Clean Code: 35 claves para dominarlo

Antonio Leiva
23 junio, 2022
15 min lectura

Comparte

Quizá has escuchado hablar mucho sobre Clean Code pero no sabes qué hacer ni por dónde empezar.

Aquí quiero darte las claves para que empieces a aplicar las enseñanzas del libro de Robert C. Martin a tus proyectos desde mañana mismo.

¿Pero tan importante es el Clean Code?

Es cierto que Robert (también conocido como Uncle Bob) muchas veces tiene más de experto en marketing que de programador, y a la vista está que el concepto de "Código Limpio" está en boca de todo programador con cierta experiencia.

Tanto, que la Clean Architecture se ha convertido en la arquitectura preferida por cientos de miles de programadores en todo el mundo. Yo mismo la enseño en mis formaciones.

Pero, en mi opinión. tras todo ese halo de publicidad y misticismo se esconden ideas que, aplicadas de forma pragmática, puede mejorar de manera importante el código ques escribes.

¿Cuál es el objetivo de escribir código limpio?

En este mismo libro se nos dice que nos pasamos más tiempo leyendo código que escribiéndolo, lo que a poco que lleves programando sabrás que es una realidad.

Por tanto, el código que escribimos no solo tiene que ser fácil de modificar, testear y

la persona que lo revisa.

Claves del Clean Code

Aunque este libro está lleno de consejos, claves y code smells, he tratado de extraer los que me parecen más importantes para que puedas empezar a aplicarlos desde hoy.

BLOQUE 1: NOMBRES

1. Usa nombres con significado

Cuando creas variables, funciones, clases... es importante utilizar nombres que aporten valor a la comunicación de lo que tu código hace.

Si al leer el nombre de un elemento no sabes para qué sirve, o necesitas poner un comentario para explicar lo que hace, entonces habrás fallado en este punto.

2. Usa nombres fáciles de pronunciar

Los seres humanos nos entendemos hablando, e instintivamente pronunciamos en nuestra cabeza lo que leemos, por lo que hay que ponérselo fácil al cerebro.

Además, imagínate debatiendo con tu compañero/a sobre un componente de la App que es imposible de pronunciar. La escena sería bastante cómica, pero poco productiva.

3. Usa nombres que puedan buscarse

Por un lado, si tienes constantes, extráelas para darles un nombre con significado.

No es lo mismo leer:

```
for(i in 0 until 7)
```

Que:

```
for (i in 0 until WEEK_DAYS)
```

Lo mismo para variables con nombres muy cortos. Si son variables locales, está bien usar nombres más acotados, pero si el ámbito de la variable es mayor, necesitamos algo más mejor.

4. Nombres de clases y métodos (o funciones)

Los nombres de clases deben ser nombres, mientras que las funciones o métodos deben ser verbos.

Mientras que las primeras representan un elemento que en mayor o menor medida refleja un objeto del mundo real, las segundas son acciones que esos objetos realizan.

5. Elige una sola palabra por concepto

Si realiza una tarea similar o tiene una responsabilidad parecida en tu código, debería usar el mismo concepto.

No llames a unas clases *Manager*, a otras *Controller* y a otras *Driver* si todas ellas solucionan un problema similar.

BLOQUE 2: FUNCIONES

6. Las funciones deben ser pequeñas

Cuanto más largas sean, más posible es que esté incumpliendo el Principio de Responsabilidad Única.

Se vuelve más difíciles de leer, de modificar, de testear...

En el libro nos recomiendan que sean muy (muy (muyyy)) cortas, de unas 3-4 líneas.

A mí esto personalmente me parece inviable, pero ponte un límite razonable de máximo 50-60 líneas.

Otra forma de limitar la longitud es que la indentación sea máximo de 1 ó 2 niveles. Si va a ser más, mejor extraer esa parte a otra función.

En cuanto a la longitud de las líneas, no uses más de 100-120 caracteres. Muchos IDEs permiten configurar una línea vertical para ayudarte con ello.

7. Haz una única cosa

Esto está muy relacionado con lo que hablábamos antes, así que no me voy a enrollar.

Las funciones deberían hacer una única cosa y hacerla bien.

Relacionado con esto, si tu función tiene más de un nivel de abstracción, entonces mejor divídela.

Nivel de abstracción se refiere a cómo de alto o bajo nivel es el código. No es lo mismo llamar a una función con un nombre `save(myObject)` que directamente comunicarse con el API del sistema de archivos.

8. No abuses de los switch/when

Es muy difícil que una función con un control de este tipo haga una única cosa.

Si necesitas usarlos, usa polimorfismo para ocultarlo y no tener que repetirlo si es necesario volver a usarlo.

A mí personalmente me parece excesivo hacer esto siempre, pero me parece interesante tratarlos como un "code smell", y al menos plantearme si existe una solución mejor.

9. ¿Cuántos argumentos debe tener una función?

En el libro se nos indica que el número ideal es 0, y que a partir de 3 argumentos, deberíamos evitarlo si es posible.

Si una función tiene muchos argumentos, es difícil razonar sobre ella, ya que hay muchas combinaciones de argumentos de entrada, y es díficil comprender qué va a ocurrir con cada posible combinación.

Esto llevado al testing es lo mismo: hay muchas posibilidades, y por tanto probar la función con todos sus posibles estados se puede volver inmanejable.

Aquí yo añadiría que los constructores de una clase que representa un estado son una excepción, y puede tener sentido que tengan bastantes más argumentos.

10. Evita los "flag arguments"

Para empezar, porque esto ya significa que la función está haciendo 2 cosas, y no una.

Y para seguir, porque a nivel de comprensión, leer por ejemplo una función que sea `render(true)` no nos explica nada.

Es mejor tener 2 funciones que no reciban argumento y que su nombre indique exactamente lo que hacen.

11. No generes “Side Effects” o efectos colaterales

Si una función pide algo pero, además, hace algo que no está representado en el nombre de la función, nos está ocultando información.

Imagina que tienes una función que pide un listado de elementos pero también los guarda en una base de datos, o modifica esos datos de alguna forma.

Pueden ocurrir cosas inesperadas de las que la función no nos está informando.

Por lo general, una función se debe encargar de hacer algo o devolver algo, pero nunca ambas cosas.

12. Don’t Repeat Yourself (No te repitas)

Un principio muy conocido dentro de la programación. Repetir código implica que si la lógica de ese código cambia, nos tenemos que acordar de modificarla en todas las partes donde se encuentre la duplicación.

Extrae el código en una función y reutilízalo donde corresponda.

BLOQUE 3: COMENTARIOS

13. Los comentarios mienten

El gran problema que tienen los comentarios es que no compilan, por lo que si el código asociado se modifica y el comentario queda obsoleto, nadie se enterará.

A partir de ese momento, efectivamente el comentario miente, y por tanto quien lo lea tendrá un conocimiento equivocado del código. Es casi mejor no entender el código que

14. Usa código autoexplicativo

En vez de crear un comentario que explique lo que el código hace, ¿por qué no escribes el código de tal forma que él mismo explique lo que hace?

Muchas veces es tan sencillo como dar nombres más descriptivos, y crear funciones con buenos nombres que encapsulen detalles del código.

Siempre que necesitemos un comentario para explicar un código confuso, plantéate si no puedes mejor reescribir el código para que se entienda.

15. A veces los comentarios son necesarios

Para algunos casos sí tiene sentido escribir comentarios.

Si por ejemplo necesitas escribir un código un poco rebuscado que no eres capaz de resolver de otra forma, o te encuentras trabajando con una API o framework que obliga a hacer algo de una manera muy especial, o tienes que resolver algún problema externo a tu código con un hack (los programadores Android seguro que han pensado en Samsung ?), estas son situaciones razonables.

16. Los comentarios dicen qué hace el código, no cómo lo hace

A modo de resumen, si tienes un comentario que dice qué pasos se están dando en el código, ese comentario no debería existir.

Un comentario debería explicar por qué se ha tomado cierta decisión que pueda generar cierta controversia, o que a simple vista no sea evidente.

BLOQUE 4: OBJETOS Y ESTRUCTURAS DE DATOS

17. Diferencias entre objetos y estructuras de datos

Eso implica que normalmente los objetos de un tipo se tratarán de una forma y los segundos de otra. No deberían mezclarse intentando crear estructuras de d vez conozcan lógica de operación

18. La Ley de Demeter

Esta ley nos dice que un módulo no debería conocer las tripas de los objetos que manipula.

No voy a entrar en detalle, porque tengo un artículo hablando de la Ley de Demeter.

BLOQUE 5: MANEJO DE ERRORES

19. Usa excepciones en lugar de código de retorno

Las excepciones ocultan menos el código que realmente importa, porque se pueden lanzar fácilmente en cualquier momento, y es más fácil que el código que llama a esta función no se olvide de gestionar el error (al fin y al cabo la App explotará si no se maneja)

20. Escribe primero el *try-catch-finally*

Cuando escribes un código de este tipo, estás asumiendo que en cualquier momento el código se puede cortar y saltar al `catch`. Por tanto, en el `catch` necesitas dejar tu programa en un estado consistente

21. Usa excepciones *unchecked*

En lenguajes como Kotlin esto es una obviedad, porque no existen, pero por ejemplo en Java sí que podemos crear *checked* exceptions (no sé muy bien cómo traducir esto ?).

Las checked exceptions obligan a ser capturadas y a que cada función que pueda devolver esta excepción se marque con un `throws`.

Esto implica que, si la excepción se está propagando mucho por nuestro código,

22. No devuelvas Null

En lenguajes que no son capaces de gestionar la nulidad (Kotlin vuelve a salvarse en este caso), devolver nulos es muy peligroso, porque el que llama a esa función no tiene por qué saber si lo que llega es un nulo o no, y de forma defensiva puede acabar con todo su código con chequeos de nulos.

O eso, o nuestro código se inundará de `NullPointerException`s.

BLOQUE 6: TESTS UNITARIOS

Teniendo claros que los tests son vitales (y si aún no lo tienes claro, te recomiendo que te descargues esta guía), vamos a ver varias reglas que aplican a ellos.

Ten en cuenta que Uncle Bob es un ferviente defensor del TDD (Test Driven Design), así que si tú no, pueden chocarte algunas reglas.

23. Las tres leyes del TDD

1. No puedes escribir código de producción hasta que hayas escrito un test que falla

2. No puedes escribir más de un test unitario que falle (y no compilar = fallar)

3. No debes escribir más código de producción del necesario para pasar el test que fallaba

24. Mantén limpios los tests

Los tests deben cambiar mietnras el código evoluciona. Cuanto más desordenados estén los tests, más difíciles son de cambiar.

Ten siempre en mente que el código de los tests es tan importante como el código de producción.

La mayor parte de las reglas de código limpio aplican también a los tests. En los tests, la legibilidad es tanto o más importante que en código de producción, porque a veces sirven como documentación del código.

26. Un *Assert* por test

Aunque no es una regla escrita en piedra, es importante que cada test compruebe una única cosa. Si no, si el test falla, no tendremos claro el motivo.

27. Un único concepto por test

Relacionado con lo anterior, cada test debería probar una única cosa. Si prueba más, no estamos seguros de que estemos cubriendo bien todos los casos.

Y de ser así, necesitaremos varios *Assert* que lo confirmen.

28. La regla F.I.R.S.T

Un test debe ser:

- Rápido (Fast)
- Independiente, para que no importe el orden de ejecución
- Repetible: que se puedan repetir en cualquier entorno.
- Auto-validable (Self-Validating): Los tests fallan o pasan, no deberíamos tener que comprobar nada extra para asegurarnos de que fue correcto.
- Oportuno (Timely): deberían ser escritos justo antes del código de producción. Si lo escribes depués, el código puede ser difícil de testear.

BLOQUE 7: CLASES

El contenido de las clases debería seguir esta estructura, siempre suponiendo que tengamos elementos de ese tipo:

1. Constantes públicas

2. Constantes privadas

3. Variables públicas (rara vez son una buena idea)

4. Variables privadas

5. Funciones públicas

6. Las funciones privadas que son llamadas por las públicas irían justo debajo de la pública que las llama.

30. Las clases deberían ser pequeñas

Las clases deben estar enfocadas en su principal objetivo, y por tanto no deberían crecer indiscriminadamente.

Una forma de detectar esto, es que no podamos encontrarle un nombre claro y conciso a la clase. También deberíamos poder escribir una breve descripción sin necesidad de usar palabras como "si", "y", "o", "pero"...

31. Principio de Responsabilidad Única

Muy relacionado con el anterior, el Single Responsibility Principle, nos dice que una clase o módulo debería tener una, y solo una, razón para cambiar.

Tengo un artículo exclusivo sobre el Principio de Responsabilidad Única.

32. Cohesión

Las clases deberían tener un número pequeño de variables de instancia. Cada uno de los métodos de la clase debería manipular una o varias de esas variables.

Una clase en la que cada variable se usa en cada método tendría la máxima cohesión.

33. Organiza tu código para prepararlo para el cambio

Las clases solo deberían tener una razón para cambiar, si tienen más implica que estamos incumpliendo el Principio de Responsabilidad única.

Tan pronto como nos encontremos abriendo una clase, debería considerar mejorar el diseño.

El Principio Open-Closed (del que también tengo otro artículo) nos ayuda mucho en este aspecto.

Otra idea vital es la de no depender de detalles de implementación, ya que si esos detalles cambian, harán que nuestro código cambie. Para minimizar este acoplamiento, podemos apoyarnos en el Principio de Inversión de Dependencias.

34. Separa la construcción de un sistema de su uso.

Hay una responsabilidad grande en el software que es el de la construcción (o instanciación) de los objetos que van a formar parte del sistema.

Muchas veces mezclamos la creación de esos objetos con el código que utiliza esos objetos.

Esto aumenta el acoplamiento que esos dos objetos tienen entre sí, y no podemos trabajar con interfaces, lo que nos limita tanto en la modificación de ese código como en el testing.

Para lograr esto, necesitamos aplicar tanto el Principio de Inversión de Dependencias del que ya hablamos como una herramienta de inyección de dependencias que cree y provisione esas dependencias.

35. Utiliza copias de objetos para trabajar con concurrencia

A no ser que, por alguna razón, necesites compartir memoria entre hilos, lo ideal cuando trabajas con concurrencia es que el estado con el que trabajen sea inmutable.

Inmutable quiere decir que no se pueda modificar, que todas sus variables sean finales.

afectados.

Conclusión

Aplicar "Clean Code" a tu código de forma sistemática y sin valorar su utilidad puede llevarte a crear un código con una sobreingeniería tal que sea peor el remedio que la enfermedad.

Pero es interesante tener todos estos conceptos en mente cuando estamos escribiendo código para ir aplicándolos cuando creamos que tiene sentido.

Esto solo cubre más o menos la mitad del libro. La segunda mitad trata sobre olores de código (o code smells) que pueden alertarte de que no estás escribiendo código limpio.

Si quieres que lo trate en un artículo futuro, puedes votarlo en la lista de Klistt.

Recursos de expertos para la solución de problemas

Ver todos

Principios SOLID: Qué son, cuáles, y qué beneficios aporta usarlos

Cápsulas Curso Arquitecturas Gratis

Patrones de diseño de software

Cápsulas Curso Arquitecturas Gratis

Cómo convertirse en un desarrollador de software: habilidades y conocimientos necesarios

Desarrollo Profesional Tips

Test Driven Development [TDD] - Qué es y cómo aplicarlo

DevExpert

Guía gratuita

Únete a nuestra comunidad y acelera tu crecimiento profesional con la ayuda de expertos.

Email

Unirme

Acepto la Política de Privacidad*