

**Ingeniería de Sistemas de Información y Gestión**

**CPE**

CURSO : Análisis y Diseño Orientado a Objetos

PROFESOR: Eric G. Coronel Castillo

TÍTULO : Principio DRY

ALUMNO: Víctor Sotomayor Hurtado

Carlos Barreda Zúñiga

Prix Samaniego

Miraflores, Setiembre 2017

**DRY: DON´T REPEAT YOURSELF**

**NTR: NO TE REPITAS**

En esta ocasión queremos iniciar una serie de consejos de programación que consideramos importantes a nivel de eficiencia, orden y -porque no decirlo – estética. Estos consejos se basan en algunas prácticas que he observado durante los pocos años de mi carrera. Algunas de esas prácticas son buenas, mientras que otras son las que quisiera evitar. También se basan en cosas interesantes que he encontrado en uno que otro artículo por el Internet (de esos artículos donde la gente gustan hablar de temas con acrónimos de 3 o 4 letras).

Iniciaré la serie compartiendo algo acerca del principio de desarrollo de software conocido como DRY (inglés: seco), que significa Don’t Repeat Yourself. Creo que ya saben más o menos por donde va la cosa, pero antes de definir el término, haremos una pequeña introducción.

**Introducción**

Algo que hemos visto muchas veces es que cuando se quieren añadir nuevos elementos a una aplicación (ventanas, controles de usuario, funciones), y estos son muy similares a elementos ya existentes, lo que hacen los programadores es copiar y pegar el código fuente correspondiente a dichos elementos, y hacer las modificaciones en la copia de acuerdo a los nuevos requerimientos. Esta técnica es vulgarmente conocida en nuestro medio como “copy-paste”, y también es aplicable a la técnica que usan los estudiantes cuando buscan sus tareas en Wikipedia, El Rincón del Vago, Taringa, o Yahoo respuestas, en el peor de los casos.

Considero que esta una práctica muy arraigada debido a que esto se considera la forma más rápida de incluir nuevos elementos en una aplicación. Pero si se analiza cuidadosamente, al final no resulta ser la forma más rápida, porque:

* Al crear un nuevo elemento a partir de la copia de uno ya existente, implica que es necesario cambiar parte del código fuente, para cambiar o añadir la funcionalidad deseada. También hay que tener particular cuidado con aquellos parámetros que han sido “quemados” (hard-coded) dentro de la aplicación.
* El resultado es una aplicación con una gran cantidad de código fuente, lo que también deriva en la existencia de varios archivos desorganizados y desperdigados en la carpeta que contiene el proyecto.
* Si se requiere cambiar una funcionalidad principal que se ha repetido en todos los elementos copiados, esta modificación debe hacerse en cada uno de ellos.

**En relación a su definición:**

Basta de hacer catarsis, procederemos a definir los conceptos más formalmente. En la “ingeniería de software”, el término DRY, o “Don’t Repeat Yourself”, se refiere a reducir la repetición de información de cualquier tipo. El principio DRY dicta: “Cada pieza de conocimiento debe tener una representación única, no ambigua y autoritaria dentro de un sistema”. El término fue acuñado por Andy Hunt y Dave Thomas en su libro “The Pragmatic Programmer”. Eric Raymond hace referencia a un término equivalente, llamado SPOT (“Single Point Of Truth”) en su libro “The Art Of Unix Programming”. Cabemos destacar que el término no se limita a código fuente ni datos, sino a un sistema cualquiera. Tampoco se refiere necesariamente a no repetir código fuente en un sistema informático, sino que este tenga una fuente única dentro del sistema.

El término contrario a este es WET (inglés: *mojado*), al cual se le han dado muchos significados: “We Enjoy Typing”, “Write Everything Twice”, “We Edit Terribly”, etc. Obviamente, significa que existen “piezas de conocimiento” con múltiples representaciones, que podría traducirse como tener la misma información repetida varias veces.

**En relación a sus ventajas de DRY**

Ustedes se preguntarán por qué tomarse la molestia de aplicar este principio, si han vivido felizmente con la técnica de copy-paste por mucho tiempo, a tal punto que ya tienen práctica con la mano izquierda para hacer las secuencias de teclas Ctrl+C y Ctrl+V. Pues bueno, las ventajas que yo he observado de primera mano en las aplicaciones que he visto son las siguientes:

* Toda elemento (función, método, procedimiento, clase) e información dentro de la aplicación posee un origen único, por lo que si se requiere modificar la funcionalidad principal de alguno de ellos, debe hacerse en un solo punto.
* La cantidad de código fuente se reduce.
* El desglose de funcionalidades permite el manejo de múltiples capas. Esto también permite una organización más clara de los archivos de código fuente.
* El trabajo de añadir nuevos elementos basados en elementos anteriores se reduce, ya que no implicará copiar y modificar código fuente para satisfacer los nuevos requerimientos, sino más bien reutilizar elementos y funcionalidades ya existentes. En el mejor de los casos, solo será necesario invocar los elementos ya existentes, con parámetros distintos, para obtener los resultados requeridos.

**En relación a sus desventajas de DRY**

A pesar de todo, y como todo en esta vida, DRY posee algunas desventajas que hay que tener en cuenta:

* Muchas veces se requerirá de análisis profundo para encontrar una forma aceptable de centralizar todos los elementos de un sistema. La construcción de este kernel para la aplicación puede tomar algo de tiempo.
* En algunos casos, la solución centralizada será menos eficiente en tiempo de ejecución que la solución repetitiva. Un buen ejemplo de ello es cuando una consulta muy utilizada en una base de datos relacional (digamos de SQL Server) se coloca en una función de usuario para poderla usar varias veces. La función por si sola puede ser eficiente, pero al intentar utilizarla dentro de otras consultas, puede ser más lento y pesado que si la consulta no hiciera uso de ella.
* Dejar de lado el hábito de copy-paste

**¿Cómo podríamos implementar el principio DRY?**

Dicho lo anterior, ahora os describiré algunas técnicas que os pueden ayudar a implementar el concepto de DRY en vuestros programillas:

* Refinamiento del sistema en partes individuales con funcionalidades específicas y diferenciadas (técnicas top-down y bottom-up). Como me decían en la materia Programación Estructurada, “hay que reducir el problema en problemas más pequeños” (o algo así).
* Agrupar elementos y funcionalidades similares, a  nivel de archivos y directorios, así como a nivel de código fuente.
* Si se observa que una funcionalidad o elemento se repite más de una vez, o bien que es son muy parecidos a otra funcionalidad o elemento existente, entonces centralizarlos en un solo lugar. Por ejemplo, en la programación orientada a objetos, si se tienen por ejemplo las clases Profesor y Estudiante con propiedades y métodos similares, lo mejor sería que los elementos comunes los heredaran de una clase padre (Persona, por ejemplo).
* En el caso de programación orientada a objetos, hacer uso de la herencia, clases abstractas, interfaces, y otros elementos que nos permiten centralizar la funcionalidad e información en clases, y hacer uso extensivo de ellas.
* Hacer uso de Frameworks, los cuáles brindan funcionalidades comunes a toda aplicación. También incluyen generadores de código fuente, que construyen automáticamente elementos de uso común en base a características específicas. Ejemplo de ello es el Generador de Formularios del Framework para PHP llamado Symfony, que genera formularios web a partir de la definición del modelo de una base de datos.

**¿Generadores de código fuente?**

En este momento podríamos surgir la duda de por qué usar un generador de código fuente, si estos podríamos generar código repetitivo. Si bien esto es cierto, no se salen del principio de DRY, ya que en realidad el código fuente generado automáticamente ha sido construido a partir de un mismo origen: el generador de código fuente. Esto implica que si se hace uso del mismo generador, este construirá el mismo código fuente en cualquier momento si se proporcionan las mismas condiciones para su generación.

**Suceso del Caso**

Como programadores pasaremos gran parte de nuestro tiempo manteniendo software ya existente y raramente estaremos escribiendo código original. Ya sea porque estemos realizando cambios, refactorizando para mejorar el programa o arreglando un bug. Por tanto, nos conviene planificar y desarrollar nuestros sistemas pensando en ese mantenimiento desde el principio.

Consideremos un elemento de nuestro sistema con más de una representación. En algún punto durante la vida de nuestro sistema haremos una modificación de alguna de estas representaciones lo que la desfasaría del resto y puede hacer que entren en conflicto. En caso de que no sea así, significa que cada vez que cambiemos una representación de un elemento, debemos buscar el resto de representaciones y actualizarlas en consecuencia. Con forme nuestro sistema evoluciona esto deriva en tremendos dolores de cabeza a la hora de realizar cualquier modificación o mantenimiento.

No es algo que se limite solo al código, sino que debe aplicarse al sistema al completo como esquemas de base de datos, test, documentación, etc. Este es el principal problema ya que debemos expresar nuestros elementos de forma que se propaguen por todas las capas del sistema.

**Principio DRY y Ortogonalidad**

Como solución a este problema, Andy Hunt y Dave Thomas formularon en su libro “The Pragmatic Programmer” el principio DRY. Este principio establece la idea o necesidad de que los diferentes elementos del sistema deben de ser únicos, no solo en el código, si no en todas las capas o dominios del sistema.

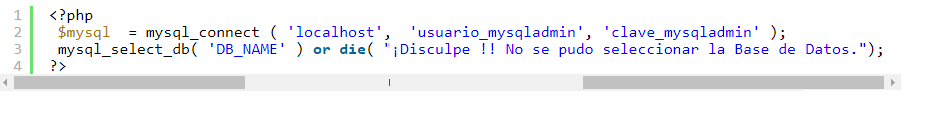
La idea de ortogonalidad se basa en que dos elementos no relacionados conceptualmente no deben de estar relacionados en el sistema. De este modo cambios en uno de los elementos no deben de provocar cambios en otros elementos *ortogonales*. Por ejemplo, un cambio en la interfaz de usuario no debe de provocar cambios en el esquema de la base de datos.

**Aquí les planteamos algunas Técnicas e implementación**

Hay muchas formas de implementar este principio (generadores de código, transformado de datos, pero todas se basan en el mismo concepto: encapsular conceptos en representaciones. Para ello se requiere una gran planificación y dividir nuestra aplicación en componentes que se relacionan. Luego podemos aplicar DRY sobre cada uno de estos componentes.

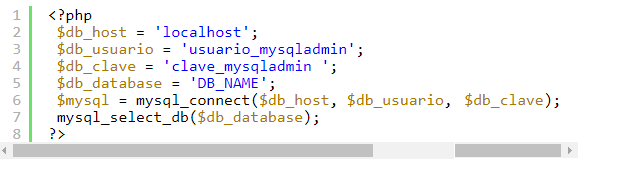
Cuando el principio DRY se aplica de forma eficiente los cambios en cualquier parte del proceso requieren cambios en un único lugar. Por el contrario, si algunas partes del proceso están repetidas por varios sitios, los cambios pueden provocar fallos con mayor facilidad si todos los sitios en los que aparece no se encuentran sincronizados.

Veamos un ejemplo práctico, es común utilizar esta codificación:



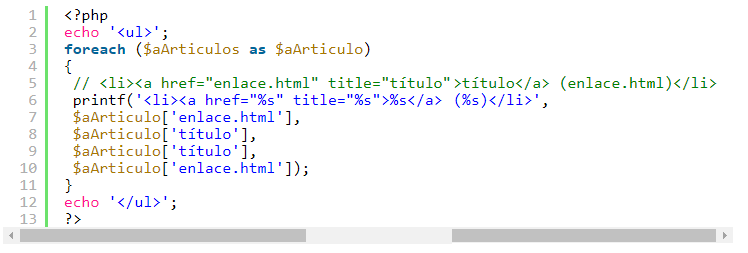
|  |  |
| --- | --- |
|  |  |

Utilizando el principio DRY, el código anterior puede reflejarse de esta forma:

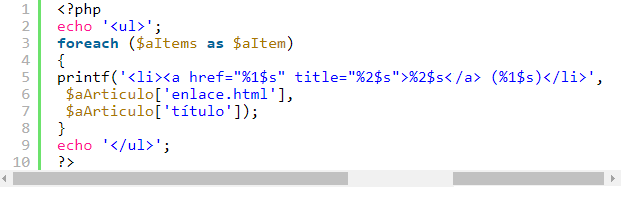


Veamos un ejemplo utilizando la función printf().

Ejemplo de codificación “incorrecta”:



El mismo código aplicando el principio DRY:



Una sugerencia de cuando podemos aplicar este principio:

1. Si se repite el mismo código varias veces en sucesión, probablemente debería estar usando un bucle.
2. Si se repite el mismo código varias veces en su programa, probablemente deba a dividir ese código a cabo en una función.
3. Si usted se encuentra escribiendo el mismo código en varios programas, probablemente debería estar usando un archivo incluido.
4. Si usted se encuentra escribiendo código complejo para hacer una tarea simple común es probable que haya una función integrada para ayudarle.
5. Si usted se encuentra escribiendo código que -. Seguramente – alguien ha hecho antes, mira las bibliotecas de recursos para el idioma que está utilizando – PEAR, PECL, CPAN, Gems… o en SourceForge.

No repitas tu código

Es un primer paso pero no debe tomarse por una solución completa a este problema. Debemos encapsular nuestro código y no ir repitiendo estructuras y funciones a lo largo de objetos y ficheros. Es algo que me he encontrado en multitud de ocasiones en las que para solucionar un problema, ya solucionado en el sistema anteriormente, se realiza el famoso “Corta-Pega” de grandes trozos de código. Muchos desarrolladores piensan que se ahorra tiempo con este tipo de soluciones o parches pero en mi experiencia acaban convirtiéndose en una fuente de problemas y pérdidas de tiempo.

Cuando te encuentres escribiendo código que es similar o igual a otro que ya has escrito antes, tomate un momento para pensar en lo que estás haciendo y no te repitas a ti mismo.

Relacionar Bases de Datos y Código: ORM

Los sistemas ORM (Object Relational Mapper) tratan de relacionar objetos o tipos de datos de los lenguajes orientados a objetos con sus representaciones en sistemas de bases de datos relacionales. Aunque su principal intención es la de facilitar la manipulación de consultas adaptándolas a la programación orientada a objetos, en algunos soluciones como [Doctrine](http://www.doctrine-project.org/) nos permiten construir y mantener el esquema de base de datos desde una entidad del código. De esa forma podemos tener una única representación de un elemento, el objeto PHP en este caso, cuyas modificaciones se sincronizan con las demás representaciones secundarias como la tabla de datos relacionada.

**Conclusión**

**El principio DRY nos presenta facilidades y ventajas con respecto a WET, ya que a la larga pueden facilitarnos el trabajo al reducir el tiempo de programación reutilizando elementos y funcionalidades ya existentes. Sin embargo, el llevar un sistema a una estructura DRY puede llevar tiempo de análisis, y muchas veces no será perfecto. Pero a pesar de esta desventaja, considero que vale la pena hacer el intento, ya que a la larga nos facilitará el trabajo a nosotros mismos quienes desarrollamos una aplicación, así como a los colegas que se encargarán de extender nuestro sistema informático en un futuro.**

**Bibliografía**

**DominandoPHP:**

[**https://dominandophp.wordpress.com/2014/07/18/tip-3-utilizar-el-principio-dry-no-te-repitas-dont-repeat-yourself/**](https://dominandophp.wordpress.com/2014/07/18/tip-3-utilizar-el-principio-dry-no-te-repitas-dont-repeat-yourself/)

**Guisho/Software Engineering:**

[**https://guisho.com/ntr-no-te-repitas-dry-dont-repeat-yourself/#more-765**](https://guisho.com/ntr-no-te-repitas-dry-dont-repeat-yourself/#more-765)

**Kroatoan:**

[**http://www.kroatoan.es/articulos/principio-software-dry/**](http://www.kroatoan.es/articulos/principio-software-dry/)

[**Souvenirs de IT**](https://itsouvenirs.wordpress.com/)

**https://itsouvenirs.wordpress.com/2012/08/16/consejos-de-programacion-1-dont-be-wet-stay-dry/**