojects/library/core/cli$ ./europio -c biblioteca2
Módulo biblioteca creado con éxito
eugenia@cococha-gnucita:~/webprojects/library/core/cli$
```

Y luego sí, ejecuté el comando para crear cada uno de los archivos, pasando como argumento el nombre de cada Clase principal del modelo (además del nombre del módulo):

```
eugenia@cococha-gnucita:~/webprojects/library/core/cli$ ./europio -f biblioteca Autor
Listo!
eugenia@cococha-gnucita:~/webprojects/library/core/cli$ ./europio -f biblioteca Revista
Listo!
eugenia@cococha-gnucita:~/webprojects/library/core/cli$ ./europio -f biblioteca Artículo
Listo!
```

Los modelos

Sin dudas, un paso sencillo ya que previamente había armado el diagrama de clases. Solo era necesario agregar las propiedades y los métodos de agregación en los casos que correspondiese:

```
# Archivo: appmodules/biblioteca/models/autor.php

class Autor extends StandardObject {

    public function __construct() {
        $this->autor_id = 0;
```

```
        $this->nombre = '';
    }
}


# Archivo: appmodules/biblioteca/models/articulo.php

import('appmodules.biblioteca.models.autor');


class Articulo extends ComposerObject {

    public function __construct(Autor $autor=NULL) {
        $this->articulo_id = 0;
        $this->titulo = '';
        $this->autor = $autor;
        $this->seccion = '';
        $this->revista = 0;
    }

}


# Archivo: appmodules/biblioteca/models/revista.php

import('appmodules.biblioteca.models.articulo');


class Revista extends StandardObject {

    public function __construct() {
        $this->revista_id = 0;
        $this->fecha = '';  # date
        $this->numero = 0;
        $this->articulo_collection = array();
    }

    public function add_articulo(Articulo $articulo) {
        $this->articulo_collection[] = $articulo;
    }

}
```

Una vez completos estos archivos, para concluir este paso solo restaba:

- Mapear los modelos creados;
- Generar las tablas en la base de datos;

En el caso particular de estos objetos, no sería necesario recurrir a Europio CLI para crear tablas, ya que ningún conector lógico intervendría en la relación. Así que el resultado del mapeo manual fue verdaderamente simple:

```
-- Archivo: biblioteca.sql

USE database_name;

-- tabla autor para persistencia de objetos Autor
CREATE TABLE autor (
    autor_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
```

```
  , nombre VARCHAR(80)
) ENGINE=InnoDB;


-- tabla revista para persistencia de objetos Revista
CREATE TABLE revista (
    revista_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
  , fecha DATE
  , numero INT(3)
) ENGINE=InnoDB;


-- tabla articulo para persistencia de objetos Articulo
CREATE TABLE articulo (
    articulo_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
  , titulo VARCHAR(100)
  , seccion VARCHAR(50)
  , revista INT(11) NOT NULL
  , FOREIGN KEY (revista)
      REFERENCES revista (revista_id)
      ON DELETE CASCADE
  , INDEX (revista)
  , autor INT(11)
  , FOREIGN KEY (autor)
      REFERENCES autor (autor_id)
      ON DELETE SET NULL
  , INDEX (autor)
) ENGINE=InnoDB;
```

Finalmente, importé el archivo para crear las tablas:

```
eugenia@cococha-gnucita:~/webprojects/library$ mysql -u usuario -p < biblioteca.sql
```

Completando recursos: agregar

Al fin le llegó el momento a las vistas y a los controladores. Lo primero que hice, fue crear los recursos `agregar` de cada uno de los modelos, para lo cual utilicé el plug-in `WebForm` en cada una de las vistas:

```
# Archivo: appmodules/biblioteca/views/autor.php
# Reemplazo del método agregar() de la vista de AutorView

    public function agregar() {
        $form = new WebForm('/biblioteca/autor/guardar');
        $form->add_text('nombre');
        $form->add_submit();
        print Template('Agregar Autor')->show($form->show());
    }


# Archivo: appmodules/biblioteca/views/revista.php
# Reemplazo del método agregar() de la vista de RevistaView

    public function agregar() {
        $form = new WebForm('/biblioteca/revista/guardar');
        $form->add_text('fecha', 'Fecha de publicación');
        $form->add_text('numero');
```

```
        $form->add_submit();
        print Template('Agregar Revista')->show($form->show());
    }
```

Para el caso de `ArticuloView`, modifiqué además el recurso `agregar` del controlador, a fin de que éste, le entregara a la vista las colecciones de `Autor` y `Revista` para que pudiesen mostrarse en dos listas desplegables:

```
# Archivo: appmodules/biblioteca/controllers/articulo.php
# Reemplazo del método agregar() de la vista de ArticuloController

    public function agregar() {
        $autor_collector = CollectorObject::get('Autor');
        $autores = $autor_collector->collection;
        $revista_collector = CollectorObject::get('Revista');
        $revistas = $revista_collector->collection;
        $this->view->agregar($autores, $revistas);
    }


# Archivo: appmodules/biblioteca/views/articulo.php
# Reemplazo del método agregar() de la vista de ArticuloView

    public function agregar($autores, $revistas) {
        $this->prepare_collection($autores, 'nombre');
        $this->prepare_collection($revistas, 'numero');

        $form = new WebForm('/biblioteca/articulo/guardar');
        $form->add_text('titulo');
        $form->add_text('seccion');
        $form->add_select('autor', $autores);
        $form->add_select('revista', $revistas);
        $form->add_submit();
        print Template('Agregar Artículo')->show($form->show());
    }

    private function prepare_collection(&$coleccion, $text) {
        $idproperty = strtolower(get_class($coleccion[0])) . "_id";
        foreach($coleccion as &$obj) {
            $obj->value = $obj->$idproperty;
            $obj->text = $obj->$text;
        }
    }
```

El método privado `prepare_collection()` en realidad solo lo creé para no tener que estar escribiendo código redundante y así, evitar ensuciar el método agregar.

Este método, como se puede ver, **modifica por referencia** la colección de objetos que se le pasa por parámetro, agregando dos propiedades (`text` y `value`) que son las que `WebForm::add_select` utilizará como `text` y `value` de los `option` respectivamente. Vale aclarar que lo correcto si se quisiera ser estricto, hubiese sido convertir los objetos a *arrays* mediante `settype()` y agregar `text` y `value` como claves en vez de hacerlo como propiedades, ya que por un lado, en realidad son solo datos y por el otro, `WebForm::add_select()` está preparado tanto para *renderizar* objetos como *arrays* asociativos.

El segundo parámetro que recibe este método, es el nombre de la propiedad que se utilizará para el valor del texto de cada una de las opciones de la lista desplegable.

El recurso guardar

Le toca el turno a los recursos guardar de cada uno de los modelos. Basándome en los campos de formulario que generé en los métodos anteriores, simplemente modifiqué las propiedades de cada objeto de forma directa, con lo recibido desde los formularios.

Por favor, notar que ningún input ha sido filtrado y/o verificado en el código fuente de los controladores, ya que implementé un plug-in en fase experimental, que filtra el array $_POST de forma automática. Para mayor información, leer el artículo sobre Seguridad Informática de esta edición.

Solo se completaron los recursos `guardar()` en los controladores de `Articulo`, `Autor` y `Revista`, sin ser necesario modificar ningún otro archivo. Los recursos `guardar`, finalmente, quedaron como los que siguen:

```
# Archivo: appmodules/biblioteca/controllers/autor.php

    public function guardar() {
        $id = (isset($_POST['id'])) ? $_POST['id'] : 0;  # Si es 0, agrega un nuevo objeto
                                                         # sino edita el objeto con dicha id
        $this->model->autor_id = $id;
        $this->model->nombre = $_POST['nombre'];
        $this->model->save();
        HTTPHelper::go("/biblioteca/autor/listar");  # Helper de Europio para redirecciones
    }


# Archivo: appmodules/biblioteca/controllers/revista.php

    public function guardar() {
        $id = (isset($_POST['id'])) ? $_POST['id'] : 0;
        $this->model->revista_id = $id;
        $this->model->fecha = $_POST['fecha'];
        $this->model->numero = (int)$_POST['numero'];
        $this->model->save();
        HTTPHelper::go("/biblioteca/revista/listar");
    }


# Archivo: appmodules/biblioteca/controllers/articulo.php

    public function guardar() {
    /*
       Pattern es una clase provista por Europio Engine cuya finalidad es la de proveer
       métodos estáticos que implementen diversos patrones de diseño, entre ellos, factory,
       quien creará un objeto del tipo especificado como primer parámetro y cuya ID coincida
       con el valor enviado como segundo argumento. Para mayor información, se puede acceder
       a la documentación oficial ingresando en http://europio.org/docs/pattern
    */
        $id = (isset($_POST['id'])) ? $_POST['id'] : 0;
        $this->model->articulo_id = $id;
        $this->model->titulo = $_POST['titulo'];
```

```
        $this->model->seccion = $_POST['seccion'];
        $this->model->autor = Pattern::factory('Autor', $_POST['autor']);
        $this->model->revista = (int)$_POST['revista'];
        $this->model->save();
        HTTPHelper::go("/biblioteca/articulo/listar");
    }
```

El método listar

Uno de los recursos más simples de todos es `listar`, el cual es solicitado por la redirección de los métodos `guardar()`. El recurso `listar` (en el controlador), ya fue creado por Europio CLI y no hizo falta modificar absolutamente nada. Solo fue necesario adaptar los métodos en las vistas, para lo cual, los sustituí valiéndome del plug-in `CollectorViewer` (del que hablé en la edición anterior) y el resultado fue el siguiente:

```
# Archivo: appmodules/biblioteca/views/autor.php

    public function listar($coleccion=array()) {
        # Solo si es administrador, muestra los botones editar y eliminar
        $boton = ($_SESSION['level'] == 1) ? True : False;

        $tabla = CollectorViewer($coleccion, 'biblioteca', 'autor', True,
            $boton, $boton)->get_table();
        print Template('Autores')->show($tabla);
    }


# Archivo: appmodules/biblioteca/views/revista.php
    public function listar($coleccion=array()) {
        foreach($coleccion as &$obj) {
            unset($obj->articulo_collection);  # Elimino lo que no deseo mostrar
            $revista = "The Original Hacker";
            unset($obj->numero);  # Elimino lo que no deseo mostrar
        }
        $boton = ($_SESSION['level'] == 1) ? True : False;
        $tabla = CollectorViewer($coleccion, 'biblioteca', 'revista', True,
            $boton, $boton)->get_table();
        print Template("Colección de Revistas")->show($tabla);
    }


# Archivo: appmodules/biblioteca/views/articulo.php

    public function listar($coleccion=array()) {
        $boton = ($_SESSION['level'] == 1) ? True : False;
        $tabla = CollectorViewer($coleccion, 'biblioteca', 'articulo', True,
            $boton, $boton)->get_table();
        print Template($titulo)->show($tabla);
    }
```

Recursos finales: editar y eliminar

Llegamos al final del ABM. En el caso particular del recurso `eliminar` no tendremos que hacer nada, ya que se encuentra creado en los controladores de cada modelo (de ello se encargó la CLI de Europio Engine al momento de crear los archivos). Quien nos dará un poco más de trabajo es el recurso `editar()` el cuál nos

demandará, en algunos casos, hacer pequeñas modificaciones en los controladores y, sin excepción alguna, sustituir todos los métodos homónimos en las vistas. A continuación, colocaré los códigos organizados por pares de controlador/vista para cada modelo.

```
# Archivo: appmodules/biblioteca/views/autor.php
# Autor no requiere cambios en el controlador

    public function editar($obj=array()) {  # El objeto nos lo envía el controlador
        $form = new WebForm('/biblioteca/autor/guardar');
        $form->add_hidden('id', $obj->autor_id);  # Necesitamos la ID en el formulario
        $form->add_text('nombre', null, $obj->nombre);
        $form->add_submit();
        print Template('Editar Autor')->show($form->show());
    }


# Archivo: appmodules/biblioteca/views/revista.php
# Revista no requiere cambios en el controlador

    public function editar($obj=array()) {
        $form = new WebForm('/biblioteca/revista/guardar');
        $form->add_hidden('id', $obj->revista_id);
        $form->add_text('fecha', 'Fecha de publicación', $obj->fecha);
        $form->add_text('numero', 'Número de edición', $obj->numero);
        $form->add_submit();
        print Template('Editar Revista')->show($form->show());
    }


# Archivo: appmodules/biblioteca/controllers/articulo.php

    public function editar($id=0) {
        $this->model->articulo_id = $id;
        $this->model->get();

        # Traer colección de autores
        $coleccion = CollectorObject::get('Autor');
        $autores = $coleccion->collection;

        # Traer colección de revistas
        $coleccion = CollectorObject::get('Revista');
        $revistas = $coleccion->collection;

        $this->view->editar($this->model, $autores, $revistas);
    }
```

Antes de pasar a la vista de artículo es necesario mencionar un pequeño *hack* que se me ocurrió implementar para que al mostrar el `select` de autores y revistas, aparezca seleccionado por defecto el autor y revista asignado al artículo que se esté editando.

Anteriormente, se creó un método privado llamado `prepare_collection` al que se le enviaba una colección de objetos más el nombre de la propiedad de la cual se extraería el valor a mostrar en las opciones (`option`) de la lista de desplegable (`select`). Aproveché este método para incorporar la verificación de la ID del objeto contrastándola con cada una de las ID de la colección y decidir si la propiedad "`extras`" tendría un valor vacío (para no seleccionar nada) o `' selected'` si se trataba de Ids coincidentes. El resultado de dicha modificación fue el siguiente (remarco los cambios en negritas a fin de que sea más fácil implementarlos):

```
# Archivo: appmodules/biblioteca/views/articulo.php
# Modificación del método privado prepare_collection

    private function prepare_collection(&$coleccion, $text, $id=0) {
        $idproperty = strtolower(get_class($coleccion[0])) . "_id";
        foreach($coleccion as &$obj) {
            $obj->value = $obj->$idproperty;
            $obj->text = $obj->$text;
            $obj->extras = ($obj->$idproperty == $id) ? ' selected' : '';
        }
    }
```

Por favor, notar que al tercer argumento (`$id`) le agregué un valor por omisión (cero) a fin de que no sea necesario realizar cambios a las llamadas previas a esta modificación, dado que un valor de `ID` solo es requerido para los métodos de edición pero no, para los recursos agregar.

Finalmente, el método `editar` de la vista `ArticuloView`, quedó así:

```
# Archivo: appmodules/biblioteca/views/articulo.php

    public function editar($obj, $autores, $revistas) {
        $this->prepare_collection($autores, 'nombre', $obj->autor->autor_id);
        $this->prepare_collection($revistas, 'numero', $obj->revista);
        $form = new WebForm('/biblioteca/articulo/guardar');
        $form->add_hidden('id', $obj->articulo_id);
        $form->add_text('titulo', null, $obj->titulo);
        $form->add_text('seccion', null, $obj->seccion);
        $form->add_select('autor', $autores);
        $form->add_select('revista', $revistas);
        $form->add_submit();
        print Template('Editar Articulo')->show($form->show());
    }
```

Seguridad Informática: Capas de Seguridad Inteligentes en PHP – Saneamiento del array superglobal $_POST

La siguiente publicación, corresponde a los resultados obtenidos hasta el momento, sobre una investigación que me encuentro realizando actualmente, cuyo objetivo es hallar una forma inequívoca de sanear, filtrar y limpiar datos enviados por HTTP POST, mediante una capa de seguridad de activación automática.

En varias ediciones de la revista Hackers & Developers Magazine, he dado sugerencias alternativas para el saneamiento de los datos recibidos desde formularios. En http://library.originalhacker.org se puede acceder a cada uno de los artículos sobre el tema y descargarlos en formato PDF.

Pero lo que hoy quiero mostrarles, no es en referencia a qué datos deben filtrarse, qué riesgos corremos a causa de cuáles vulnerabilidades conocidas ni qué herramientas existen para prevenir estos problemas.

Hoy, quiero hablarles sobre **una investigación que vengo llevando a cabo en los últimos meses**, mediante la cual, **valiéndome de una característica de PHP** que personalmente no logra convencerme **y de las raíces más básicas de la inteligencia artificial**, es posible **crear capas de seguridad** en nuestras aplicaciones, **que se activen de forma automática** y **que sean capaces de reconocer el tipo de información** que el programador espera y sobre ese análisis, procedan a "limpiar" el *array superglobal* $_POST de forma tal que al ser accedido por el programador, ya haya atravesado toda acción de saneamiento.

Mutabilidad del array $_POST

En el párrafo anterior comentaba acerca de una característica de PHP que aún no lograba convencerme. Sin embargo, al final de este documento, probablemente termine reconociendo lo contrario -o no-. Se trata de la **capacidad mutable del *array superglobal* $_POST**.

Uno podría esperar que se tratase de una variable de solo lectura, pero no lo es. De hecho, **$_POST no almacena los datos en crudo enviados mediante HTTP POST**, sino que lo hace habiéndolos tratado previamente. **Un claro ejemplo de esto, es lo que antiguamente sucedía con los datos cuando las famosas comillas mágicas se encontraban activadas.** De forma automática, los datos enviados mediante HTTP POST llegaban al *array superglobal* $_POST con las comillas dobles, simples, barras invertidas y valores nulos ya escapados con su barra invertida correspondiente.

Retomando el caso de mutabilidad de este *array*, la misma nos permite, por ejemplo, agregar nuevas claves al *array*, como se muestra a continuación:

```
$_POST['esto_no_era_un_campo_del_form'] = 'foo';
```

Y como era de esperarse, también nos permite modificar un valor existente:

```
$_POST['campo_del_form'] = htmlentities($_POST['campo_del_form']);
```

Esta última característica, es la que más nos interesa. Gracias a esta falta de inmutabilidad del *array*, podremos hacer que esté disponible con los datos ya saneados.

Automatización

Para que el proceso de saneamiento que se lleve a cabo se active de forma automática, éste debería ser invocado desde un archivo común a toda la aplicación, que siempre se encuentre disponible y que tenga relación directa con esta capa. Por ejemplo, una clase `Template` siempre podría estar disponible y sin embargo, no tendría ninguna relación con una capa de seguridad.

Después de mucho pensar, concluí en que debía ser la propia capa de seguridad quien se active a sí misma. De esta forma, al momento de ser importada (por ejemplo, desde un `settings`), ya estaría actuando de forma natural.

No obstante, también noté la necesidad de poder decidir la desactivación de esta capa si fuese requerido sin tener que estar comentando o eliminando la línea de importanción. Así fue que lo más coherente, me pareció que sería definir una constante *booleana* en un archivo `settings` y dependiendo de su valor, se activara o no la capa.

Disponibilidad de los datos originales

La inteligencia artificial y la lógica de negocio de una aplicación, deben siempre estar al servicio de las personas y jamás debería darse el caso contrario (aunque lamentablemente, se da en más del 80% de las aplicaciones, pero eso, ya no es tema de debate en este artículo).

Después de probar múltiples alternativas, analizarlas una a una y descubrir que ninguna me terminaba de convencer, decidí recorrer la documentación de PHP de punta a punta. Ya había descartado el uso de la variable `$HTTP_RAW_POST_DATA` puesto que dependía del valor de la directiva `always_populate_raw_post_data` del archivo `php.ini` y realmente, me resultaba poco lógico e inviable, tener que modificar esto para poder disponer de los datos HTTP POST puros. Leyendo sobre esta directiva, fue que la solución llegó a mis manos: **`php://input`**, una envoltura de **flujo de solo lectura**.

Sin embargo, cabe destacar que `php://input` cuenta con algunas restricciones interesantes, como por ejemplo, su no disponibilidad cuando el tipo de codificación del formulario es declarado como `multipart/form-data`.

`php://input NO FUNCIONA con enctype="multipart/form-data"`

Por favor, recordemos que el tipo de contenido `multipart/form-data`, es el que nos permite en un formulario, enviar grandes cantidades de datos sobre todo binarios y con codificación diferente a ASCII.
Para mayor información sobre este tema, recomiendo dirigirse a
http://www.w3.org/TR/html401/interact/forms.html#didx-multipartform-data y
http://www.ietf.org/rfc/rfc2388.txt.

A pesar de todo lo anterior, esta solución resulta la más acertada posible. Cuando `php://input` no se encuentre disponible, habría que pensar entonces, en recurrir como última alternativa, a la variable `$HTTP_RAW_POST_DAT` que podría también no estar disponible si la directiva `always_populate_raw_post_data` del archivo `php.ini` fuese `False`.

> *Para ser honesta, en PHP no existe una forma inequívoca y exacta de poder recuperar los datos enviados a través de HTTP POST sin previo tratamiento.*

Si se quisiera trabajar con `php://input`, debería hacerse mediante `fopen` y `fgets` como se muestra en el siguiente ejemplo:

```
$fp = fopen('php://input', 'r');
$data = fgets($fp);
fclose($fp);

# Donde la variable $data, podría retornar algo como:
foo=Hola+mundo%21&bar=0.65

# Equivalente a haber enviado por POST:
foo = Hola mundo!
bar = 0.65
```

Inteligencia Artificial

Para saber cómo se debería filtrar cada uno de los campos y qué filtros aplicar, podemos recurrir a las raíces más básicas de la inteligencia artificial: los diccionarios de nombres. De esta forma, podemos deducir que un campo que solicite un e-mail al usuario, podría tener como parte del nombre, la cadena "`mail`". Así, campos llamados "email", "mail_address" o "repeat_email" coincidirían con la cadena "mail". La misma lógica, podría aplicarse a otros tipos de campo:

```
IP                      ip
URL                     url, uri, site, web
Nombre de usuario       uname, username, user, usuario
Contraseña              pass (coincidiría además con passwd y password), pwd, clave
etc.
```

Claro que esta lógica no debe ser demasiado minuciosa. Es decir, la mayoría de los campos serán campos de texto (*strings*) que no van a necesitar de comprobaciones de formatos especiales.

De esta forma, creando diccionarios y aplicando a cada caso los filtros correspondientes, se podría abarcar un gran número de posibilidades que podrían sanear de forma predeterminada a decenas de miles de aplicaciones.

Implementación de la IA

Ya en muchos documentos anteriores me dediqué a hablar de los filtros que se deben aplicar en cada caso y en cada tipo de campo, así que aquí solo me concentraré en poner en código todo lo que expliqué anteriormente referido a la IA.

Dado un diccionario como el siguiente:

```
$diccionarios = array(
    'e-mail' => array('mail'),
    'contraseña' => array('pass', 'clave')
);
```

Y suponiendo un formulario con los siguientes campos y valores:

```
$_POST = array(
    'email_address' => 'foo@bar',
    'passwd' => '123456',
    'passwd2' => '123456'
);
```

El siguiente algoritmo podría utilizarse para detectar a qué tipo de dato se correspondería cada uno de los campos:

```
foreach($_POST as $key=>$value) {
    foreach($diccionarios as $tipo=>$diccionario) {
        foreach($diccionario as $entrada) {
            if(strpos($key, $entrada) !== False) {
                print "El campo {$key} es de tipo {$tipo}" . chr(10);
            }
        }
    }
}
```

Obteniendo un resultado como el siguiente:

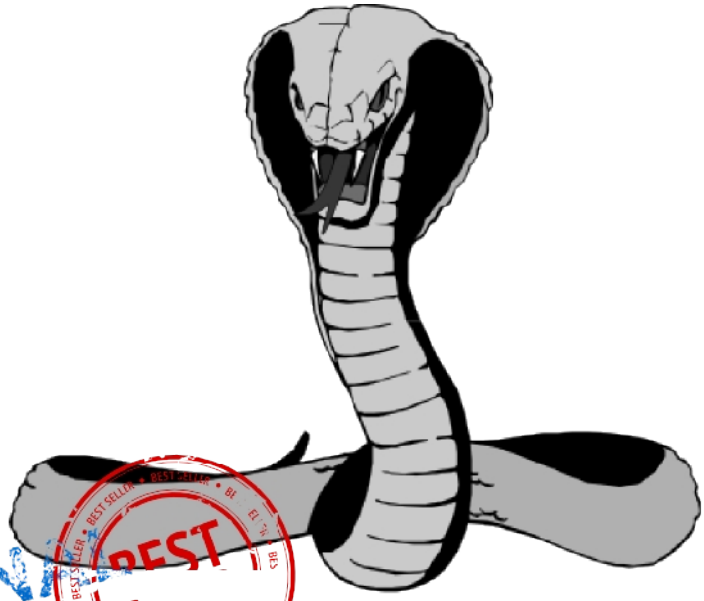
```
El campo email_address es de tipo e-mail
El campo passwd es de tipo contraseña
El campo passwd2 es de tipo contraseña
```

Luego, basados en el tipo de campo, solo es cuestión de aplicar los filtros que correspondan en cada caso.

Más información

Para conocer más sobre filtros en PHP, recomiendo leer las siguientes páginas del manual:

- Función filter_input:
  http://us3.php.net/manual/es/function.filter-input.php
- Filtros disponibles:
  http://us3.php.net/manual/es/filter.filters.php
- Función strip_tags:
  http://us3.php.net/manual/es/function.strip-tags.php
- Función htmlentities:
  http://us3.php.net/manual/es/function.htmlentities.php
- Función htmlspecialchars:
  http://php.net/manual/es/function.htmlspecialchars.php



The Original Hacker #_2_