

CLEAN CODE, SOLID Y TESTING

APLICADO A JAVASCRIPT

A large, bold, black 'JS' logo, where the letters are formed by thick, rounded strokes. The logo is centered on a yellow-to-black gradient background.

CONTIENE EJEMPLOS INTERACTIVOS EN
JAVASCRIPT Y EN TYPESCRIPT

MIGUEL A. GÓMEZ

Clean Code, SOLID y Testing aplicado a JavaScript

Contiene ejemplos en REPL online interactiva tanto en JavaScript como en TypeScript

Software Crafters

Este libro está a la venta en <http://leanpub.com/cleancodejavascript>

Esta versión se publicó en 2019-10-27



Este es un libro de [Leanpub](#). Leanpub anima a los autores y publicadoras con el proceso de publicación. [Lean Publishing](#) es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener feedback del lector hasta conseguir tener el libro adecuado.

© 2019 Software Crafters

Índice general

OFERTA DE LANZAMIENTO	1
Prefacio	2
Qué no es este libro	2
Agradecimientos	3
Sobre Software Crafters	4
Sobre el autor	5
Otros intereses	5
Introducción	7
Deuda técnica	9
Tipos de deuda	10
Refactoring, las deudas se pagan	10
Mejor prevenir que curar, las reglas del diseño simple	10
SECCIÓN I: CLEAN CODE	11
¿Qué es Clean Code?	12
Variables y nombres	14
Nombres pronunciables y expresivos	15
Uso correcto de <i>var</i> , <i>let</i> y <i>const</i>	15
Ausencia de información técnica en los nombres	15
Léxico coherente	15
Nombres según el tipo de dato	15

ÍNDICE GENERAL

Arrays	15
Booleanos	15
Números	15
Funciones	16
Clases	16
Funciones	17
Tamaño y función	17
Número de argumentos	17
Estilo declarativo frente al imperativo	17
Funciones anónimas	17
Transparencia referencial	17
Principio DRY	17
Evita el uso de comentarios	18
Formato coherente	19
Problemas similares, soluciones simétricas	19
Densidad, apertura y distancia vertical	19
Lo más importante primero	19
Indentación	19
Clases	20
Tamaño reducido	20
Organización	20
Prioriza la composición frente a la herencia	20
SECCIÓN II: PRINCIPIOS SOLID	21
Introducción a SOLID	22
De STUPID a SOLID	23
¿Qué es un <i>code smell</i> ?	23
El patrón singleton	23
Alto acoplamiento	23
Acoplamiento y cohesión	23
Código no testeable	24
Optimizaciones prematuras	24

ÍNDICE GENERAL

Complejidad esencial y complejidad accidental	24
Nombres poco descriptivos	24
Duplicidad de código	24
Duplicidad real	24
Duplicidad accidental	24
Principios SOLID al rescate	25
SRP - Principio de responsabilidad única	26
¿Qué entendemos por responsabilidad?	26
Aplicando el SRP	26
Detectar violaciones del SRP:	26
OCP - Principio Abierto/Cerrado	27
Aplicando el OCP	27
Patrón adaptador	28
Detectar violaciones del OCP	31
LSP - Principio de sustitución de Liskov	32
Aplicando el LSP	32
Detectar violaciones del LSP	32
ISP - Principio de segregación de la interfaz	33
Aplicando el ISP	33
Detectar violaciones del ISP	33
DIP - Principio de inversión de dependencias	34
Módulos de alto nivel y módulos de bajo nivel	34
Depender de abstracciones	36
Inyección de dependencias	36
Aplicando el DIP	36
Detectando violaciones del DIP	36
Introducción al testing	37
Tipos de tests de software	39
¿Qué entendemos por testing?	39
Test manuales vs automáticos	39

ÍNDICE GENERAL

Test funcionales vs no funcionales	39
Tests funcionales	39
Tests no funcionales	39
Pirámide de testing	39
Antipatrón del cono de helado	39
Tests unitarios	40
Características de los tests unitarios	41
Anatomía de un test unitario	41
Jest, el framework de testing JavaScript definitivo	42
Características	42
Instalación y configuración	42
Nuestro primer test	43
Aserciones	43
Organización y estructura	43
Gestión del estado: before y after	43
Code coverage	43
TDD - Test Driven Development	44
Las tres leyes del TDD	44
El ciclo Red-Green-Refactor	44
TDD como herramienta de diseño	44
Estrategias de implementación, de rojo a verde.	44
Implementación falsa	45
Triangular	45
Implementación obvia	45
Limitaciones del TDD	45
TDD Práctico: La kata FizzBuzz	46
Las katas de código	46
La kata FizzBuzz	46
Descripción del problema	46
Diseño de la primera prueba	46
Ejecutamos y... ¡rojo!	46
Pasamos a verde	47
Añadiendo nuevas pruebas	47

ÍNDICE GENERAL

Refactorizando la solución, aplicando pattern matching.	47
Referencias	48
OFERTA DE LANZAMIENTO	50

OFERTA DE LANZAMIENTO

Hasta final de año, diciembre de 2019, el e-book tendrá un descuento del 25%. No dejes escapar la oportunidad, estoy convencido de que te puede aportar algún detalle de mucho valor.

Recuerda que si no es lo que esperas, te devolvemos tu dinero. Durante los primeros 45 días de compra, puedes obtener un reembolso del 100%. El riesgo es cero y el beneficio podría ser muy elevado.

Puedes adquirir el e-book desde aquí.¹

¹<https://softwarecrafters.io/cleancode-solid-testing-js>

Prefacio

JavaScript se ha convertido en uno de los lenguajes más utilizados del mundo, se encuentra en infraestructuras críticas de empresas muy importantes (Facebook, Netflix o Uber lo utilizan).

Por esta razón, se ha vuelto indispensable la necesidad de escribir código de mayor calidad y legibilidad. Y es que, los desarrolladores, por norma general, solemos escribir código sin la intención explícita de que vaya a ser entendido por otra persona, ya que la mayoría de las veces nos centramos simplemente en implementar una solución que funcione y que resuelva el problema. La mayoría de las veces, tratar de entender el código de un tercero o incluso el que escribimos nosotros mismos hace tan solo unas semanas, se puede volver una tarea realmente difícil.

Este pequeño e-book pretende ser una referencia concisa de cómo aplicar *clean code*, *SOLID*, *unit testing* y *TDD*, para aprender a escribir código JavaScript más legible, mantenable y tolerante a cambios. En este encontrarás múltiples referencias a otros autores y ejemplos sencillos que, sin duda, te ayudarán a encontrar el camino para convertirte en un mejor desarrollador.

Qué no es este libro

Antes de comprar este e-book, tengo que decirte que su objetivo no es enseñar a programar desde cero, sino que intento exponer de forma clara y concisa cuestiones fundamentales relacionadas con buenas prácticas para mejorar tu código JavaScript.

Agradecimientos

Este es el típico capítulo que nos saltamos cuando leemos un libro, a pesar de esto me gusta tener presente una frase que dice que **no es la felicidad lo que nos hace agradecidos, es agradecer lo que nos hace felices**. Es por ello que quiero aprovechar este apartado para dar las gracias a todos los que han hecho posible este e-book.

Empecemos con los más importantes: mi familia y en especial a mi hermano, sin lugar a dudas la persona más inteligente que conozco, eres un estímulo constante para mí.

Gracias amiga especial por tu apoyo y, sobre todo, por aguantar mis aburridas e interminables chapas.

Dicen que somos la media de las personas que nos rodean y yo tengo el privilegio de pertenecer a un círculo de amigos que son unos auténticos cracks, tanto en lo profesional como en lo personal. Gracias especialmente a los Lambda Coders [Juan M. Gómez](#)², Carlos Bello, Dani García, [Ramón Esteban](#)³, [Patrick Hertling](#)⁴ y, como no, gracias a [Carlos Blé](#)⁵ y a Joel Aquiles.

También quiero agradecer a Christina por todo el esfuerzo que ha realizado en la revisión de este e-book.

Por último, quiero darte las gracias a ti, querido lector, (aunque es probable que no leas este capítulo) por darle una oportunidad a este pequeño libro. Espero que te aporte algo de valor.

²https://twitter.com/_jmgomez_

³<https://twitter.com/ramonesteban78>

⁴<https://twitter.com/PatrickHertling>

⁵<https://twitter.com/carlosble>

Sobre Software Crafters

Software Crafters⁶ es una web sobre **artesanía del software, DevOps y tecnologías software** con aspiraciones a plataforma de formación y consultoría.

En Software Crafters nos alejamos de dogmatismos y entendemos la artesanía del software, o software craftsmanship, como un mindset en el que, como desarrolladores y apasionados de nuestro trabajo, tratamos de generar el máximo valor, mostrando la mejor versión de uno mismo a través del aprendizaje continuo.

En otras palabras, interpretamos la artesanía de software como un largo camino hacia la maestría, en el que debemos buscar constantemente la perfección, siendo a la vez conscientes de que esta es inalcanzable.

⁶<https://softwarecrafters.io/>

Sobre el autor

Mi nombre es Miguel A. Gómez, soy de Tenerife y estudié ingeniería en Radioelectrónica e Ingeniería en Informática. Me considero un artesano de software (**Software Craftsman**), sin los dogmatismos propios de la comunidad y muy interesado en el desarrollo de software con [Haskell](#)⁷.

Actualmente trabajo como **Senior Software Engineer** en una *startup* estadounidense dedicada al desarrollo de soluciones software para el sector de la abogacía, en la cual he participado como desarrollador principal en diferentes proyectos.

Entre los puestos más importantes destacan **desarrollador móvil multiplataforma con Xamarin y C#** y el de desarrollador **fullStack**, el puesto que desempeño actualmente. En este último, aplico un estilo de programación híbrido entre orientación a objetos y [programacion funcional reactiva \(FRP\)](#)⁸, tanto para el **frontend** con [Typescript](#)⁹, [RxJS](#)¹⁰ y [ReactJS](#)¹¹, como para el **backend** con [Typescript](#), [NodeJS](#), [RxJS](#) y [MongoDB](#), además de gestionar los procesos [DevOps](#)¹² con [Docker](#) y [Azure](#).

Por otro lado, soy cofundador de la start-up [Omnirooms.com](#)¹³, un proyecto con el cual pretendemos eliminar las barreras con las que se encuentran las personas con movilidad reducida a la hora de reservar sus vacaciones. Además, soy fundador de [SoftwareCrafters.io](#)¹⁴, una web sobre artesanía del software, DevOps y tecnologías software con aspiraciones a plataforma de formación y consultoría.

Otros intereses

Además del desarrollo de software y el emprendimiento, estoy muy interesado en el mundo de las finanzas y de los mercados. Mi filosofía de inversión se basa en el [value](#)

⁷<https://www.haskell.org/>

⁸https://en.wikipedia.org/wiki/Functional_reactive_programming

⁹<https://softwarecrafters.io/typescript/typescript-javascript-introduccion>

¹⁰<https://softwarecrafters.io/javascript/introduccion-programacion-reactiva-rxjs>

¹¹<https://softwarecrafters.io/react/tutorial-react-js-introduccion>

¹²<https://es.wikipedia.org/wiki/DevOps>

¹³<https://www.omnirooms.com>

¹⁴<https://softwarecrafters.io/>

investing¹⁵ combinado con principios de la **escuela austriaca de economía**¹⁶.

Por otro lado, siento devoción por el **estoicismo**¹⁷, una filosofía que predica ideas vitales como vivir conforme a la naturaleza del ser humano, profesar el agradecimiento, ignorar las opiniones de los demás, centrarse en el presente o mantenerse firme ante las adversidades, siendo consciente de lo que está bajo nuestro control y lo que no. Aunque, como Séneca, no siempre predico con el ejemplo.

También practico entrenamientos de fuerza con patrones de movimiento en los que se involucra todo el cuerpo de manera coordinada. Soy aficionado al **powerlifting**¹⁸ y al **culturismo natural**¹⁹, motivo por el cual le doy mucha importancia a la nutrición. He tratado de incorporar a mis hábitos alimenticios un enfoque evolutivo, priorizando la comida real (procedente de ganadería y agricultura sostenible siempre que sea posible) y evitando los alimentos ultraprocesados.

¹⁵https://en.wikipedia.org/wiki/Value_investing

¹⁶https://es.wikipedia.org/wiki/Escuela_austriaca

¹⁷<https://es.wikipedia.org/wiki/Estoicismo>

¹⁸<https://en.wikipedia.org/wiki/Powerlifting>

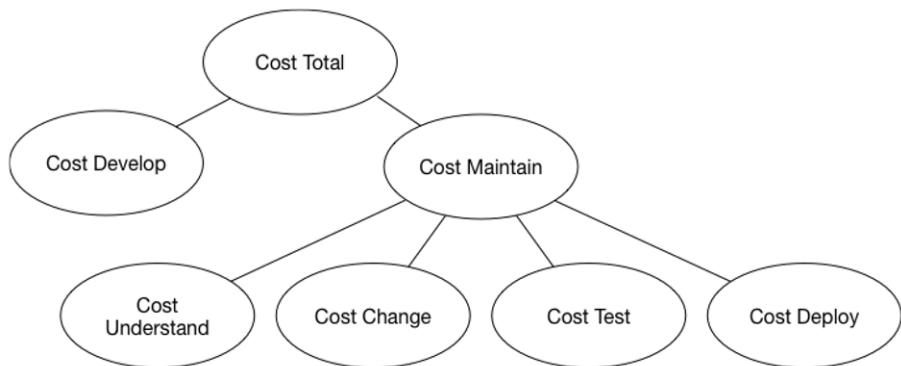
¹⁹https://en.wikipedia.org/wiki/Natural_bodybuilding

Introducción

“La fortaleza y la debilidad de JavaScript reside en que te permite hacer cualquier cosa, tanto para bien como para mal”. – [Reginald Braithwaite²⁰](#)

En los últimos años, JavaScript se ha convertido en uno de los lenguajes más utilizados del mundo. Su principal ventaja, y a la vez su mayor debilidad, es su versatilidad. Esa gran versatilidad ha derivado en algunas malas prácticas que se han ido extendiendo en la comunidad. Aún así, JavaScript se encuentra en infraestructuras críticas de [empresas muy importantes²¹](#) (Facebook, Netflix o Uber lo utilizan), en las cuales limitar los costes derivados del mantenimiento del software se vuelve esencial.

El coste total de un producto *software* viene dado por la suma de los costes de desarrollo y de mantenimiento, siendo este último mucho más elevado que el coste del propio desarrollo inicial. A su vez, como expone Kent Beck en su libro *Implementation Patterns²²*, el coste de mantenimiento viene dado por la suma de los costes de entender el código, cambiarlo, testearlo y desplegarlo.



Esquema de costes de Kent Beck

²⁰<https://twitter.com/raganwald>

²¹<https://stackshare.io/javascript>

²²<https://amzn.to/2BHRU8P>

Además de los costes mencionados, [Dan North](#)²³ famoso por ser uno de los creadores de [BDD](#)²⁴, también hace hincapié en el coste de oportunidad y en el coste por el retraso en las entregas. Aunque en este libro no voy a entrar en temas relacionados con la gestión de proyectos, si que creo que es importante ser conscientes de cuales son los costes que generamos los desarrolladores y sobre todo en qué podemos hacer para minimizarlos.

En la primera parte del libro trataré de exponer algunas maneras de minimizar el coste relacionado con la parte de entender el código, para ello trataré de sintetizar y ampliar algunos de los conceptos relacionados con esto que exponen [Robert C. Martin](#)²⁵, [Kent Beck](#)²⁶, [Ward Cunningham](#)²⁷ sobre Clean Code y otros autores aplicándolos a JavaScript.

En la segunda parte veremos cómo los principios SOLID nos pueden ayudar a escribir código mucho más intuitivo que nos ayudará a reducir los costes de mantenimiento relacionados con la tolerancia al cambio de nuestro código.

En la tercera y última parte, trataremos de ver cómo nos pueden ayudar los test unitarios y el diseño dirigido por test (TDD) a escribir código de mayor calidad y robustez, lo cual nos ayudará, además de a prevenir la deuda técnica, a minimizar el coste relacionado con testear el software.

²³<https://twitter.com/tastapod?lang=es>

²⁴https://en.wikipedia.org/wiki/Behavior-driven_development

²⁵<https://twitter.com/unclebobmartin>

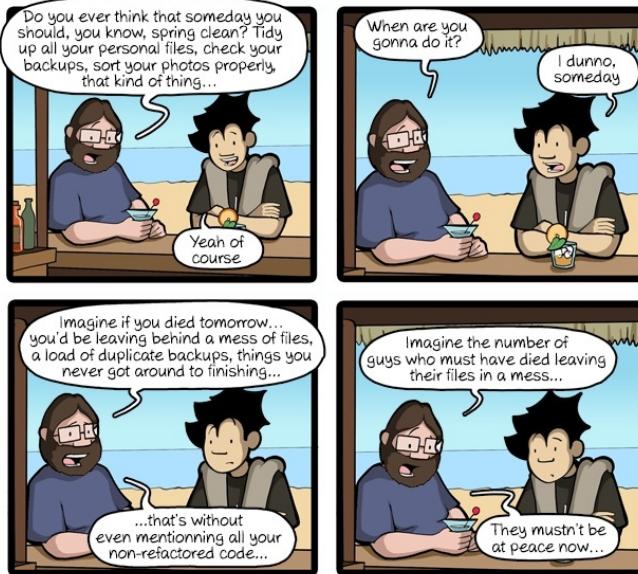
²⁶<https://twitter.com/KentBeck>

²⁷<https://twitter.com/WardCunningham>

Deuda técnica

“Un Lannister siempre paga sus deudas”. – Game of Thrones

Podemos considerar la deuda técnica como una metáfora que trata de explicar que la falta de calidad en el código de un proyecto *software* genera una deuda que repercutirá en sobrecostes futuros. Dichos sobrecostes están directamente relacionados con la tolerancia al cambio del sistema *software* en cuestión.





La última deuda técnica. Viñeta de CommitStrip

El concepto de deuda técnica fue introducido en primera instancia por Ward Cunningham en la conferencia OOPSLA del año 1992. Desde entonces, diferentes autores han tratado de extender la metáfora para abarcar más conceptos económicos y otras situaciones en el ciclo de vida del *software*.

Tipos de deuda

...

Refactoring, las deudas se pagan

...

Mejor prevenir que curar, las reglas del diseño simple

...

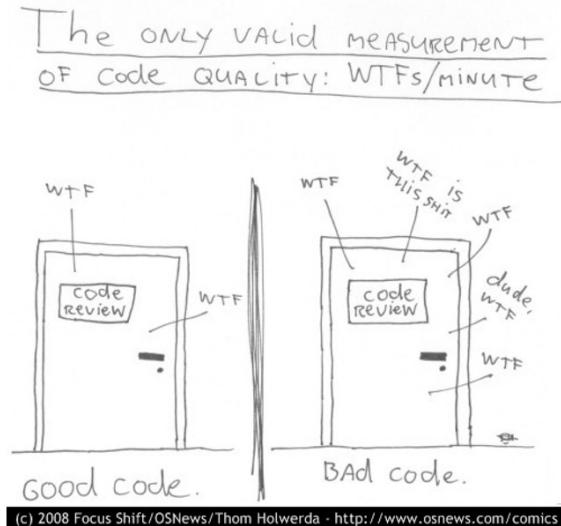
SECCIÓN I: CLEAN CODE

¿Qué es Clean Code?

“Programar es el arte de decirle a otro humano lo que quieras que el ordenador haga”. – Donald Knuth²⁸

Clean Code, o Código Limpio en español, es un término al que ya hacían referencia desarrolladores de la talla de Ward Cunningham o Kent Beck, aunque no se popularizó hasta que Robert C. Martin²⁹, también conocido como Uncle Bob, publicó su libro “Clean Code: A Handbook of Agile Software Craftsmanship³⁰” en 2008.

El libro, aunque sea bastante dogmático y quizás demasiado focalizado en la programación orientada a objetos, se ha convertido en un clásico que no debe faltar en la estantería de ningún desarrollador que se precie, aunque sea para criticarlo.



Viñeta de osnews.com/comics/ sobre la calidad del código

²⁸https://es.wikipedia.org/wiki/Donald_Knuth

²⁹<https://twitter.com/unclebobmartin>

³⁰<https://amzn.to/2TUywwB>

Existen muchas definiciones para el término Clean Code, pero yo personalmente me quedo con la de mi amigo Carlos Blé, ya que además casa muy bien con el objetivo de este libro:

“Código Limpio es aquel que se ha escrito con la intención de que otra persona (o tú mismo en el futuro) lo entienda.” – *Carlos Blé*³¹

Los desarrolladores solemos escribir código sin la intención explícita de que vaya a ser entendido por otra persona, ya que la mayoría de las veces nos centramos simplemente en implementar una solución que funcione y que resuelva el problema.

Tratar de entender el código de un tercero, o incluso el que escribimos nosotros mismos hace tan solo unas semanas, se puede volver una tarea realmente difícil. Es por ello que hacer un esfuerzo extra para que nuestra solución sea legible e intuitiva es la base para reducir los costes de mantenimiento del *software* que producimos.

A continuación veremos algunas de las secciones del libro de Uncle Bob que más relacionadas están con la legibilidad del código. Si conoces el libro o lo has leído, podrás observar que he añadido algunos conceptos y descartado otros, además de incluir ejemplos sencillos aplicados a JavaScript.

³¹<https://twitter.com/carlosble>

Variables y nombres

“Nuestro código tiene que ser simple y directo, debería leerse con la misma facilidad que un texto bien escrito”. – Grady Booch³²

Nuestro código debería poder leerse con la misma facilidad con la que leemos un texto bien escrito, es por ello que escoger buenos nombres es fundamental. Los nombres de variables, métodos y clases deben seleccionarse con cuidado para que den expresividad y significado a nuestro código.



Viñeta de Commit Strip sobre el nombrado de variables.

A continuación veremos algunas pautas y ejemplos para tratar de mejorar a la hora de escoger buenos nombres:

³²https://es.wikipedia.org/wiki/Grady_Booch

Nombres pronunciables y expresivos

...

Uso correcto de *var*, *let* y *const*

...

Ausencia de información técnica en los nombres

...

Léxico coherente

...

Nombres según el tipo de dato

Arrays

...

Booleanos

...

Números

...

Funciones

...

Clases

...

Funciones

“Sabemos que estamos desarrollando código limpio cuando cada función hace exactamente lo que su nombre indica”. – Ward Cunningham³³

Tamaño y función

...

Número de argumentos

...

Estilo declarativo frente al imperativo

...

Funciones anónimas

...

Transparencia referencial

...

Principio DRY

...

³³https://es.wikipedia.org/wiki/Ward_Cunningham

Evita el uso de comentarios

“No comentes el código mal escrito, reescríbelo”. – Brian W. Kernighan³⁴

...

³⁴https://es.wikipedia.org/wiki/Brian_Kernighan

Formato coherente

“El buen código siempre parece estar escrito por alguien a quien le importa”. – Michael Feathers³⁵

...

Problemas similares, soluciones simétricas

...

Densidad, apertura y distancia vertical

...

Lo más importante primero

...

Indentación

...

³⁵<https://twitter.com/mfeathers?lang=es>

Clases

“Siquieres ser un programador productivo esfuérzate en escribir código legible”.

– Robert C. Martin³⁶

...

Tamaño reducido

...

Organización

...

Prioriza la composición frente a la herencia

...

³⁶<https://twitter.com/unclebobmartin>

SECCIÓN II: PRINCIPIOS SOLID

Introducción a SOLID

En la sección sobre Clean Code aplicado a JavaScript, vimos que el coste total de un producto *software* viene dado por la suma de los costes de desarrollo y de mantenimiento, siendo este último mucho más elevado que el coste del propio desarrollo inicial.

En dicha sección nos centramos en la idea de minimizar el coste de mantenimiento relacionado con la parte de entender el código, y ahora nos vamos a focalizar en cómo nos pueden ayudar los principios SOLID a escribir un código más intuitivo, testeable y tolerante a cambios.

Antes de profundizar en SOLID, vamos a hablar de qué sucede en nuestro proyecto cuando escribimos código STUPID.

De STUPID a SOLID

Tranquilidad, no pretendo herir tus sentimientos, STUPID es simplemente un acrónimo basado en seis *code smells* que describen cómo NO debe ser el *software* que desarrollamos.

- Singleton: patrón singleton
- Tight Coupling: alto acoplamiento
- Untestability: código no testeable
- Premature optimization: optimizaciones prematuras
- Indescriptive Naming: nombres poco descriptivos
- Duplication: duplicidad de código

¿Qué es un *code smell*?

...

El patrón singleton

...

Alto acoplamiento

...

Acoplamiento y cohesión

Código no testeable

...

Optimizaciones prematuras

...

Complejidad esencial y complejidad accidental

...

Nombres poco descriptivos

...

Duplicidad de código

...

Duplicidad real

...

Duplicidad accidental

...

Principios SOLID al rescate

Los principios de SOLID nos indican cómo organizar nuestras funciones y estructuras de datos en componentes y cómo dichos componentes deben estar interconectados. Normalmente éstos suelen ser clases, aunque esto no implica que dichos principios solo sean aplicables al paradigma de la orientación a objetos, ya que podríamos tener simplemente una agrupación de funciones y datos, por ejemplo, en una *Closure*. En definitiva, cada producto *software* tiene dichos componentes, ya sean clases o no, por lo tanto tendría sentido aplicar los principios SOLID.

El acrónimo SOLID fue creado por [Michael Feathers³⁷](#), y, como no, popularizado por Robert C. Martin en su libro *Agile Software Development: Principles, Patterns, and Practices*. Consiste en cinco principios o convenciones de diseño de *software*, ampliamente aceptados por la industria, que tienen como objetivos ayudarnos a mejorar los costes de mantenimiento derivados de cambiar y testear nuestro código.

- Single Responsibility: Responsabilidad única.
- Open/Closed: Abierto/Cerrado.
- Liskov substitution: Sustitución de Liskov.
- Interface segregation: Segregación de interfaz.
- Dependency Inversion: Inversión de dependencia.

Es importante resaltar que se trata de principios, no de reglas. Una regla es de obligatorio cumplimiento, mientras que los principios son recomendaciones que pueden ayudar a hacer las cosas mejor.

³⁷<https://michaelfeathers.silvrback.com/>

SRP - Principio de responsabilidad única

“Nunca debería haber más de un motivo por el cual cambiar una clase o un módulo”. – Robert C. Martin

...

¿Qué entendemos por responsabilidad?

...

Aplicando el SRP

...

Detectar violaciones del SRP:

...

OCP - Principio Abierto/Cerrado

“Todas las entidades software deberían estar abiertas a extensión, pero cerradas a modificación”. – [Bertrand Meyer](#)³⁸

El principio *Open-Closed* (Abierto/Cerrado), enunciado por Bertrand Meyer, nos recomienda que, en los casos en los que se introduzcan nuevos comportamientos en sistemas existentes, en lugar de modificar los componentes antiguos, se deben crear componentes nuevos. La razón es que si esos componentes o clases están siendo usadas en otra parte (del mismo proyecto o de otros) estaremos alterando su comportamiento y provocando efectos indeseados.

Este principio promete mejoras en la estabilidad de tu aplicación al evitar que las clases existentes cambien con frecuencia, lo que también hace que las cadenas de dependencia sean un poco menos frágiles, ya que habrá menos partes móviles de las que preocuparse. Cuando creamos nuevas clases es importante tener en cuenta este principio para facilitar su extensión en un futuro. Pero, en la práctica, ¿cómo es posible modificar el comportamiento de un componente o módulo sin modificar el código existente?

Aplicando el OCP

Aunque este principio puede parecer una contradicción en sí mismo, existen varias técnicas para aplicarlo, pero todas ellas dependen del contexto en el que estemos. Una de estas técnicas podría ser utilizar un mecanismos de extensión, como la herencia o la composición, para utilizar esas clases a la vez que modificamos su comportamiento. Como comentamos en el capítulo de clases en la sección de Clean Code, deberías tratar de priorizar la composición frente a la herencia.

Creo que un buen contexto para ilustrar cómo aplicar el OCP podría ser tratar de desacoplar un elemento de infraestructura de la capa de dominio. Imagina que tenemos un sistema de gestión de tareas, concretamente tenemos una clase llamada

³⁸https://en.wikipedia.org/wiki/Bertrand_Meyer

TodoService, que se encarga de realizar una petición HTTP a una API REST para obtener las diferentes tareas que contiene el sistema:

Principio abierto/cerrado

```
1 const axios = require('axios');
2
3 class TodoExternalService{
4
5   requestTodoItems(callback){
6     const url = 'https://jsonplaceholder.typicode.com/todos/';
7
8     axios
9       .get(url)
10      .then(callback)
11    }
12  }
13
14 new TodoExternalService()
15   .requestTodoItems(response => console.log(response.data))
```

Puedes acceder al ejemplo interactivo [desde aquí³⁹](#).

En este ejemplo están ocurriendo dos cosas, por un lado estamos acoplando un elemento de infraestructura y una librería de terceros en nuestro servicio de dominio y, por otro, nos estamos saltando el principio de abierto/cerrado, ya que si quisieramos reemplazar la librería *axios* por otra, como *fetch*, tendríamos que modificar la clase. Para solucionar estos problemas vamos a hacer uso del patrón adaptador.

Patrón adaptador

El patrón *adapter* o adaptador pertenece a la categoría de patrones estructurales. Se trata de un patrón encargado de homogeneizar APIs, esto nos facilita la tarea de desacoplar tanto elementos de diferentes capas de nuestro sistema como librerías de terceros.

³⁹<https://repl.it/@SoftwareCrafter/SOLID-OCP-2>

Para aplicar el patrón *adapter* en nuestro ejemplo, necesitamos crear una nueva clase que vamos a llamar *ClientWrapper*. Dicha clase va a exponer un método *makeRequest* que se encargará de realizar las peticiones para una determinada URL recibida por parámetro. También recibirá un *callback* en el que se resolverá la petición:

Patrón adaptador

```
1 class ClientWrapper{  
2     makeGetRequest(url, callback){  
3         return axios  
4             .get(url)  
5             .then(callback);  
6     }  
7 }
```

ClientWrapper es una clase que pertenece a la capa de infraestructura. Para utilizarla en nuestro dominio de manera desacoplada debemos inyectarla vía constructor (profundizaremos en la inyección de dependencias en el capítulo de inversión de dependencias). Así de fácil:

Principio abierto/cerrado

```
1 //infrastructure/ClientWrapper  
2 const axios = require('axios');  
3  
4 export class ClientWrapper{  
5     makeGetRequest(url, callback){  
6         return axios  
7             .get(url)  
8             .then(callback);  
9     }  
10 }  
11  
12 //domain/TodoService  
13 export class TodoService{  
14     client;
```

```
16  constructor(client){
17      this.client = client;
18  }
19
20  requestTodoItems(callback){
21      const url = 'https://jsonplaceholder.typicode.com/todos/';
22      this.client.makeGetRequest(url, callback)
23  }
24 }
25
26 //index
27 import {ClientWrapper} from './infrastructure/ClientWrapper'
28 import {TodoService} from './domain/TodoService'
29
30 const start = () => {
31     const client = new ClientWrapper();
32     const todoService = new TodoService(client);
33
34     todoService.requestTodoItems(response => console.log(response.data))
35 }
36
37 start();
```

Puedes acceder al ejemplo completo [desde aquí](#).⁴⁰

Como puedes observar, hemos conseguido eliminar la dependencia de *axios* de nuestro dominio. Ahora podríamos utilizar nuestra clase *ClientWrapper* para hacer peticiones HTTP en todo el proyecto. Esto nos permitiría mantener un bajo acoplamiento con librerías de terceros, lo cual es tremadamente positivo para nosotros, ya que si quisieramos cambiar la librería *axios* por *fetch*, por ejemplo, tan solo tendríamos que hacerlo en nuestra clase *ClientWrapper*:

⁴⁰<https://repl.it/@SoftwareCrafter/SOLID-OCP1>

Patrón adaptador

```
1 export class ClientWrapper{  
2     makeGetRequest(url, callback){  
3         return fetch(url)  
4             .then(response => response.json())  
5             .then(callback)  
6     }  
7 }
```

De esta manera hemos conseguido cambiar *requestTodoItems* sin modificar su código, con lo que estaríamos respetando el principio abierto/cerrado.

Detectar violaciones del OCP

Como habrás podido comprobar, este principio está estrechamente relacionado con el de responsabilidad única. Normalmente, si un elevado porcentaje de cambios suele afectar a nuestra clase, es un síntoma de que dicha clase, además de estar demasiado acoplada y de tener demasiadas responsabilidades, está violando el principio abierto cerrado.

Además, como vimos en el ejemplo, el principio se suele violar muy a menudo cuando involucramos diferentes capas de la arquitectura del proyecto.

LSP - Principio de sustitución de Liskov

“Las funciones que utilicen punteros o referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo”. – Robert C. Martin

...

Aplicando el LSP

...

Detectar violaciones del LSP

...

ISP - Principio de segregación de la interfaz

“Los clientes no deberían estar obligados a depender de interfaces que no utilicen”. – Robert C. Martin

...

Aplicando el ISP

...

Detectar violaciones del ISP

...

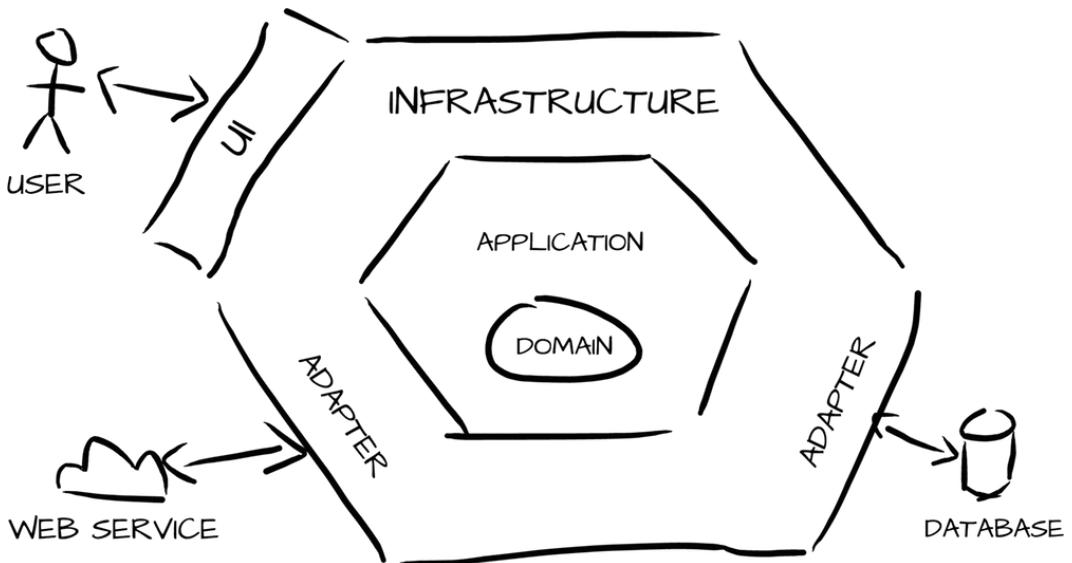
DIP - Principio de inversión de dependencias

“Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de concreciones. Los detalles deben depender de abstracciones”. – Robert C. Martin

En este capítulo vamos a tratar el quinto y último de los principios, la inversión de dependencia. Este principio se atribuye a Robert C. Martin y se remonta nada menos que al año 1995. Este principio viene a decir que las clases o módulos de las capas superiores no deberían depender de las clases o módulos de las capas inferiores, sino que ambas deberían depender de abstracciones. A su vez, dichas abstracciones no deberían depender de los detalles, sino que son los detalles los que deberían depender de las mismas. Pero, ¿esto qué significa? ¿A qué se refiere con módulos de bajo y alto nivel? ¿Cuál es el motivo de depender de abstracciones?

Módulos de alto nivel y módulos de bajo nivel

Cuando Uncle Bob dice que los módulos de alto nivel no deberían depender de módulos de bajo nivel, se refiere a que los componentes importantes (capas superiores) no deberían depender de componentes menos importantes (capas inferiores). Desde el punto de vista de la arquitectura hexagonal, los componentes más importantes son aquellos centrados en resolver el problema subyacente al negocio, es decir, la capa de dominio. Los menos importantes son los que están próximos a la infraestructura, es decir, aquellos relacionados con la UI, la persistencia, la comunicación con API externas, etc. Pero, ¿esto por qué es así? ¿Por qué la capa de infraestructura es menos importante que la capa de dominio?



Esquema arquitectura hexagonal

Imagina que en nuestra aplicación usamos un sistema de persistencia basado en ficheros, pero por motivos de rendimiento o escalabilidad queremos utilizar una base de datos documental tipo mongoDB. Si hemos desacoplado correctamente la capa de persistencia, por ejemplo aplicando el patrón repositorio, la implementación de dicha capa le debe ser indiferente a las reglas de negocio (capa de dominio). Por lo tanto cambiar de un sistema de persistencia a otro, una vez implementado el repositorio, se vuelve prácticamente trivial. En cambio, una modificación de las reglas de negocio sí que podría afectar a qué datos se deben almacenar, con lo cual afectaría a la capa de persistencia.

Esto pasa exactamente igual con la capa de presentación, a nuestra capa de dominio le debe dar igual si utilizamos *React*, *Vue* o *Angular*. Incluso, aunque se trata de un escenario poco realista, deberíamos tener la capacidad de poder reemplazar la librería que usemos en nuestras vistas, por ejemplo *React*, por *Vue* o *Angular*. En cambio, una modificación en las reglas de negocio sí es probable que se vea reflejada en la UI.

Depender de abstracciones

...

Inyección de dependencias

...

Aplicando el DIP

...

Detectando violaciones del DIP

...

Introducción al testing

“El testing de software puede verificar la presencia de errores pero no la ausencia de ellos”. – [Edsger Dijkstra](#)⁴¹

La primera de las cuatro reglas del diseño simple de Kent Beck nos dice que nuestro código debe de pasar correctamente el conjunto de test automáticos. Para Kent Beck esta es la regla más importante y tiene todo el sentido, ya que si no puedes verificar que tu sistema funciona, de nada sirve que hayas hecho un gran diseño a nivel de arquitectura o que hayas aplicado todas las buenas prácticas que hemos visto hasta ahora.



Viñeta de commit strip sobre la importancia de los tests.

El *testing de software* cobra especial importancia cuando trabajamos con lenguajes dinámicos como JavaScript, sobre todo cuando la aplicación adquiere cierta comple-

⁴¹https://es.wikipedia.org/wiki/Edsger_Dijkstra

jidad. La principal razón de esto es que no hay una fase de compilación como en los lenguajes de tipado estático, con lo cual no podemos detectar fallos hasta el momento de ejecutar la aplicación.

Esta es una de las razones por lo que se vuelve muy interesante el uso de TypeScript, ya que el primer control de errores lo realiza su compilador. Esto no quiere decir que no tengamos *testing* si lo usamos, sino que, en mi opinión, lo ideal es usarlos de forma combinada para obtener lo mejor de ambos mundos.

A continuación veremos algunos conceptos generales sobre el *testing* de *software*, como los diferentes tipos que existen. Para luego centrarnos en los que son más importantes desde el punto de vista del desarrollador: los tests unitarios.

Tipos de tests de software

...

¿Qué entendemos por testing?

...

Test manuales vs automáticos

...

Test funcionales vs no funcionales

...

Tests funcionales

...

Tests no funcionales

...

Pirámide de testing

...

Antipatrón del cono de helado

...

Tests unitarios

“Nos pagan por hacer software que funcione, no por hacer tests”. – Kent Beck.

El *unit testing*, o test unitarios en castellano, no es un concepto nuevo en el mundo del desarrollo de software. Ya en la década de los años 70, cuando surgió el lenguaje **Smalltalk**⁴², se hablaba de ello, aunque con diferencias a como lo conocemos hoy en día.

La popularidad del *unit testing* actual se la debemos a Kent Beck. Primero lo introdujo en el lenguaje *Smalltalk* y luego consiguió que se volviera mainstream en otros muchos lenguajes de programación. Gracias a él, las pruebas unitarias se han convertido en una práctica extremadamente útil e indispensable en el desarrollo de *software*. Pero, ¿qué es exactamente una prueba o test unitario?

Según Wikipedia: “Una prueba unitaria es una forma de comprobar el correcto funcionamiento de una unidad de código”. Entendiendo por unidad de código una función o una clase.

Desde mi punto de vista, esa definición está incompleta. Lo primero que me chirría es que no aparece la palabra “automatizado” en la definición. Por otro lado, una clase es una estructura organizativa que suele incluir varias unidades de código. Normalmente, para probar una clase vamos a necesitar varios test unitarios, a no ser que tengamos clases con un solo método, lo cual, como ya comentamos en el capítulo de responsabilidad única, no suele ser buena idea. Quizás una definición más completa sería:

Un test unitario es un pequeño programa que comprueba que una unidad de *software* tiene el comportamiento esperado. Para ello prepara el contexto, ejecuta dicha unidad y, a continuación, verifica el resultado obtenido a través de una o varias aserciones que comparan el resultado obtenido con el esperado.

⁴²<https://es.wikipedia.org/wiki/Smalltalk>

Características de los tests unitarios

...

Anatomía de un test unitario

...

Jest, el framework de testing JavaScript definitivo

Un *framework* de *testing* es una herramienta que nos permite escribir test de una manera sencilla, además nos provee de un entorno de ejecución que nos permite extraer información de los mismos de manera sencilla.

Históricamente JavaScript ha sido uno de los lenguajes con más *frameworks* y librerías de test, pero a la vez es uno de los lenguajes con menos cultura de *testing* entre los miembros de su comunidad. Entre dichos *frameworks* y librerías de *testing* automatizado destacan [Mocha⁴³](#), [Jasmine⁴⁴](#) y [Jest⁴⁵](#), entre otras. Nosotros nos vamos a centrar en Jest, ya que simplifica el proceso gracias a que integra todos los elementos que necesitamos para poder realizar nuestros test automatizados.

Jest es un *framework* de *testing* desarrollado por el equipo de Facebook basado en RSpec. Aunque nace en el contexto de [React⁴⁶](#), es un framework de testing generalista que podemos utilizar en cualquier situación. Se trata de un framework flexible, rápido y con un output sencillo y comprensible, que nos permite completar un ciclo de feedback rápido y con la máxima información en cada momento.

Características

...

Instalación y configuración

...

⁴³<https://mochajs.org/>

⁴⁴<https://jasmine.github.io/>

⁴⁵<https://facebook.github.io/jest/>

⁴⁶<https://softwarecrafters.io/react/tutorial-react-js-introduccion>

Nuestro primer test

...

Aserciones

...

Organización y estructura

...

Gestión del estado: before y after

...

Code coverage

...

TDD - Test Driven Development

Test Driven Development (TDD), o desarrollo dirigido por test en castellano, es una técnica de ingeniería de *software* para, valga la redundancia, diseñar *software*. Como su propio nombre indica, esta técnica dirige el desarrollo de un producto a través de ir escribiendo pruebas, generalmente unitarias.

El TDD fue desarrollada por Kent Beck a finales de la década de los 90 y forma parte de la metodología *extreme programming*⁴⁷. Su autor y los seguidores del TDD aseguran que con esta técnica se consigue un código más tolerante al cambio, robusto, seguro, más barato de mantener e, incluso, una vez que te acostumbras a aplicarlo, promete una mayor velocidad a la hora de desarrollar.

Las tres leyes del TDD

...

El ciclo Red-Green-Refactor

...

TDD como herramienta de diseño

...

Estrategias de implementación, de rojo a verde.

...

⁴⁷https://en.wikipedia.org/wiki/Extreme_programming

Implementación falsa

...

Triangular

...

Implementación obvia

...

Limitaciones del TDD

...

TDD Práctico: La kata FizzBuzz

“Muchos son competentes en las aulas, pero llévalos a la práctica y fracasan estrepitosamente”. – Epicteto

...

Las katas de código

...

La kata FizzBuzz

...

Descripción del problema

...

Diseño de la primera prueba

...

Ejecutamos y... ¡rojo!

...

Pasamos a verde

...

Añadiendo nuevas pruebas

...

Refactorizando la solución, aplicando pattern matching.

...

Referencias

- Clean Code: A Handbook of Agile Software Craftsmanship de Robert C. Martin⁴⁸
- Clean Architecture: A Craftsman's Guide to Software Structure and Design de Robert C Martin⁴⁹
- The Clean Coder: A Code of Conduct for Professional Programmers de Robert C. Martin⁵⁰
- Test Driven Development. By Example de Kent Beck⁵¹
- Extreme Programming Explained de Kent Beck⁵²
- Implementation Patterns de Kent Beck⁵³
- Refactoring: Improving the Design of Existing Code de Martin Fowler⁵⁴
- Design patterns de Erich Gamma, John Vlissides, Richard Helm y Ralph Johnson⁵⁵
- Effective Unit Testing de Lasse Koskela⁵⁶
- The Art of Unit Testing: with examples in C# de Roy Osherove⁵⁷
- JavaScript Allonge de Reg “raganwald” Braithwaite⁵⁸
- You Don’t Know JS de Kyle Simpson⁵⁹
- Diseño Ágil con TDD de Carlos Blé⁶⁰
- Testing y TDD para PHP de Fran Iglesias⁶¹
- Cursos de Codely.TV⁶²

⁴⁸<https://amzn.to/2TUywwB>

⁴⁹<https://amzn.to/2ph2wrZ>

⁵⁰<https://amzn.to/2q5xgws>

⁵¹<https://amzn.to/2j1zWSH>

⁵²<https://amzn.to/2VHQkNg>

⁵³<https://amzn.to/2Hnh7cC>

⁵⁴<https://amzn.to/2MGmeFy>

⁵⁵<https://amzn.to/2EW7MXv>

⁵⁶<https://amzn.to/2VCcsbP>

⁵⁷<https://amzn.to/31ahpK6>

⁵⁸<https://leanpub.com/javascript-allonge>

⁵⁹<https://amzn.to/2OJ24xu>

⁶⁰<https://www.carlosble.com/libro-tdd/?lang=es>

⁶¹<https://leanpub.com/testingytdparaph>

⁶²<https://codely.tv/pro/cursos>

- **Repositorio de Ryan McDermott⁶³**
- **Guía de estilo de Airbnb⁶⁴**
- **El artículo “From Stupid to SOLID code” de Willian Durand⁶⁵**
- **Blog Koalite⁶⁶**
- Conversaciones con los colegas **Carlos Blé⁶⁷**, Dani García, **Patrick Hertling⁶⁸** y **Juan M. Gómez⁶⁹**.

⁶³<https://github.com/ryanmcdermott/clean-code-javascript>

⁶⁴<https://github.com/airbnb/javascript>

⁶⁵<https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/>

⁶⁶<http://blog.koalite.com/>

⁶⁷<https://twitter.com/carlosble>

⁶⁸<https://twitter.com/PatrickHertling>

⁶⁹https://twitter.com/_jmgomez_

OFERTA DE LANZAMIENTO

Hasta final de año, diciembre de 2019, el e-book tendrá un descuento del 25%. No dejes escapar la oportunidad, estoy convencido de que te puede aportar algún detalle de mucho valor.

Recuerda que si no es lo que esperas, te devolvemos tu dinero. Durante los primeros 45 días de compra, puedes obtener un reembolso del 100%. El riesgo es cero y el beneficio podría ser muy elevado.

Puedes adquirir el e-book desde aquí.⁷⁰

⁷⁰<https://softwarecrafters.io/cleancode-solid-testing-js>