

Clean Code

4 - Objetos

Muchas clases pequeñas bien encapsuladas.



Para TrainingIT

Por Alberto Basalo

4 - Objetos y lógica de negocio

Muchas clases pequeñas bien encapsuladas.

"La encapsulación es importante. Pero la razón por la cual es importante es aún más importante. La encapsulación nos ayuda a razonar sobre nuestro código."

--  **Michael C. Feathers.**

Otra frase dura, aunque en este caso lo difícil es entenderla bien para luego aplicarla. Nos habla de *lógica de negocio* y usa el término *encapsular*. Supongo que todo ello unido es lo que genera incompreensión. Vayamos por partes.

Lógica de negocio

Hemos visto cómo [expresar la lógica de negocio de nuestra aplicación en funciones](#), o procedimientos o rutinas; da igual, eso es cosa del lenguaje. Pero es en esos bloques en donde reside la inteligencia. Donde escribimos **los algoritmos con sus condiciones y repeticiones**.

Si lo hacemos bien acabaremos teniendo **muchas funciones pequeñas bien nombradas**. Y nuestro siguiente reto consiste en agrupar esas funciones en módulos con algún criterio.

Clases

En programación orientada a objetos a esos módulos les llamamos clases. En otros paradigmas pueden ser *name spaces*, paquetes, librerías o simplemente módulos. De nuevo esto no es lo transcendental.

Lo importante es **el criterio que usas para agrupar las funciones**. Y aquí ya no hay recetas mágicas. Hay que conocer el negocio y aprender de la experiencia para ir ajustando el reparto de responsabilidades en clases. Esta es la razón por la cual la encapsulación es importante: porque **te obliga a razonar sobre tu desarrollo**.

Encapsulación

En esos módulos viven encerradas las funciones. Y en esos módulos viven aún mas encerrados los datos con los que operan las funciones. Esos son los objetos y esa es la otra clave de la encapsulación: **exponer lógica y proteger datos**.

SOLID

Veremos algunas claves para organizar toda esta lógica. Bajo el acrónimo **SOLID** se esconden una serie de **principios que permiten flexibilizar el mantenimiento** de los sistemas de objetos complejos.

Pero por ahora la clave de estos objetos nos la resume *Uncle Bob* en la segunda de sus dos máximas:

"Los objetos protegen sus datos detrás de abstracciones y exponen las funciones que operan con esos datos."

--  **Robert C. Martin**

4.1 - Cohesión de funciones

Agrupación de funciones con sentido de negocio.

"Al diseñar nuestras clases debemos juntar las características relacionadas, de modo que cada vez que cambien sea por la misma razón. Y deberíamos separar las características que cambian por diferentes razones."

--  **Steve Fenton**

Otra vez empezado fuerte con una frase que hay que desmenuzar para entenderla. La verdad es que casi todo hay que destriparlo para comprenderlo. Supongo que así pensaba *Jack the ripper*...

Bromas macabras aparte, cuando esta regla no se cumple los problemas y dificultades en el mantenimiento aumentan. Así que merece la pena entenderla y después aplicarla.

Distribuir con criterio

La idea es que dadas n funciones que vas a distribuir en m clases, lo hagas con este criterio: **junta en una misma clase las cambian por el mismo motivo**. Y no metas en una misma clase aquellas que cambian por otros motivos. Ejemplos de motivos pueden ser estos:

- **Cada vez que** cambie el cómo leemos o escribimos datos en un fichero.
- **Cada vez que** cambia la política de precios de nuestros productos.
- **Cada vez que** validamos temas de seguridad.
- **Cada vez que** mostramos alertas de proceso a los usuarios.

Si lo haces de esta manera, sabrás **dónde hay que tocar para realizar cada cambio**, en función del origen o tipo de cambio. Saber dónde hay que tocar es la primera parte de arreglar o modificar algo. Y **es fundamental para no repetirlo** en caso de que ya exista.

Así que vuelve a leer esta regla, asegúrate de entenderla y empieza a implantarla y difundirla.

Los objetos encapsulan La Lógica

Dado que la lógica se expresa en instrucciones dentro de las funciones. Y decimos que un módulo, clase o como le llames, es un conjunto de funciones. Pues queda claro que esos objetos encapsulan, guardan, la lógica del programa.

Usan estructuras de datos

¿Y los datos? Pues son recibidos, mantenidos, creados y enviados entre estos objetos. Son los argumentos de sus métodos públicos. Son el resultado que retornan. Son la materia prima de los objetos.

Cohesionan funciones relacionadas

Al cumplir las reglas y límites del código limpio se acaban generando muchas, muchísimas funciones. Al cumplir con la regla de *Fenton* acabamos por cohesionar esas funciones en ficheros según un criterio; y evitamos que acaben desperdigadas, o lo que es peor, repetidas.

Relacionan unas entidades con otras.

Un objeto no puede ni debe saberlo y hacerlo él todo. Por fuerza ha de delegar en otros. Instanciar e invocar métodos de otras clases es la manera de relacionarse que tienen nuestros objetos. Es decir las relaciones entre las entidades.









Interfaces mejor que herencia

Mas temprano que tarde aparecerán objetos que implanten lógica similar o incluso la misma pero en otro contexto. En P.O.O. técnicamente podremos echar mano de la herencia para reutilizar código. Pero una vez más, casi siempre va a ser una mala idea. La solución recomendada será declarar, implementar y depender de interfaces. Pero eso requiere un estudio aparte.

Límites

Cuanto más avanzas en tu destreza y conocimiento del código y su expresividad, más nos alejamos de recetas triviales. Muchos de los límites que propongo en este curso se pueden recomendar a casi cualquier código con los ojos cerrados.

Pero llegados a tocar el core de la lógica... ya hay que hilar más fino. De todas formas se pueden dar **unas recomendaciones y establecer unos indicadores** que nos alerten de si algo está yendo mal.

-  4 ↔ 16  *propiedades y métodos públicos*
-  0 ↔ 2  *argumentos por método*
-  0 ↔ 1  *niveles de herencia*
-  100 ↔ 200  *instrucciones por clase*

Una última recomendación. Si todo va bien verás que los métodos de tus clases usan con mucha frecuencia sus propios datos, es decir, sus propiedades. Desconfía de un objeto que usa demasiado o necesita saber demasiado de otros objetos.

4.2 - Principios sólidos para finales flexibles

SOLID: Principios para organizar clases.

"No caigas en la flexibilidad innecesaria."

--  **Steve Maguire-**

Este acrónimo se hizo famoso por lo rotundo de su nombre y por la vehemencia con la que lo defiende su autor. Y porque bien aplicados mejoran mucho el flexibilidad del código. Pero antes de explicarlo lanzo una advertencia; y lo repetiré más tarde.

Cuidado: recuerda el KISS  vs YAGNI 

Estos buenos principios deben aplicarse, pero no deben introducir más complejidad de la necesaria. Vamos con ellos.



S

SRP : Single responsibility principle

Principio de responsabilidad única.

Un objeto solo debería tener una única responsabilidad, o razón para cambiar.

Está íntimamente relacionado con el principio de cohesión de Fenton. En cada objeto sólo debería haber cosas que cambian por la misma razón.

 [lab SRP](#)

O

OCP : Open/closed principle

Principio de abierto/cerrado.

Las entidades de software deben estar abiertas para su extensión, pero cerradas para su modificación.

Quiere decir que el código ya escrito no debería tocarse cuando tengamos que agregar funcionalidad. Que esta siempre debería ser un extra. Este principio se incumple por ejemplo al usar un `switch`, pues un nuevo `case` implica tocar el código.

 lab OCP

L 

LSP : Liskov substitution principle

Principio de sustitución de Liskov.

Los objetos deberían ser reemplazables por subtipos sin alterar el funcionamiento del programa.

Toma su nombre de Barbar Liskov que lo anunció de manera más formal. Establece serias limitaciones al uso de la herencia. De tal forma que acaba por usarse en muy pocas ocasiones.

 lab LSP

I 

ISP : Interface segregation principle

Principio de segregación de la interfaz.

Muchas interfaces específicas son mejores que una interfaz de propósito general.

Se trata de mantener la complejidad funcional distribuida. Definiendo múltiples capacidades que se puedan implementar con pocos métodos o propiedades. De nuevo *muchas cosas pequeñas*. Lo grande será una composición de pequeñas capacidades.

 lab ISP

D 

DIP : Dependency inversion principle

Principio de inversión de la dependencia.

Depender de abstracciones, no de implementaciones concretas. Resolver en ejecución usando la Inyección de Dependencias.

Este principio es el que mayor flexibilidad aporta al código. Pero a cambio exige un mayor esfuerzo cognitivo al programador. De ahí que a veces no se aplique en su totalidad y se quede sólo en su fundamento:

Depender de abstracciones y no de implementaciones concretas.

 lab DIP

Estos cinco famosos principios en contraposición con us acrónimo SOLID, aportan flexibilidad al código. Cada uno de ellos resuelve un problema de rigidez que hace que los cambios en un programa sean más costosos

cuanto mayor es su base de código, su tamaño.

No son gratis, y no deben usarse siempre y en todo caso como una ley universal. *Mantener la sencillez está por delante de cualquier arrogancia técnica.* Si tuviera que quedarme con un par de consejos que te acercan a estos principios serían:

- Escribe clases pequeñas.
- Escribe clases más pequeñas.