

Clean Code

3 - Datos.

Abstracciones de información




Para TrainingIT

Por Alberto Basalo

3 - Datos, Abstracciones de información

Cohesiona variables y reduce la complejidad.

"Los malos programadores se preocupan por el código. Los buenos se preocupan por las estructuras de datos y sus relaciones."

--  **Linus Torvalds.**

Esta frase lapidaria no me atrevería a ponerla si no viniese firmada por uno de los grandes de la programación. Quizás yo la hubiera suavizado diciendo que **las estructuras de datos nos ayudan a mejorar** nuestros programas.

Pero, ¿a qué se refiere al pedirnos que nos preocupemos por las estructuras de datos? ¿No es algo que ya hacemos todos? Vamos a puntualizar. Lo que hacemos todos es usar estructuras de datos para mostrar, almacena y transmitir información relevante para el problema de negocio tratado.

Esto es imprescindible y se ha estandarizado en leyes, buenas prácticas, patrones y anti-patrones según cada cual; pues hay para elegir: documentos, normalización relacional, *DTOs*, *ActiveRecord*, *POJOs*... Efectivamente, creo que todos nos preocupamos por este tipo de estructuras.

En código limpio nos preocupamos además por dos usos de los datos con **impacto en la legibilidad y mantenimiento** del código.

Por un lado está la **cohesión de tipos primitivos** en estructuras que aporten orden y significado. El infame *code smell* "*Primitive obsession*".

Por otra parte tenemos el uso de **estructuras para simplificar condiciones lógicas** que de otro modo están *hard coded* dificultando el mantenimiento.

En cualquier caso se resuelve creando unas **estructuras muy simples**. Según el lenguaje (idioma) en el que programes puede que tengan nombre propio. Por ejemplo **struct** en C# o un **object literal** de *JavaScript*. A veces requerirán una clase para darle cuerpo; pero **nunca expondrán métodos con lógica** de negocio. Esos son otros objetos que aún no tocan en este tutorial.

Nos lo resume *Uncle Bob* en dos máximas; aquí va la primera:

"La estructura de datos expone sus propiedades y no tiene funciones significativas"

--  **Robert C. Martin**

3.1 - Cohesion de primitivos

Agrupación de variables con sentido de negocio.

"Asigna un valor de negocio a lo que son datos sueltos."

--  **Alguien que ha programado mucho**

Este tema lo he titulado en positivo "Cohesión de primitivos". Suele relacionarse negativamente con el anti patrón o *bad smell* **Primitive Obsession**.

La idea central es **reducir el uso de variables de tipos básicos**, primitivos, de tu lenguaje. En concreto booleanos, números, cadenas y fechas.

Y dirás, claro, pero es que justo esos son los tipos de datos más comunes. Lo sé; y está bien usarlos. Pero no para crear obsesivamente variables o argumentos de funciones.

Mejor emplearlos **dentro de estructuras de datos, en forma de propiedades**.



Guías

Para ayudarte he recogido una serie de consejos que te puede servir de guía. Se basan en la creación de estructuras de datos no inteligentes.

Te recuerdo la primera máxima de *Uncle Bob* al respecto.

"La estructura de datos expone sus propiedades y no tiene funciones significativas"

--  **Robert C. Martin**



Sin comportamiento de negocio: poca o ninguna función

Por lo tanto, si tu lenguaje te lo permite, debes utilizar el artificio más simple. Quizá un **struct** o un simple *Object Literal* `{ prop: value }`.



Cohesionan variables relacionadas

Cuando dos o más variables aparecen juntas al inicio de un módulo, en una función o van como argumentos... cabe preguntarse si tienen una relación.



```
let amount = 10;
let currency = 'EUR';
let price = {
  amount: 10,
  currency: 'EUR'
}
```

Suelen tener nombres de Entidades

Al ser almacenadores de datos es normal que se comporten como cualquier otra variable o clase. Por tanto las reglas de nombrado que le aplicamos es la del **sustantivo**. Pero además, en muchas ocasiones reflejan entidades del modelo de negocio.

Composición mejor que herencia









Los proyectos reales manejan una enorme cantidad de información. Y en muchas ocasiones sus datos tienen un formato similar.

En *el paradigma orientado a objetos* es tentador recurrir a la herencia para aprovechar trabajo. Pero casi nunca es buena idea. Más temprano que tarde aparecerán casos con herencias múltiples e incompatibles.

La solución recomendada para reutilizar código es **la composición de estructuras**. Es decir crear estructuras muy pequeñas, que sirvan para montar jerarquías más grandes. Consulta [el laboratorio](#) para un ejemplo.

Límites

Para terminar, intenta establecer unos límites que te ayuden a detectar problemas.

-  1 ↔ 2  *variables juntas con tipos primitivos*
-  2 ↔ 8  *propiedades por estructura*
-  1 ↔ 4  *niveles de jerarquía*
-  0 ↔ 1  *niveles de herencia*

Son rangos de confianza para examinar objetivamente el código del equipo. Pero siempre con sentido común.

"Crea muchas estructuras pequeñas, y agrúpalas en jerarquías cuando sea necesario."

--  **Alguien que ha programado mucho**

3.2 - Condiciones y algoritmos

Simplificación de algoritmos.

"Algoritmos + Estructuras de datos = Programas"

--  **Niklaus Wirth**

La lógica puede estar en los datos

Normalmente **los requisitos funcionales son complejos y volátiles**. Este tandem genera mucho ruido en el código. Manipulaciones constantes de secciones complejas son fuente de dolor de cabeza.

Hemos dedicado un tema a las [estructuras repetitivas y condicionales](#) y hemos visto cómo reducir la complejidad o al menos clarificar la intención de la lógica.

En estas regiones de código es dónde más claramente se expresa **la lógica del negocio que estamos modelando** y debemos prestarles especial atención. Pero mucho mejor sería no tener que hacerlo.

Usa estructuras de datos que eviten el uso de estructuras condicionales

Si la lógica cambia y no queremos cambiar el código; tenemos un problema. La solución pasa por reducir el uso de las **estructuras condicionales** sustituyéndolas por **estructuras de datos**.

3 La regla de 3

Este tipo de actuación en el código es **exigente en términos de destreza técnica y dominio del negocio**. Pero eso no debe echarte atrás. Simplemente quiere decir que empieces poco a poco y que lo apliques como un *refactor* cuando lo veas necesario.

Como regla para recordar, nos sugieren que optemos por la sustitución de estructuras lógicas en cuanto haya **tres modificaciones** de cualquier regla establecida. Es por eso que se recuerda como *la regla de tres*. Veamos un ejemplo.

1 No hay que anticipar nada cuando te expresan una primera regla. Por ejemplo "*Mi empresa opera en España*".





2 Puedes empezar sospechar ante un primer cambio "*Vamos a abrir también en México con estas condiciones*".

3 Pero ante el tercer caso... no dudes: **aplícate y generaliza** tu código. "*En dos meses estaremos en Colombia*"

El **if** y sobre todo el **switch** huelen mal

Como consecuencia de los cambios en las reglas de negocio tendrás que implantar o modificar muchas instrucciones con **if else** o peor aún con **switch case**. Cuanto menos toques el código mejor. Así que procura usar menos el *if* y el *switch*

- Reduce los **if** evitando  flags en las funciones, creando funciones distintas.

- Sustituye un  **switch** por un objeto, un array o un mapa y busca en él un valor o función.
- Incluso valora cambiar un  **switch** por un sistema de clases con herencia  usando la inversión de control .

Verás que poco a poco **tu código será más genérico y admitirá más cambios** funcionales sin necesidad de recompilar. Verás entonces que el mundo está llenos de estructuras de datos por todas partes.

