

CLEAN JAVASCRIPT

A large, bold, black "JS" logo is centered on a yellow-to-black gradient background. The letters are thick and have a slight curve to them.

APRENDE A APLICAR CÓDIGO
LIMPIO, SOLID Y TESTING

MIGUEL A. GÓMEZ

Clean JavaScript

Aprende a aplicar código limpio, SOLID y Testing

Software Crafters

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

Esta versión se publicó en 2020-07-05



Éste 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 retroalimentación del lector hasta conseguir el libro adecuado.

© 2019 - 2020 Software Crafters

Índice general

Prefacio	1
Qué no es este libro	1
Agradecimientos	2
Sobre Software Crafters	3
Sobre el autor	4
Introducción	5
Deuda técnica	7
Tipos de deuda	8
Refactoring, las deudas se pagan	9
Mejor prevenir que curar, las reglas del diseño simple	10
SECCIÓN I: CLEAN CODE	11
¿Qué es Clean Code?	12
Variables, nombres y ámbito	14
Uso correcto de <i>var</i> , <i>let</i> y <i>const</i>	15
Nombres pronunciables y expresivos	16
Ausencia de información técnica en los nombres	17
Establecer un lenguaje ubicuo	17
Nombres según el tipo de dato	18
Arrays	18
Booleanos	19
Números	19

ÍNDICE GENERAL

Funciones	20
Clases	21
Ámbito o <i>scope</i> de las variables	22
Ámbito global	22
Ámbito de bloque	22
Ámbito estático vs. dinámico	23
<i>Hoisting</i>	24
Funciones	26
Declaración de una función	26
Expresión de una función	26
Expresiones con funciones flecha (<i>arrow functions</i>)	27
Funcionamiento del objeto <i>this</i> en <i>arrow functions</i>	28
Funciones autoejecutadas IIFE	30
Parámetros y argumentos	31
Limita el número de argumentos	32
Parámetros por defecto	33
Parámetro <i>rest</i> y operador <i>spread</i>	34
Tamaño y niveles de indentación	36
Cláusulas de guarda	38
Evita el uso de <i>else</i>	38
Prioriza las condiciones asertivas	39
Estilo declarativo frente al imperativo	40
Funciones anónimas	41
Transparencia referencial	42
Principio DRY	43
<i>Command–Query Separation (CQS)</i>	45
Algoritmos eficientes	46
Notación O grande (<i>big O</i>)	46
Clases	49
<i>Prototype</i> y <i>ECMAScript</i> moderno	49
Constructores y funciones constructoras	49
Métodos	51
Herencia y cadena de prototipos	52
Tamaño reducido	53

ÍNDICE GENERAL

Organización	56
Prioriza la composición frente a la herencia	57
Comentarios y formato	60
Evita el uso de comentarios	60
Formato coherente	60
Problemas similares, soluciones simétricas	61
Densidad, apertura y distancia vertical	61
Lo más importante primero	61
Indentación	61
SECCIÓN II: PRINCIPIOS SOLID	62
Introducción a SOLID	63
De STUPID a SOLID	64
¿Qué es un <i>code smell</i> ?	64
El patrón singleton	64
Alto acoplamiento	66
Acoplamiento y cohesión	66
Código no testeable	66
Optimizaciones prematuras	67
Complejidad esencial y complejidad accidental	67
Nombres poco descriptivos	68
Duplicidad de código	68
Duplicidad real	68
Duplicidad accidental	68
Principios SOLID al rescate	69
SRP - Principio de responsabilidad única	70
¿Qué entendemos por responsabilidad?	70
Aplicando el SRP	72
Detectar violaciones del SRP:	74
OCP - Principio Abierto/Cerrado	76
Aplicando el OCP	76
Patrón adaptador	77

ÍNDICE GENERAL

Detectar violaciones del OCP	80
LSP - Principio de sustitución de Liskov	81
Aplicando el LSP	81
Detectar violaciones del LSP	86
ISP - Principio de segregación de la interfaz	87
Aplicando el ISP	87
Detectar violaciones del ISP	92
DIP - Principio de inversión de dependencias	94
Módulos de alto nivel y módulos de bajo nivel	94
Depender de abstracciones	96
Inyección de dependencias	96
Aplicando el DIP	99
Detectando violaciones del DIP	100
SECCIÓN III: TESTING Y TDD	101
Introducción al testing	102
Tipos de tests de software	104
¿Qué entendemos por testing?	104
Test manuales vs automáticos	104
Test funcionales vs no funcionales	105
Tests funcionales	105
Tests no funcionales	106
Pirámide de testing	107
Antipatrón del cono de helado	108
Tests unitarios	110
Características de los tests unitarios	111
Anatomía de un test unitario	111
Jest, el framework de testing JavaScript definitivo	114
Características	114
Instalación y configuración	115
Nuestro primer test	117

ÍNDICE GENERAL

Aserciones	118
Organización y estructura	120
Gestión del estado: before y after	120
Code coverage	121
TDD - Test Driven Development	123
Las tres leyes del TDD	123
El ciclo Red-Green-Refactor	123
TDD como herramienta de diseño	125
Estrategias de implementación, de rojo a verde.	126
Implementación falsa	126
Triangular	128
Implementación obvia	131
Limitaciones del TDD	131
TDD Práctico: La kata FizzBuzz	133
Las katas de código	133
La kata FizzBuzz	134
Descripción del problema	134
Diseño de la primera prueba	135
Ejecutamos y... ¡rojo!	136
Pasamos a verde	137
Añadiendo nuevas pruebas	138
Refactorizando la solución, aplicando pattern matching.	144
Siguientes pasos	147
Referencias	148

Descargado en: www.detodoprogramacion.org

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](#)¹, Carlos Bello, Dani García, [Ramón Esteban](#)², [Patrick Hertling](#)³ y, como no, gracias a [Carlos Blé](#)⁴ 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.

¹https://twitter.com/_jmgomez_

²<https://twitter.com/ramonesteban78>

³<https://twitter.com/PatrickHertling>

⁴<https://twitter.com/carlosble>

Sobre Software Crafters

Software Crafters⁵ 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.

⁵<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](https://www.haskell.org/)⁶.

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 [programación funcional reactiva \(FRP\)](https://en.wikipedia.org/wiki/Functional_reactive_programming)⁷, tanto para el **frontend** con [TypeScript](https://softwarecrafters.io/typescript/typescript-javascript-introduccion)⁸, [RxJS](https://softwarecrafters.io/javascript/introduccion-programacion-reactiva-rxjs)⁹ y [ReactJS](https://softwarecrafters.io/react/tutorial-react-js-introduccion)¹⁰, como para el **backend** con [TypeScript](https://softwarecrafters.io/react/tutorial-react-js-introduccion), [NodeJS](https://softwarecrafters.io/javascript/introduccion-programacion-reactiva-rxjs), [RxJS](https://softwarecrafters.io/react/tutorial-react-js-introduccion) y [MongoDB](https://softwarecrafters.io/mongodb/introduccion-mongodb), además de gestionar los procesos [DevOps](https://es.wikipedia.org/wiki/DevOps)¹¹ con [Docker](https://www.omnirooms.com) y [Azure](https://softwarecrafters.io/).

Por otro lado, soy cofundador de la start-up [Omnirooms.com](https://www.omnirooms.com)¹², 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](https://softwarecrafters.io/)¹³, una comunidad sobre artesanía del software, DevOps y tecnologías software con aspiraciones a plataforma de formación y consultoría.

⁶<https://www.haskell.org/>

⁷https://en.wikipedia.org/wiki/Functional_reactive_programming

⁸<https://softwarecrafters.io/typescript/typescript-javascript-introduccion>

⁹<https://softwarecrafters.io/javascript/introduccion-programacion-reactiva-rxjs>

¹⁰<https://softwarecrafters.io/react/tutorial-react-js-introduccion>

¹¹<https://es.wikipedia.org/wiki/DevOps>

¹²<https://www.omnirooms.com>

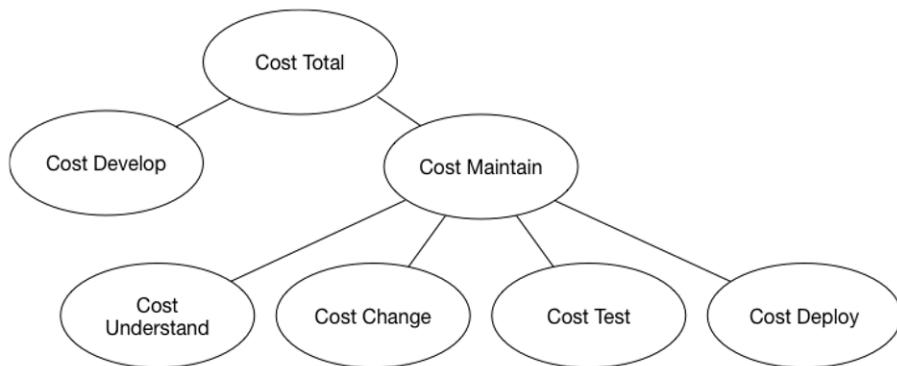
¹³<https://softwarecrafters.io/>

Introducción

“La fortaleza y la debilidad de JavaScript reside en que te permite hacer cualquier cosa, tanto para bien como para mal”. – [Reginald Braithwaite¹⁴](#)

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 importantes¹⁵](#) (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 Patterns¹⁶*, 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

¹⁴<https://twitter.com/raganwald>

¹⁵<https://stackshare.io/javascript>

¹⁶<https://amzn.to/2BHRU8P>

Además de los costes mencionados, [Dan North¹⁷](#) famoso por ser uno de los creadores de [BDD¹⁸](#), 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. Martin¹⁹](#), [Kent Beck²⁰](#), [Ward Cunningham²¹](#) sobre Clean Code y otros autores aplicándolos a JavaScript. Además abordaré algunos conceptos propios del lenguaje que, una vez comprendidos, nos ayudarán a diseñar mejor software.

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.

¹⁷<https://twitter.com/tastapod?lang=es>

¹⁸https://en.wikipedia.org/wiki/Behavior-driven_development

¹⁹<https://twitter.com/unclebobmartin>

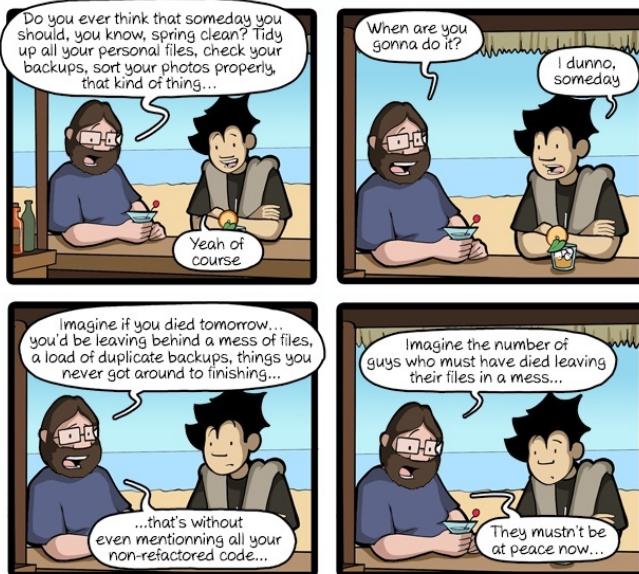
²⁰<https://twitter.com/KentBeck>

²¹<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

Según Martin Fowler, la deuda técnica se puede clasificar en cuatro tipos dependiendo de dónde proviene. Para ilustrarlo confeccionó lo que se conoce como el cuadrante de la deuda técnica:



Esquema de deuda técnica de Martin Fowler

- **Deuda imprudente y deliberada:** En este tipo de deuda el desarrollador actúa conscientemente de forma imprudente y deliberada. Esto suele derivar en un proyecto de mala calidad y poco tolerante al cambio.
- **Deuda imprudente e inadvertida:** Es probable que esta sea el tipo de deuda más peligrosa, ya que es generada desde el desconocimiento y la falta de experiencia. Normalmente suele ser generada por un desarrollador con un perfil júnior o lo que es peor, un falso senior.
- **Deuda prudente y deliberada:** Es el tipo de deuda a la que se refería Ward Cunningham cuando decía que un poco de deuda técnica puede venir bien para acelerar el desarrollo de un proyecto, siempre y cuando esta se pague lo antes posible. El peligro viene con la deuda que no se paga, ya que cuanto más tiempo pasamos con código incorrecto, más se incrementan los intereses.
- **La deuda prudente e inadvertida:** Es común que se dé en la mayoría de los proyectos, ya que esta está relacionada con los conocimientos adquiridos por el propio programador a lo largo del desarrollo del proyecto, el cual llega un momento en el que se hace consciente de que podía haber optado por un diseño mejor. Llegado ese momento es necesario evaluar si la deuda adquirida se debe de pagar o si se puede posponer.

Refactoring, las deudas se pagan

Tal y como hemos visto, a pesar de sus repercusiones negativas, incurrir en deuda técnica es a menudo inevitable. En esos casos, debemos asegurarnos de que somos conscientes de las implicaciones y tratar, como buen Lannister, de pagar la deuda tan pronto como sea posible. Pero, ¿cómo paga un desarrollador la deuda técnica? Pues fácil, por medio de la refactorización.

La refactorización, o *refactoring*, es simplemente un proceso que tiene como objetivo mejorar el código de un proyecto sin alterar su comportamiento para que sea más entendible y tolerante a cambios.

Para comenzar a refactorizar es **imprescindible** que en nuestro proyecto existan **test automáticos**, ya sean unitarios o de integración, que nos permitan saber, en cualquier momento, si el código que hemos cambiado sigue cumpliendo los requisitos. Sin estos test el proceso de refactoring se vuelve complicado, ya que probablemente

adoptaremos actitudes defensivas del tipo “si funciona, no lo toques”, debido al riesgo implícito que conlleva modificar el sistema.

En el caso de que el proyecto contenga test, el siguiente paso es saber detectar cuándo debemos refactorizar. Por norma general, debemos refactorizar cuando detectamos que nuestro código no tiene la **suficiente calidad** o cuando detectamos algún **code smell**, veremos unos cuantos de estos a lo largo del libro. A nivel personal me gusta refactorizar a diario, considero que es un “buen calentamiento”, es decir, una buena forma de reconectar con lo que estaba haciendo el día anterior.

Por otro lado, puede darse el caso en el que refactorizar no sea suficiente y necesitemos cambiar la arquitectura o rediseñar algunos componentes. Por ejemplo, imaginemos una solución en *NodeJS* que debía ser muy escalable y aceptar un número elevado de peticiones concurrentes. Probablemente hubiera sido interesante haber diseñado un sistema de colas para ello. Pero el equipo, de forma prudente y deliberada, puede decidir lanzar las primeras versiones con una arquitectura más simple para validar el producto en el mercado, aún sabiendo que a corto o medio plazo habrá que cambiar la arquitectura.

Mejor prevenir que curar, las reglas del diseño simple

La mala calidad en el *software* siempre la acaba pagando o asumiendo alguien, ya sea el cliente, el proveedor con recursos o el propio desarrollador dedicando tiempo a refactorizar o malgastando tiempo programando sobre un sistema frágil. Es por ello que, como dice el refrán, es mejor prevenir que curar.

Un buen punto de partida para prevenir la deuda técnica es intentar valorar si estamos cumpliendo las **cuatro reglas del diseño simple** planteadas por Kent Beck:

- El código pasa correctamente los test
- Revela la intención del diseño
- Respeta el principio DRY
- Tiene el menor número posible de elementos.

A lo largo del libro veremos cómo aplicar Clean Code, SOLID, TDD y otros muchos conceptos asociados que nos ayudarán a tratar de cumplir estas cuatro reglas.

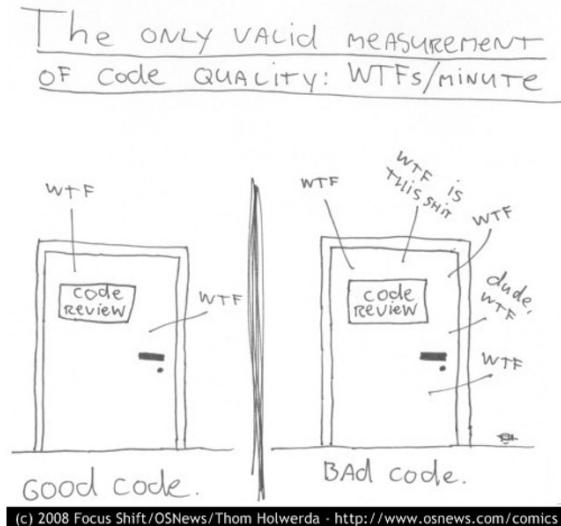
SECCIÓN I: CLEAN CODE

¿Qué es Clean Code?

“Programar es el arte de decirle a otro humano lo que quieres que el ordenador haga”. – Donald Knuth²²

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. Martin²³, también conocido como Uncle Bob, publicó su libro “Clean Code: A Handbook of Agile Software Craftsmanship²⁴” 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

²²https://es.wikipedia.org/wiki/Donald_Knuth

²³<https://twitter.com/unclebobmartin>

²⁴<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*²⁵

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.

²⁵<https://twitter.com/carlosble>

Variables, nombres y ámbito

“Nuestro código tiene que ser simple y directo, debería leerse con la misma facilidad que un texto bien escrito”. – Grady Booch²⁶

Nuestro código debería poder leerse con la misma facilidad con la que leemos un texto bien escrito, es por ello que escoger buenos nombres, hacer un uso correcto de la declaración de las variables y entender el concepto de ámbito es fundamental en JavaScript.



Viñeta de Commit Strip sobre el nombrado de variables.

Los nombres de variables, métodos y clases deben seleccionarse con cuidado para que den expresividad y significado a nuestro código. En este capítulo, además de profundizar en algunos detalles importantes relacionados con las variables y su

²⁶https://es.wikipedia.org/wiki/Grady_Booch

ámbito, veremos algunas pautas y ejemplos para tratar de mejorar a la hora de escoger buenos nombres.

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

En JavaScript clásico, antes de ES6, únicamente teníamos una forma de declarar las variables y era a través del uso de la palabra *var*. A partir de ES6 se introducen *let* y *const*, con lo que pasamos a tener tres palabras reservadas para la declaración de variables.

Lo ideal sería tratar de evitar a toda costa el uso de *var*, ya que no permite definir variables con un ámbito de bloque, lo cual puede derivar en comportamientos inesperados y poco intuitivos. Esto no ocurre con las variables definidas con *let* y *const*, que sí permiten definir este tipo de ámbito, como veremos al final de este capítulo.

La diferencia entre *let* y *const* radica en que a esta última no se le puede reasignar su valor, aunque sí modificarlo. Es decir, se puede modificar (mutar) en el caso de un objeto, pero no si se trata de un tipo primitivo. Por este motivo, usar *const* en variables en las que no tengamos pensado cambiar su valor puede ayudarnos a mejorar la intencionalidad de nuestro código.

Ejemplo de uso de *var*

```
1 var variable = 5;
2 {
3     console.log('inside', variable); // 5
4     var variable = 10;
5 }
6
7 console.log('outside', variable); // 10
8 variable = variable * 2;
9 console.log('changed', variable); // 20
```

Puedes acceder al ejemplo interactivo [desde aquí²⁷](#)

²⁷<https://repl.it/@SoftwareCrafter/CLEAN-CODE-var>

Ejemplo de uso de let

```
1 let variable = 5;
2
3 {
4     console.log('inside', variable); // error
5     let variable = 10;
6 }
7
8 console.log('outside', variable); // 5
9 variable = variable * 2;
10 console.log('changed', variable); // 10
```

Puedes acceder al ejemplo interactivo [desde aquí²⁸](#)

Ejemplo de uso de const

```
1 const variable = 5;
2 variable = variable*2; // error
3 console.log('changed', variable); // doesn't get here
```

Puedes acceder al ejemplo interactivo [desde aquí²⁹](#)

Nombres pronunciables y expresivos

Los nombres de las variables, imprescindiblemente en inglés, deben ser pronunciables. Esto quiere decir que no deben ser abreviaturas ni llevar guión bajo o medio, priorizando el estilo CamelCase. Por otro lado, debemos intentar no ahorrarnos caracteres en los nombres, la idea es que sean lo más expresivos posible.

²⁸<https://repl.it/@SoftwareCrafter/CLEAN-CODE-let>

²⁹<https://repl.it/@SoftwareCrafter/CLEAN-CODE-const>

Nombres pronunciables y expresivos

```
1 //bad
2 const yyyyMMddstr = moment().format('YYYY/MM/DD');
3
4 //better
5 const currentDate = moment().format('YYYY/MM/DD');
```

Ausencia de información técnica en los nombres

Si estamos construyendo un *software* de tipo vertical (orientado a negocio), debemos intentar que los nombres no contengan información técnica en ellos, es decir, evitar incluir información relacionada con la tecnología, como el tipo de dato o [la notación húngara³⁰](#), el tipo de clase, etc. Esto sí se admite en desarrollo de *software* horizontal o librerías de propósito general.

Evitar que los nombres contengan información técnica

```
1 //bad
2 class AbstractUser(){...}
3
4 //better
5 class User(){...}
```

Descargado en: www.detodoprogramacion.org

Establecer un lenguaje ubicuo

El término “lenguaje ubicuo” lo introdujo Eric Evans en su famoso libro sobre DDD, *Implementing Domain-Driven Design*, también conocido como “el libro rojo del DDD”. Aunque el DDD queda fuera del ámbito de este libro, creo que hacer uso del lenguaje ubicuo es tremadamente importante a la hora de obtener un léxico coherente.

³⁰https://es.wikipedia.org/wiki/Notaci%C3%B3n_h%C3%A1ngara

El lenguaje ubicuo es un proceso en el cual se trata de establecer un lenguaje común entre programadores y *stakeholders* (expertos de dominio), basado en las definiciones y terminología que se emplean en el negocio.

Una buena forma de comenzar con este el proceso podría ser crear un glosario de términos. Esto nos permitirá, por un lado, mejorar la comunicación con los expertos del negocio, y por otro, ayudarnos a escoger nombres más precisos para mantener una nomenclatura homogénea en toda la aplicación.

Por otro lado, debemos usar el mismo vocabulario para hacer referencia al mismo concepto, no debemos usar en algunos lados *User*, en otro *Client* y en otro *Customer*, a no ser que representen claramente conceptos diferentes.

Léxico coherente

```
1 //bad
2 getUserInfo();
3 getClientData();
4 getCustomerRecord();
5
6 //better
7 getUser()
```

Nombres según el tipo de dato

Arrays

Los *arrays* son una lista iterable de elementos, generalmente del mismo tipo. Es por ello que pluralizar el nombre de la variable puede ser una buena idea:

Arrays

```
1 //bad
2 const fruit = ['manzana', 'platano', 'fresa'];
3 // regular
4 const fruitList = ['manzana', 'platano', 'fresa'];
5 // good
6 const fruits = ['manzana', 'platano', 'fresa'];
7 // better
8 const fruitNames = ['manzana', 'platano', 'fresa'];
```

Booleanos

Los *booleanos* solo pueden tener 2 valores: verdadero o falso. Dado esto, el uso de prefijos como *is*, *has* y *can* ayudará inferir el tipo de variable, mejorando así la legibilidad de nuestro código.

Booleanos

```
1 //bad
2 const open = true;
3 const write = true;
4 const fruit = true;
5
6 // good
7 const isOpen = true;
8 const canWrite = true;
9 const hasFruit = true;
```

Números

Para los números es interesante escoger palabras que describan números, como *min*, *max* o *total*:

Números

```
1 //bad
2 const fruits = 3;
3
4 //better
5 const maxFruits = 5;
6 const minFruits = 1;
7 const totalFruits = 3;
```

Funciones

Los nombres de las funciones deben representar acciones, por lo que deben construirse usando el verbo que representa la acción seguido de un sustantivo. Estos deben de ser descriptivos y, a su vez, concisos. Esto quiere decir que el nombre de la función debe expresar lo que hace, pero también debe de abstraerse de la implementación de la función.

Funciones

```
1 //bad
2 createUserIfNotExists()
3 updateUserIfNotEmpty()
4 sendEmailIfFieldsValid()
5
6 //better
7 createUser(...)
8 updateUser(...)
9 sendEmail()
```

En el caso de las funciones de acceso, modificación o predicado, el nombre debe ser el prefijo *get*, *set* e *is*, respectivamente.

Funciones de acceso, modificación o predicado

```
1 getUser()  
2 setUser(...)  
3 isValidUser()
```

Get y set

En el caso de los *getters* y *setters*, sería interesante hacer uso de las palabras clave *get* y *set* cuando estamos accediendo a propiedades de objetos. Estas se introdujeron en ES6 y nos permiten definir métodos accesores:

Get y set

```
1 class Person {  
2     constructor(name) {  
3         this._name = name;  
4     }  
5  
6     get name() {  
7         return this._name;  
8     }  
9  
10    set name(newName) {  
11        this._name = newName;  
12    }  
13 }  
14  
15 let person = new Person('Miguel');  
16 console.log(person.name); // Outputs 'Miguel'
```

Clases

Las clases y los objetos deben tener nombres formados por un sustantivo o frases de sustantivo como *User*, *UserProfile*, *Account* o *AdressParser*. Debemos evitar nombres genéricos como *Manager*, *Processor*, *Data* o *Info*.

Hay que ser cuidadosos a la hora de escoger estos nombres, ya que son el paso previo a la hora de definir la responsabilidad de la clase. Si escogemos nombres demasiado genéricos tendemos a crear clases con múltiples responsabilidades.

Ámbito o *scope* de las variables

Además de escribir nombres adecuados para las variables, es fundamental entender cómo funciona su *scope* en JavaScript. El *scope*, que se traduce como “ámbito” o “alcance” en español, hace referencia a la visibilidad y a la vida útil de una variable. El ámbito, en esencia, determina en qué partes de nuestro programa tenemos acceso a una cierta variable.

En JavaScript existen principalmente tres tipos de ámbitos: el ámbito global, el ámbito local o de función y el ámbito de bloque.

Ámbito global

Cualquier variable que no esté dentro de un bloque de una función, estará dentro del ámbito global. Dichas variables serán accesibles desde cualquier parte de la aplicación:

Ambito global

```
1 let greeting = 'hello world!';  
2  
3 function greet(){  
4     console.log(greeting);  
5 }  
6  
7 greet(); //Hello world;
```

Ámbito de bloque

Los bloques en Javascript se delimitan mediante llaves, una de apertura ‘{’, y otra de cierre ‘}’. Como comentamos en el apartado de “Uso correcto de *var*, *let* y *const*”, para definir variables con alcance de bloque debemos hacer uso de *let* o *const*:

Ámbito de bloque

```
1  {
2      let greeting = "Hello world!";
3      var lang = "English";
4      console.log(greeting); //Hello world!
5  }
6
7  console.log(lang); //English
8  console.log(greeting); //Uncaught ReferenceError: greeting is not defined
```

En este ejemplo queda patente que las variables definidas con `var` se pueden emplear fuera del bloque, ya que este tipo de variables no quedan encapsuladas dentro de los bloques. Por este motivo, y de acuerdo con lo mencionado anteriormente, debemos evitar su uso para no encontrarnos con comportamientos inesperados.

Ámbito estático vs. dinámico

El ámbito de las variables en JavaScript tiene un comportamiento de naturaleza estática. Esto quiere decir que se determina en tiempo de compilación en lugar de en tiempo de ejecución. Esto también se suele denominar **ámbito léxico (*lexical scope*)**. Veamos un ejemplo:

Ámbito estático vs. dinámico

```
1 const number = 10;
2 function printNumber() {
3     console.log(number);
4 }
5
6 function app() {
7     const number = 5;
8     printNumber();
9 }
10
11 app(); //10
```

En el ejemplo, `console.log (number)` siempre imprimirá el número 10 sin importar desde dónde se llame la función `printNumber()`. Si JavaScript fuera un lenguaje con el ámbito dinámico, `console.log(number)` imprimiría un valor diferente dependiendo de dónde se ejecutará la función `printNumber()`.

Hoisting

En JavaScript las declaraciones de las variables y funciones se asignan en memoria en tiempo de compilación; a nivel práctico es como si el intérprete moviera dichas declaraciones al principio de su ámbito. Este comportamiento es conocido como **hoisting**. Gracias al *hoisting* podríamos ejecutar una función antes de su declaración:

Hoisting

```
1 greet(); //”Hello world”;
2 function greet(){
3     let greeting = ‘Hello world!’;
4     console.log(greeting);
5 }
```

Al asignar la declaración en memoria es como si “subiera” la función al principio de su ámbito:

Hoisting

```
1 function greet(){
2     let greeting = ‘Hello world!’;
3     console.log(greeting);
4 }
5 greet(); //”Hello world”;
```

En el caso de las variables, el *hoisting* puede generar comportamientos inesperados, ya que como hemos dicho solo aplica a la declaración y no a su asignación:

Hoisting

```
1 var greet = "Hi";
2 (function () {
3     console.log(greet); // "undefined"
4     var greet = "Hello";
5     console.log(greet); // "Hello"
6 })();
```

En el primer `console.log` del ejemplo, lo esperado es que escriba “Hi”, pero como hemos comentado, el intérprete “eleva” la declaración de la variable a la parte superior de su *scope*. Por lo tanto, el comportamiento del ejemplo anterior sería equivalente a escribir el siguiente código:

Hoisting

```
1 var greet = "Hi";
2 (function () {
3     var greet;
4     console.log(greet); // "undefined"
5     greet = "Hello";
6     console.log(greet); // "Hello"
7 })();
```

He usado este ejemplo porque creo que es muy ilustrativo para explicar el concepto de **hoisting**, pero volver a declarar una variable con el mismo nombre y además usar `var` para definirlas es muy mala idea.

Funciones

“Sabemos que estamos desarrollando código limpio cuando cada función hace exactamente lo que su nombre indica”. – Ward Cunningham³¹

Las funciones son la entidad organizativa más básica en cualquier programa. Por ello, deben resultar sencillas de leer y de entender, además de transmitir claramente su intención. Antes de profundizar en cómo deberían ser, exploraremos las diferentes maneras en las que se pueden definir: declaración, expresiones y funciones *arrow*. Además, en esta última explicaremos el funcionamiento del objeto *this*, del cual podemos adelantar que tiene un comportamiento poco intuitivo en JavaScript.

Declaración de una función

La forma clásica de definir funciones en JavaScript es a través de la declaración de funciones. Esta se declara con la palabra clave *function* seguida del nombre de la función y los paréntesis. Puede tener o no parámetros. A continuación, entre llaves, tendremos el conjunto de instrucciones y opcionalmente la palabra clave *return* y el valor de retorno.

Declaración de una función

```
1 function doSomething(){  
2     return "Doing something";  
3 }  
4  
5 doSomething() // "Doing something"
```

Expresión de una función

Una expresión de una función tiene una sintaxis similar a la declaración de una función, con la salvedad de que asignamos la función a una variable:

³¹https://es.wikipedia.org/wiki/Ward_Cunningham

Expresión de una función

```
1 const doSomething = function(){
2     return "Doing something";
3 }
4
5 doSomething() // "Doing something"
```

Expresiones con funciones flecha (*arrow functions*)

Con la aparición de ES6, se añadió al lenguaje la sintaxis de la función flecha, una forma de definir funciones mucho más legible y concisa.

Expresión con funciones flecha

```
1 const doSomething = () => "Doing something";
2 // El return está implícito si no añadimos las llaves.
```

Las *arrow functions* son ideales para declarar **expresiones lambda (funciones en línea)**, puesto que se reduce el ruido en la sintaxis y se mejora la expresividad e intencionalidad del código.

Expresiones lambda

```
1 // Sin arrows functions
2 [1, 2, 3].map(function(n){return n * 2})
3 // Con arrows functions
4 [1, 2, 3].map(n => n * 2)
```

Las *arrow functions* también son muy útiles a la hora de escribir **funciones currificadas**. Como sabrás, una función de este tipo toma un argumento, devuelve una función que toma el siguiente argumento, y así sucesivamente. Con las *arrow functions*, este proceso puede acortarse, lo que permitirá obtener un código mucho más legible.

Funciones currificadas y aplicación parcial

```
1 //Sin arrow functions
2 function add(x){
3     return function(y){
4         return x + y;
5     }
6 }
7 //con arrow functions
8 const add = x => y => x + y;
9
10 const addTwo = add(2);
11 console.log(addTwo(5))//7
```

Funcionamiento del objeto *this* en *arrow functions*

Otra característica interesante de las *arrow functions* es que cambian el comportamiento por defecto del objeto *this* en JavaScript. Cuando creamos una *arrow function*, su valor de *this* queda asociado permanentemente al valor de *this* de su ámbito externo inmediato; *window* en el caso del ámbito global del navegador o *global* en el caso del ámbito global de *NodeJS*:

this en arrow functions

```
1 const isWindow = () => this === window;
2 isWindow(); // true
```

En el caso de encontrarse en el ámbito local de un método, el valor de *this* sería el valor del ámbito de la función. Veamos un ejemplo:

this en arrow functions

```
1 const counter = {  
2     number: 0,  
3     increase() {  
4         setInterval(() => ++this.number, 1000);  
5     }  
6 };  
7  
8 counter.increment(); //1 2 3 4 5
```

Dentro de la *arrow function*, el valor de *this* es el mismo que en el método *increase()*. Aunque esto pueda parecer el comportamiento esperado, no sucedería así si no empleáramos *arrow functions*. Veamos el mismo ejemplo haciendo uso de una función creada con la palabra clave *function*:

this en arrow functions

```
1 const counter = {  
2     number: 0,  
3     increase() {  
4         setInterval(function(){ ++this.number}, 1000);  
5     }  
6 };  
7  
8 counter.increase(); //NaN NaN NaN ...
```

Aunque *NaN (not a number)* no es el resultado intuitivo, tiene sentido en JavaScript, ya que dentro de *setInterval()* *this* ha perdido la referencia al objeto *counter*. Antes de la aparición de las *arrow functions*, este problema con los *callbacks* se solía corregir haciendo una copia del objeto *this*:

this en arrow functions

```
1 const counter = {  
2     number: 0,  
3     increase() {  
4         const that = this;  
5         setInterval(function(){ ++that.number}, 1000);  
6     }  
7 };  
8  
9 counter.increment(); //1 2 3 ...
```

O, “bindeando” el objeto *this*, mediante la función *bind*:

this en arrow functions

```
1 const counter = {  
2     number: 0,  
3     increase() {  
4         setInterval(function(){ ++this.number}.bind(this), 1000);  
5     }  
6 };  
7  
8 counter.increment();
```

En este ejemplo queda demostrado que ambas soluciones generan mucho ruido, por ello utilizar las *arrow functions* en los *callbacks* en los que se haga uso de *this* se vuelve imprescindible.

Funciones autoejecutadas IIFE

Una forma de definir funciones menos conocida es a través de las IIFE. Las IIFE (*Immediately-Invoked Function Expressions*), o funciones autoejecutadas en castellano, son funciones que se ejecutan al momento de definirse:

Funciones autoejecutadas IIFE

```
1 (function(){
2   // ... do something
3 })()
```

Este patrón era muy utilizado para crear un ámbito de bloque antes de que se introdujeran *let* y *const*. Desde la aparición de ES6 esto no tiene demasiado sentido, pero es interesante conocerlo:

Funciones autoejecutadas IIFE

```
1 (function() {
2   var number = 42;
3 })();
4
5 console.log(number); // ReferenceError
```

Cómo hemos visto en la sección de ámbito de bloque, el uso de *let* y *const* para definirlo es mucho más intuitivo y conciso:

Bloques

```
1 {
2   let number = 42;
3 }
4
5 console.log(number); // ReferenceError
```

Parámetros y argumentos

Los argumentos son los valores con los que llamamos a las funciones, mientras que los parámetros son las variables nombradas que reciben estos valores dentro de nuestra función:

Parámetros y argumentos

```
1 const double = x => x * 2; // x es el parámetro de nuestra función
2 double(2); // 2 es el argumento con el que llamamos a nuestra función
```

Limita el número de argumentos

Una recomendación importante es la de limitar el número de argumentos que recibe una función. En general deberíamos limitarnos a tres parámetros como máximo. En el caso de tener que exceder este número, podría ser una buena idea añadir un nivel más de indirección a través de un objeto:

Limita el número de argumentos

```
1 function createMenu(title, body, buttonText, cancellable) {
2   // ...
3 }
4
5 function createMenu({ title, body, buttonText, cancellable }) {
6   // ...
7 }
8
9 createMenu({
10   title: 'Foo',
11   body: 'Bar',
12   buttonText: 'Baz',
13   cancellable: true
14 });
15 {title="Parámetros y argumentos", lang=javascript}
```

Generalmente, como vimos en el capítulo de variables y nombres, debemos evitar hacer uso de abreviaciones salvo cuando se trate de expresiones lambda (funciones en línea), ya que su ámbito es muy reducido y no surgirían problemas de legibilidad:

Limita el número de argumentos

```
1 const numbers = [1, 2, 3, 4, 5];
2 numbers.map(n => n * 2); // El argumento de map es una expresión lambda.
```

Parámetros por defecto

Desde ES6, JavaScript permite que los parámetros de la función se inicialicen con valores por defecto.

Parámetros por defecto

```
1 //Con ES6
2 function greet(text = 'world') {
3     console.log('Hello ' + text);
4 }
5 greet(); // sin parámetro. Hello world
6 greet(undefined); // undefined. Hello world
7 greet('crafter') // con parámetro. Hello crafter
```

En JavaScript clásico, para hacer algo tan sencillo como esto teníamos que comprobar si el valor está sin definir y asignarle el valor por defecto deseado:

Parámetros por defecto

```
1 //Antes de ES6
2 function greet(text){
3     if(typeof text === 'undefined')
4         text = 'world';
5
6     console.log('Hello ' + text);
7 }
```

Aunque no debemos abusar de los parámetros por defecto, esta sintaxis nos puede ayudar a ser más concisos en algunos contextos.

Parámetro *rest* y operador *spread*

El operador ... (tres puntos) es conocido como el **parámetro *rest*** o como **operador *spread* (propagación en español)**, dependiendo de dónde se emplee.

El parámetro *rest* unifica los argumentos restantes en la llamada de una función cuando el número de argumentos excede el número de parámetros declarados en esta.☒

En cierta manera, el parámetro *rest* actúa de forma contraria a *spread* (operador propagación en español): mientras que *spread* “expande” los elementos de un *array* u objeto dado, *rest* unifica un conjunto de elementos en un *array*.

Parámetro *rest*

```
1 function add(x, y) {  
2     return x + y;  
3 }  
4  
5 add(1, 2, 3, 4, 5) //3  
6  
7 function add(...args){  
8     return args.reduce((previous, current)=> previous + current, 0)  
9 }  
10  
11 add(1, 2, 3, 4, 5) //15  
12  
13 //El parámetro rest es el último parámetro de la función y, como hemos \  
14 mencionado, se trata de un array:  
15  
16 function process(x, y, ...args){  
17     console.log(args)  
18 }  
19 process(1, 2, 3, 4, 5);//[3, 4, 5]
```

Al igual que los parámetros por defecto, esta característica se introdujo en ES6. Para poder acceder a los argumentos adicionales en JavaScript clásico disponemos del objeto **arguments**:

Objeto arguments

```
1 function process(x, y){  
2     console.log(arguments)  
3 }  
4  
5 process(1, 2, 3, 4, 5);//[1, 2, 3, 4, 5]
```

El objeto **arguments** presenta algunos problemas. El primero de ellos es que, aunque parece un *array*, no lo es, y por consiguiente no implementa las funciones de *array.prototype*. Además, a diferencia de *rest*, puede sobrescribirse y no contiene los argumentos restantes, sino todos ellos. Por ello suele desaconsejarse su uso.

Por otro lado, el **operador spread** divide un objeto o un array en múltiples elementos individuales. Esto permite expandir expresiones en situaciones donde se esperan múltiples valores como en llamadas a funciones o en array literales:

Operador spread

```
1 function doStuff (x, y, z) { }  
2 const args = [0, 1, 2];  
3 //con spread  
4 doStuff(...args);  
5  
6 //sin spread  
7 doStuff.apply(null, args);  
8  
9 //spread en funciones de math  
10 const numbers = [9, 4, 7, 1];  
11 Math.min(...numbers);//1
```

Spread también nos permite clonar objetos y arrays de una forma muy simple y expresiva:

Descargado en: www.detodoprogramacion.org

Operador spread

```
1 const post = {title: "Operador spread", content:"lorem ipsum..."}  
2 //clonado con Object.assign();  
3 const postCloned = Object.assign({}, book);  
4 //clonado con el spread operator  
5 const postCloned = { ...book };  
6  
7 const myArray = [1, 2, 3];  
8 //clonado con slice()  
9 const myArrayCloned = myArray.slice();  
10 //clonado con el spread operator()  
11 const myArrayCloned = [ ...myArray ];
```

También podemos usar el operador spread para concatenar arrays:

Operador spread

```
1 const arrayOne = [1, 2, 3];  
2 const arrayTwo = [4, 5, 6];  
3  
4 //concatenación con concat()  
5 const myArray = arrayOne.concat(arrayTwo); //[1, 2, 3, 4, 5, 6]  
6  
7 //concatenacion con spread operator  
8 const myArray = [...arrayOne, ...arrayTwo]; //[1, 2, 3, 4, 5, 6]
```

Tamaño y niveles de indentación

La simplicidad es un pilar fundamental a la hora de escribir buen código y, por ello, una de las recomendaciones clave es que nuestras funciones sean de un tamaño reducido.

Definir un número exacto es complicado: en ocasiones escribo funciones de una sola línea, aunque normalmente suelen tener 4 ó 5 líneas. Esto no quiere decir que nunca

escriba funciones de mayor tamaño, por ejemplo de 15 ó 20 líneas. Sin embargo, cuando alcanzo esta cifra intento analizar si puedo dividirla en varias funciones.

Si tus funciones, como norma general, suelen tener un tamaño demasiado grande o demasiados niveles de indentación, es probable que hagan demasiadas cosas. Esto nos lleva a otra recomendación, quizás la más importante: **las funciones deben hacer una única cosa y hacerla bien**. Otro punto fundamental para lograr que nuestras funciones sigan siendo simples es tratar de limitar los niveles de indentación a 1 ó 2 niveles. Para ello debemos evitar la anidación de condicionales y bucles. Esto nos permitirá mantener a raya el “[código espagueti](#)³²” además de reducir la [complejidad ciclomática](#)³³ de la función.

Veamos un ejemplo de cómo NO deberían ser nuestras funciones:

Tamaño y niveles de indentación

```
1 const getPayAmount = () => {
2     let result;
3     if (isDead){
4         result = deadAmount();
5     }
6     else {
7         if (isSeparated){
8             result = separatedAmount();
9         }
10    else {
11        if (isRetired){
12            result = retiredAmount();
13        }
14    else{
15        result = normalPayAmount();
16    }
17    }
18 }
19 return result;
20 }
```

³²https://es.wikipedia.org/wiki/C%C3%B3digo_espagueti

³³https://es.wikipedia.org/wiki/Complejidad_ciclom%C3%A1tica

Cláusulas de guarda

Las cláusulas de guarda, también conocidas como aserciones o precondiciones, son una pieza de código que comprueba una serie de condiciones antes de continuar con la ejecución de la función.

En el ejemplo anterior, como puedes observar, tenemos demasiados condicionales anidados. Para resolverlo podríamos sustituir los casos *edge* por cláusulas de guarda:

Cláusulas de guarda

```
1 const getPayAmount = () => {
2   if (isDead)
3     return deadAmount();
4
5   if (isSeparated)
6     return separatedAmount();
7
8   if (isRetired)
9     return retiredAmount();
10
11  return normalPayAmount();
12 }
```

Evita el uso de *else*

Otra estrategia que suelo utilizar para evitar la anidación es no emplear la palabra clave *else*. En mis proyectos intento evitarlo siempre que sea posible; de hecho, he trabajado en proyectos con miles de líneas de código sin utilizar un solo *else*. Para lograrlo, suelo priorizar el estilo declarativo, hacer uso de las cláusulas de guarda cuando uso estructuras condicionales o reemplazo las estructuras *if/else* por el operador **ternario**, lo que da lugar a un código mucho más comprensible y expresivo:

Operador ternario

```
1 //if/else
2 const isRunning = true;
3 if(isRunning){
4     stop()
5 }
6 else{
7     run()
8 }
9 //operador ternario
10 isRunning ? stop() : run()
```

Cuando hagas uso de este operador debes intentar mantener la expresión lo más simple posible ya que, de lo contrario, podría volverse poco legible.

Prioriza las condiciones asertivas

Aunque este apartado no está directamente relacionado con los niveles de indentación, creo que es interesante mencionarlo, ya que nos ayuda a mejorar la legibilidad de los condicionales.

La evidencia nos dice que las frases afirmativas suelen ser más fáciles de entender que las negativas, por esta razón deberíamos invertir, siempre que sea posible, las condiciones negativas para convertirlas en afirmativas:

Condiciones asertivas

```
1 //Negative
2 if(!canNotFormat){
3     format()
4 }
5
6 //Positive
7 if(canFormat){
8     format()
9 }
```

Estilo declarativo frente al imperativo

Aunque JavaScript no es un lenguaje funcional puro, sí que nos ofrece algunos elementos de la [programación funcional³⁴](#) que nos permiten escribir un código mucho más declarativo. Una buena práctica podría ser priorizar las funciones de alto nivel map, filter y reduce sobre las estructuras control y condicionales. Esto, además de favorecer la composición, nos permitirá obtener funciones mucho más expresivas y de tamaño más reducido.

Prioriza el estilo declarativo frente al imperativo

```
1 const orders = [
2     { productTitle: "Product 1", amount: 10 },
3     { productTitle: "Product 2", amount: 30 },
4     { productTitle: "Product 3", amount: 20 },
5     { productTitle: "Product 4", amount: 60 }
6 ];
7
8 //worse
9 function imperative(){
10     let totalAmount = 0;
11
12     for (let i = 0; i < orders.length; i++) {
13         totalAmount += orders[i].amount;
14     }
15
16     console.log(totalAmount); // 120
17 }
18 //better
19 function declarative(){
20     function sumAmount(currentAmount, order){
21         return currentAmount + order.amount;
22     }
23
24     function getTotalAmount(orders) {
```

³⁴<https://softwarecrafters.io/javascript/introduccion-programacion-funcional-javascript>

```
25     return orders.reduce(sumAmount, 0);
26 }
27
28 console.log(getTotalAmount(orders)); // 120
29 }
30
31 imperative();
32 declarative();
```

Puedes acceder al ejemplo interactivo [desde aquí³⁵](#)

Funciones anónimas

Como vimos en la sección de los nombres, el valor de un buen nombre es fundamental para la legibilidad. Cuando escogemos un mal nombre sucede todo lo contrario, por ello a veces la mejor forma de escoger buenos nombres es no tener que hacerlo. Aquí es donde entra la fortaleza de las funciones anónimas y por lo que, siempre que el contexto lo permita, deberías utilizarlas. De este modo, evitarás que se propaguen alias y malos nombres por tu código. Veamos un ejemplo:

Usa funciones anónimas

```
1 function main(){
2   const stuffList = [
3     { isEnabled: true, name: 'justin' },
4     { isEnabled: false, name: 'lauren' },
5     { isEnabled: false, name: 'max' },
6   ];
7
8   const filteredStuff = stuffList.filter(stuff => !stuff.isEnabled);
9   console.log(filteredStuff);
10 }
11
12 main();
```

³⁵<https://repl.it/@SoftwareCrafter/CLEAN-CODE-declarativo-vs-imperativo>

La función stuff \Rightarrow !stuff.isEnabled es un predicado tan simple que extraerlo no tiene demasiado sentido. Puedes acceder al ejemplo completo [desde aquí](#)³⁶

Transparencia referencial

Muchas veces nos encontramos con funciones que prometen hacer una cosa y que en realidad generan efectos secundarios ocultos. Esto debemos tratar de evitarlo en la medida de lo posible, para ello suele ser buena idea aplicar el principio de transparencia referencial sobre nuestras funciones.

Se dice que una función cumple el principio de transparencia referencial si, para un valor de entrada, produce siempre el mismo valor de salida. Este tipo de funciones también se conocen como funciones puras y son la base de la programación funcional.

Transparencia referencial

```
1 //bad
2 function withoutReferentialTransparency(){
3     let counter = 1;
4
5     function increaseCounter(value) {
6         counter = value + 1;
7     }
8
9     increaseCounter(counter);
10    console.log(counter); // 2
11 }
12
13 //better
14 function withReferentialTransparency(){
15     let counter = 1;
16
17     function increaseCounter(value) {
18         return value + 1;
19     }
```

³⁶<https://repl.it/@SoftwareCrafter/CLEAN-CODE-funciones-anonimas>

```
20
21   console.log(increaseCounter(counter)); // 2
22   console.log(counter); // 1
23 }
24
25 withoutReferentialTransparency();
26 withReferentialTransparency();
```

Puedes acceder al ejemplo completo [desde aquí³⁷](#)

Principio DRY

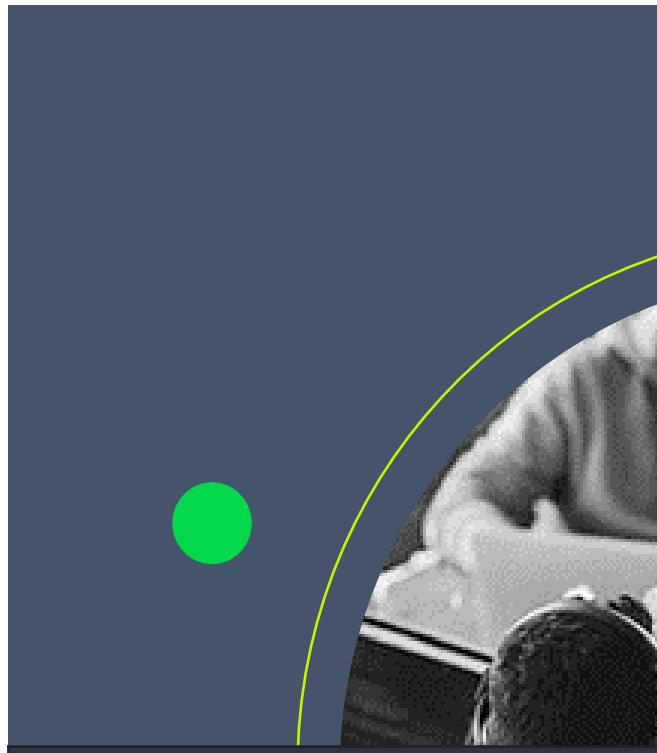
Teniendo en cuenta que la duplicación de código suele ser la raíz de múltiples problemas, una buena práctica sería la implementación del principio DRY (don't repeat yourself). Este principio, que en español significa no repetirse, nos evitará múltiples quebraderos de cabeza como tener que testear lo mismo varias veces, además de ayudarnos a reducir la cantidad de código a mantener.

Para ello lo ideal sería extraer el código duplicado a una clase o función y utilizarlo donde nos haga falta. Muchas veces esta duplicidad no será tan evidente y será nuestra experiencia la que nos ayude a detectarla, no tengas miedo a refactorizar cuando detectes estas situaciones.

Principio DRY

```
1 const reportData = {
2   name: "Software Crafters",
3   createdAt: new Date(),
4   purchases: 100,
5   conversionRate: 10,
6 }
7
8 function withOutDRY(){
9   function showReport(reportData) {
10     const reportFormatted =
```

³⁷<https://repl.it/@SoftwareCrafter/CLEAN-CODE-transparencia-referencial>



El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la
Programación Java,
C/C++/C#, Visual.Net, SQL,
Python, Javascript, Oracle,
Algoritmos, CSS, Desarrollo
Web, Joomla, jquery, Ajax y
Mucho Mas...

VISITA

www.detodoprogramacion.org
www.detodopython.com
www.gratiscodigo.com



```
11     Name: ${reportData.name}
12     Created at: ${reportData.createdAt}
13     Purchases: ${reportData.purchases}
14     Conversion Rate: ${reportData.conversionRate}%
15     console.log("Showing report", reportFormatted)
16 }
17
18 function saveReport(reportData) {
19     const reportFormatted = `
20         Name: ${reportData.name}
21         Created at: ${reportData.createdAt}
22         Purchases: ${reportData.purchases}
23         Conversion Rate: ${reportData.conversionRate}%
24     console.log("Saving report...", reportFormatted)
25 }
26
27 showReport(reportData)
28 saveReport(reportData)
29 }
30
31 function withDRY(){
32     function formatReport(reportData){
33         return `
34             Name: ${reportData.name}
35             Created at: ${reportData.createdAt}
36             Purchases: ${reportData.purchases}
37             Conversion Rate: ${reportData.conversionRate}%
38     }
39
40     function showReport(reportData) {
41         console.log("Showing report...", formatReport(reportData));
42     }
43
44     function saveReport(reportData) {
45         console.log("Saving report...", formatReport(reportData));
46     }
```

```
47  
48     showReport(reportData)  
49     saveReport(reportData)  
50 }
```

Puedes acceder al ejemplo completo [desde aquí³⁸](#)

Command-Query Separation (CQS)

El principio de diseño *Command-Query Separation*, separación de comandos y consultas en español, fue introducido por primera vez por Bertrand Meyer en su libro *Object Oriented Software Construction*. La idea fundamental de este principio es que debemos tratar de dividir las funciones de un sistema en dos categorías claramente separadas:

- **Consultas (*queries*)**: son funciones puras que respetan el principio de transparencia referencial, es decir, devuelven un valor y no alteran el estado del sistema. Siempre devuelven un valor.
- **Comandos (*commands*)**: son funciones que cambian el estado intrínseco del sistema, es decir, generan un *side effect*. También pueden ser conocidos como **modificadores (*modifiers*)** o **mutadores (*mutators*)**. No deberían devolver ningún valor (*void*).

Echemos un vistazo a la firma de la siguiente interfaz:

³⁸<https://repl.it/@SoftwareCrafter/CLEAN-CODE-principio-DRY>

Command–Query Separation

```
1 interface UserRepository
2 {
3     Create(user:User): void;
4     GetByEmail(email:string):User;
5     GetAllByName(string name): User[]
6 }
```

Como puedes observar, el método *Create()* no devuelve ningún valor. Su única acción es la de crear un nuevo usuario, muta el estado del sistema y, por lo tanto, se trata de un **comando**.

Por otro lado, las funciones *GetByEmail* y *GetAllByName*, son **consultas** que devuelven un usuario por email o varios usuarios por nombre, respectivamente. Si están bien diseñadas, no deberían generar ningún efecto secundario, es decir, no deberían cambiar el estado del sistema.

El valor principal de este principio reside en que puede ser extremadamente útil separar claramente los comandos de las consultas. Esto nos permitirá reutilizar y componer las consultas en las diferentes partes del código donde las vayamos necesitando. Como consecuencia, obtendremos un código más robusto y libre de duplicidades.

Algoritmos eficientes

Bjarne Stroustrup, inventor de C++ y autor de varios libros, entiende el concepto de código limpio como aquel código que es elegante y eficiente. Es decir, no solo es un placer leerlo sino que además tiene un rendimiento óptimo. Pero, ¿cómo sabemos si nuestro código tiene un rendimiento adecuado? Pues para ello debemos conocer la notación *Big O* y cómo se clasifican, en base a ella, los algoritmos que codificamos.

Notación O grande (*big O*)

La notación *Big-O*, también conocida como notación asintótica o notación Landau (en honor a uno de sus inventores, [Edmund Landau³⁹](#)), se utiliza para medir el

³⁹https://en.wikipedia.org/wiki/Edmund_Landau

rendimiento o la complejidad de un algoritmo.

En esencia se trata de una aproximación matemática que nos ayuda a describir el comportamiento de un algoritmo, tanto temporal como espacial. Es decir, cuánto tiempo va a tardar en ejecutarse o cuánta memoria va a ocupar mientras se ejecuta, basándose en el número de elementos que se deben procesar.

Por ejemplo, si el tiempo de ejecución de un algoritmo crece linealmente con el número de elementos, diremos que el algoritmo es de complejidad $O(n)$. En cambio, si el algoritmo es independiente de la cantidad de datos que se van a procesar, estaremos ante un algoritmo de complejidad $O(1)$. A continuación veremos las notaciones *big O* más comunes, ordenadas de menor a mayor complejidad, junto a algunos ejemplos.

- **$O(1)$ constante:** la operación no depende del tamaño de los datos. Por ejemplo, acceder a un elemento de un *array*.
- **$O(\log n)$ logarítmica:** la complejidad logarítmica se da en casos en los que no es necesario recorrer todos los elementos. Por ejemplo, el algoritmo de [búsqueda binaria⁴⁰](#) en una lista ordenada o atravesar un árbol binario.
- **$O(n)$ lineal:** el tiempo de ejecución es directamente proporcional al tamaño de los datos. Crece en una línea recta. Como ejemplo nos vale cualquier algoritmo que haga uso de un bucle simple, como podría ser una [búsqueda secuencial⁴¹](#).
- **$O(\log n)$:** es algo peor que la lineal, pero no mucho más. Se aplica en el caso de algoritmos de ordenación como [Quicksort⁴²](#) o [Heapsort⁴³](#).
- **$O(n^2)$ cuadrática:** es típico de algoritmos que necesitan realizar una iteración por todos los elementos en cada uno de los elementos que necesita procesar. Por ejemplo, como cualquier algoritmo que haga uso de dos bucles anidados, como podría ser el algoritmo de búsqueda [Bubble Sort⁴⁴](#). En el caso de añadir otro bucle más, el algoritmo pasaría a ser de complejidad cúbica.
- **$O(2^n)$ exponencial:** se trata de funciones que duplican su complejidad con cada elemento añadido al procesamiento. Suele darse en las llamadas recursivas múltiples. Un ejemplo es el cálculo de la serie Fibonacci de forma recursiva. Desarrollaremos el algoritmo de Fibonacci con TDD al final del libro.

⁴⁰[https://es.wikipedia.org/wiki/Búsqueda_binaria#:~:text=Encienciasdelacomputación,valor en un array ordenado.](https://es.wikipedia.org/wiki/B%C3%BAsqueda_binaria#:~:text=Encienciasdelacomputaci%C3%B3n,valor en un array ordenado.)

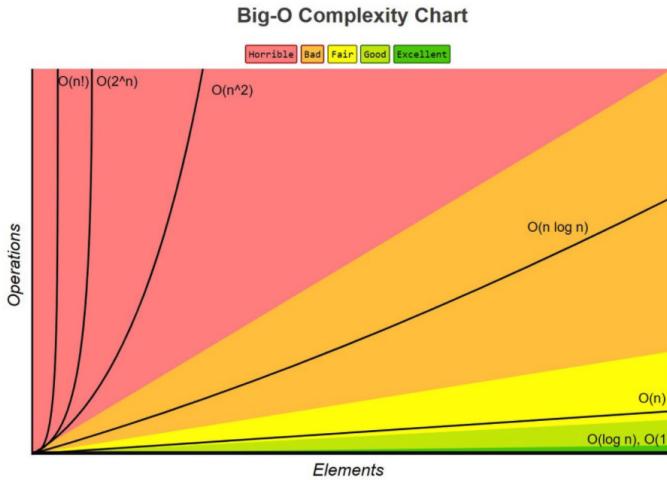
⁴¹[https://es.wikipedia.org/wiki/Búsqueda_lineal#:~:text=En informática%2Clas búsquedas lineal,los elementos hayan sido comparados.](https://es.wikipedia.org/wiki/B%C3%BAsqueda_lineal#:~:text=En inform%C3%A1tica%2Clas b%C3%BAsquedas lineal,los elementos hayan sido comparados.)

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

⁴³<https://es.wikipedia.org/wiki/Heapsort>

⁴⁴https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

- **$O(n!)$ explosión combinatoria:** se trata de algoritmos que no pueden resolverse en tiempo polinómico, también conocidos como **NP (*nondeterministic polynomial time*)**.⁴⁵ Un ejemplo típico es el problema del viajante⁴⁶.



Esquema de deuda técnica de Martin Fowler

Como puedes observar en la gráfica anterior, a partir de la complejidad cuadrática los algoritmos se pueden volver demasiado lentos cuando manejan grandes cantidades de datos. A veces esto puede suponer un *trade-off* que nos coloque en la tesitura de escoger entre un diseño más elegante o más eficiente. En estos casos suele ser buena idea no apresurarse a realizar una optimización prematura, pero todo depende del contexto.

⁴⁵[https://en.wikipedia.org/wiki/NP_\(complexity\)](https://en.wikipedia.org/wiki/NP_(complexity))

⁴⁶https://es.wikipedia.org/wiki/Problema_del_viajante

Clases

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

– Robert C. Martin⁴⁷

Una clase, además de ser una abstracción mediante la cual representamos entidades o conceptos, es un elemento organizativo muy potente. Es por ello que debemos tratar de prestar especial atención a la hora de diseñarlas. Antes de profundizar en cómo deberían diseñarse, vamos a ver algunas peculiaridades de las clases y los objetos en JavaScript.

JavaScript es un lenguaje orientado a objetos basado en prototipos, en lugar de estar basado en clases como tal. En la versión ES6 del lenguaje se introdujo la palabra reservada `class` para definir clases siguiendo el mismo patrón de lenguajes clásicos de POO como Java o C#. Aunque en realidad, esta sintaxis de clases no ofrece realmente una funcionalidad adicional simplemente aportan un estilo más limpio y elegante en comparación con el uso directo de funciones constructoras y/o de la cadena de prototipos.

Prototype y ECMAScript moderno

Todos los objetos de JavaScript enlazan con un objeto prototipo del que heredan todas sus propiedades. Los prototipos permiten integrar muchas de las técnicas clásicas de diseño orientado a objetos, pero antes de ES6, se hacía mediante un mecanismo desordenado y complejo. A continuación, compararemos cómo se implementan algunos elementos de la POO antes y después de ES6.

Constructores y funciones constructoras

Como sabrás, en POO un constructor no es más que una función que nos permite inicializar un objeto de una clase en concreto. Antes de ES6 no teníamos una notación

⁴⁷<https://twitter.com/unclebobmartin>

formal de clases y teníamos que recurrir a las funciones constructoras para este fin. La única particularidad de este tipo de funciones es que utilizan la palabra clave *this* para definir las propiedades que tendrá el objeto y se inicializan con la palabra clave *new*:

Funciones constructoras

```
1 // Antes de ES6
2 function Person(name) {
3     this.name = name;
4 }
5
6 var person = new Person("Miguel");
7 console.log(person.name); // 'Miguel'
```

A partir de ES6 podemos hacer uso de la palabra clave *class* para definir “clases”, tal y como hacemos en otros lenguajes orientados a objetos, aunque internamente JavaScript sigue usando los prototipos. Veamos cómo quedaría el ejemplo anterior con esta sintaxis:

Constructores

```
1 class Person{
2     constructor(person){
3         this.name = name;
4     }
5 }
6
7 const person = new Person("miguel");
8 console.log(person.name); // 'Miguel'
```

Como puedes observar, esta sintaxis es mucho más intuitiva que la anterior. En la actualidad, hacer uso de funciones constructoras en proyectos de JavaScript moderno no tiene ningún sentido ya que empeoran la legibilidad del proyecto.

Métodos

Como ya sabes, los métodos representan operaciones que se pueden realizar con los objetos de una clase en concreto. Antes de ES6, para definirlos, teníamos que asignarlos directamente al objeto *prototype* tras declarar la función constructora:

Métodos

```
1 // Antes de ES6
2 function Person(name) {
3     this.name = name;
4 }
5
6 Person.prototype.greet = function(){
7     return "Hola soy " + this.name;
8 }
9
10 var person = new Person("Miguel");
11 console.log(person.greet()); // 'Hola soy Miguel'
```

La sintaxis de clases de ES6 nos permite realizar esto de una forma más legible y cohesiva:

Métodos

```
1 class Person{
2     constructor(person){
3         this.name = name;
4     }
5
6     greet(){
7         return "Hola soy ${this.name}";
8     }
9 }
10
11 const person = new Person("miguel");
12 console.log(person.greet()); // 'Hola soy Miguel'
```

Herencia y cadena de prototipos

La herencia es una técnica típica de la POO que nos permite definir nuevas clases basadas en otras ya existentes con el objetivo de reutilizar código. Sin embargo, como veremos en este mismo capítulo, la herencia no es la mejor opción para la reutilización de código, aunque sí que existen ciertos contextos en los que se aplica muy bien.

Veamos cómo se implementa la herencia usando prototipos con la sintaxis tradicional de ES5. Para ello, crearemos un objeto programador que hereda del objeto persona (y sí, en JavaScript es más preciso hablar de “herencia entre objetos” que de “clases”) que creamos en los ejemplos anteriores:

Herencia y cadena de prototipos

```
1 // ES5
2 function Programmer(name) {
3     this.name = name;
4 }
5
6 Programmer.prototype = Object.create(Person.prototype);
7
8 Programmer.prototype.writeCode = function(coffee) {
9     if(coffee)
10         console.log( 'Estoy programando ' );
11     else
12         console.log('No puedo, no tengo café.');
13 }
14
15 var programmer = new Programmer("Miguel");
16 programmer.greet(); // 'Hola soy Miguel'
17 programmer.writeCode(); // 'No puedo, no tengo café'
```

Como se observa, en primer lugar definimos una nueva función constructora llamada *Programmer*. A continuación, asignamos a su prototipo un nuevo objeto basado en el prototipo del objeto *Person*, lo que nos permite heredar toda la funcionalidad

implementada en el objeto *Person*. Por último, definimos el método *writeCode(coffee)* del mismo modo que hicimos en el ejemplo anterior.

Queda patente que el uso directo de *prototype* no es nada intuitivo. Veamos cómo queda el mismo ejemplo con la sintaxis de clases:

Herencia y cadena de prototipos

```
1 // ES6
2 class Programmer extends Person{
3     constructor(name){
4         super(name);
5     }
6
7     writeCode(coffee){
8         coffee ? console.log( 'Estoy programando' ) : console.log( 'No puedo,\ \
9         no tengo café.' );
10    }
11 }
12
13 const programmer = new Programmer("Miguel");
14 programmer.greet(); // 'Hola soy Miguel'
15 programmer.writeCode(); // 'No puedo, no tengo café'
```

La sintaxis de clases permite escribir un código más legible e intuitivo. Por cierto, en el constructor de la clase le estamos pasando a la clase padre el parámetro *name* a través de la palabra clave *super*; este tipo de prácticas debemos minimizarlas ya que aumentan la rigidez y el acoplamiento de nuestro código.

Tamaño reducido

Las clases, al igual que vimos en las funciones, deben tener un tamaño reducido. Para conseguir esto debemos empezar por **escoger un buen nombre**. Un nombre adecuado es la primera forma de limitar el tamaño de una clase, ya que nos debe describir la responsabilidad que desempeña la clase.

Otra pauta que nos ayuda a mantener un tamaño adecuado de nuestras clases es tratar de aplicar el **principio de responsabilidad única**. Este principio viene a decir que una clase no debería tener más de una responsabilidad, es decir, no debería tener más de un motivo por el que ser modificada (ampliaremos esta definición en la sección de principios SOLID). Veamos un ejemplo:

Principio de responsabilidad única

```
1 class UserSettings {
2     private user: User;
3     private settings: Settings;
4
5     constructor(user) {
6         this.user = user;
7     }
8
9     changeSettings(settings) {
10        if (this.verifyCredentials()) {
11            // ...
12        }
13    }
14
15    verifyCredentials() {
16        // ...
17    }
18 }
```

La clase *UserSettings* tiene dos responsabilidades: por un lado tiene que gestionar las *settings* del usuario y, además, se encarga del manejo de las credenciales. En este caso podría ser interesante extraer la verificación de las credenciales a otra clase, por ejemplo *UserAuth*, y que dicha clase sea la responsable de gestionar las operaciones relacionadas con el manejo de las credenciales. Nosotros tan solo tendríamos que inyectarla a través del constructor de la clase *UserSettings* y usarla en donde la necesitemos, en este caso en el método *changeSettings*.

Principio de responsabilidad única refactorizado

```
1 class UserAuth{
2     private user: User;
3
4     constructor(user: User){
5         this.user = user
6     }
7
8     verifyCredentials(){
9         //...
10    }
11 }
12
13 class UserSettings {
14     private user: User;
15     private settings: Settings;
16     private auth: UserAuth;
17
18     constructor(user: User, auth:UserAuth) {
19         this.user = user;
20         this.auth = auth;
21     }
22
23     changeSettings(settings) {
24         if (this.auth.verifyCredentials()) {
25             // ...
26         }
27     }
28 }
```

Esta forma de diseñar las clases nos permite mantener la responsabilidades bien definidas, además de contener el tamaño de las mismas. Profundizaremos en esto en el capítulo de principio de responsabilidad única.

Organización

Las clases deben comenzar con una lista de variables. En el caso de que hayan constantes públicas, estas deben aparecer primero. Seguidamente deben aparecer las variables estáticas privadas y después las de instancia privadas; en el caso de que utilizaremos variables de instancia públicas estas deben ir en último lugar

Los métodos o funciones públicas deberían ir a continuación de la lista de variables. Para ello comenzaremos con el método constructor. En el caso de usar un named constructor, este iría antes y, seguidamente, el método constructor privado. A continuación situaremos las funciones estáticas de la clase y, si dispone de métodos privados relacionados, los situaremos a continuación. Seguidamente irían el resto de métodos de instancia ordenados de mayor a menor importancia, dejando para el final los accesores (getters y setters).

Para este ejemplo usaremos una pequeña clase construida con Typescript, ya que nos facilita la tarea de establecer métodos y variables privadas.

Organización de clases

```
1 class Post {  
2     private title : string;  
3     private content: number;  
4     private createdAt: number;  
5  
6     static create(title:string; content:string){  
7         return new Post(title, content)  
8     }  
9  
10    private constructor(title:string; content:string){  
11        this.setTitle(title);  
12        this.setContent(content);  
13        this.createdAt = Date.now();  
14    }  
15  
16    setTitle(title:string){  
17        if(StringUtils.isNullOrEmpty(title))
```

```
18         throw new Error('Title cannot be empty')
19
20     this.title = title;
21 }
22
23 setContent(content:string){
24     if(StringUtils.isNullOrEmpty((content)))
25         throw new Error('Content cannot be empty')
26
27     this.content = content;
28 }
29
30 getTitle(){
31     return this.title;
32 }
33
34 getContent(){
35     return this.content;
36 }
37 }
```

Prioriza la composición frente a la herencia

Tanto la herencia como la composición son dos técnicas muy comunes aplicadas en la reutilización de código. Como sabemos, la herencia permite definir una implementación desde una clase padre, mientras que la composición se basa en ensamblar objetos diferentes para obtener una funcionalidad más compleja.

Optar por la composición frente a la herencia nos ayuda a mantener cada clase encapsulada y centrada en una sola tarea (principio de responsabilidad), favoreciendo la modularidad y evitando el acoplamiento de dependencias. Un alto acoplamiento no solo nos obliga a arrastrar con dependencias que no necesitamos, sino que además limita la flexibilidad de nuestro código a la hora de introducir cambios.

Esto no quiere decir que nunca debas usar la herencia. Hay situaciones en las que la herencia casa muy bien, la clave está en saber diferenciarlas. Una buena forma

de hacer esta diferenciación es preguntándote si la clase que hereda **es** realmente un hijo o simplemente **tiene** elementos del padre. Veamos un ejemplo:

Composición frente a la herencia

```
1 class Employee {  
2     private name: string;  
3     private email: string;  
4  
5     constructor(name:string, email:string) {  
6         this.name = name;  
7         this.email = email;  
8     }  
9  
10    // ...  
11 }  
12  
13 class EmployeeTaxData extends Employee {  
14     private ssn: string;  
15     private salary: number;  
16  
17     constructor(ssn:string, salary:number) {  
18         super();  
19         this.ssn = ssn;  
20         this.salary = salary;  
21     }  
22    // ...  
23 }
```

Como podemos ver, se trata de un ejemplo algo forzado de herencia mal aplicada, ya que en este caso un empleado “tiene” *EmployeeTaxData*, no “es” *EmployeeTaxData*. Si refactorizamos aplicando composición, las clases quedarían de la siguiente manera:

Composición frente a la herencia

```
1 class EmployeeTaxData{  
2     private ssn: string;  
3     private salary: number;  
4  
5     constructor(ssn:string, salary:number) {  
6         this.ssn = ssn;  
7         this.salary = salary;  
8     }  
9     //...  
10 }  
11  
12 class Employee {  
13     private name: string;  
14     private email: string;  
15     private taxData: EmployeeTaxData;  
16  
17     constructor(name:string, email:string) {  
18         this.name = name;  
19         this.email = email;  
20     }  
21  
22     setTaxData(taxData:EmployeeTaxData){  
23         this.taxData = taxData;  
24     }  
25     // ...  
26 }
```

Como podemos observar, la responsabilidad de cada una de las clases queda mucho más definida de esta manera, además de generar un código menos acoplado y modular.

Comentarios y formato

Evita el uso de comentarios

“No comentes el código mal escrito, reescríbelo”. – Brian W. Kernighan⁴⁸

Cuando necesitas añadir comentarios a tu código es porque este no es lo suficientemente autoexplicativo, lo cual quiere decir que no estamos siendo capaces de escoger buenos nombres. Cuando veas la necesidad de escribir un comentario, trata de refactorizar tu código y/o nombrar los elementos del mismo de otra manera.

A menudo, cuando usamos librerías de terceros, APIS, frameworks, etc., nos encontraremos ante situaciones en las que escribir un comentario será mejor que dejar una solución compleja o un hack sin explicación. En definitiva, la idea es que los comentarios sean la excepción, no la regla.

En todo caso, si necesitas hacer uso de los comentarios, lo importante es comentar el porqué, más que comentar el qué o el cómo. Ya que el cómo lo vemos, es el código, y el qué no debería ser necesario si escribes código autoexplicativo. Pero el por qué has decidido resolver algo de cierta manera a sabiendas de que resulta extraño, eso sí que deberías explicarlo.

Formato coherente

“El buen código siempre parece estar escrito por alguien a quien le importa”. – Michael Feathers⁴⁹

En todo proyecto software debe existir una serie de pautas sencillas que nos ayuden a armonizar la legibilidad del código de nuestro proyecto, sobre todo cuando trabajamos en equipo. Algunas de las reglas en las que se podría hacer hincapié son:

⁴⁸https://es.wikipedia.org/wiki/Brian_Kernighan

⁴⁹<https://twitter.com/mfeathers?lang=es>

Problemas similares, soluciones simétricas

Es capital seguir los mismos patrones a la hora de resolver problemas similares dentro del mismo proyecto. Por ejemplo, si estamos resolviendo un CRUD de una entidad de una determinada forma, es importante que para implementar el CRUD de otras entidades sigamos aplicando el mismo estilo.

Densidad, apertura y distancia vertical

Las líneas de código con una relación directa deben ser verticalmente densas, mientras que las líneas que separan conceptos deben de estar separadas por espacios en blanco. Por otro lado, los conceptos relacionados deben mantenerse próximos entre sí.

Lo más importante primero

Los elementos superiores de los ficheros deben contener los conceptos y algoritmos más importantes, e ir incrementando los detalles a medida que descendemos en el fichero.

Indentación

Por último, y no menos importante, debemos respetar la indentación o sangrado. Debemos indentar nuestro código de acuerdo a su posición dependiendo de si pertenece a la clase, a una función o a un bloque de código.

Esto es algo que puede parecer de sentido común, pero quiero hacer hincapié en ello porque no sería la primera vez que me encuentro con este problema. Es más, en la universidad tuve un profesor que, como le entregaras un ejercicio con una mala indentación, directamente ni te lo corregía.

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 término *code smell*, o mal olor en el código, fue acuñado por Kent Beck en uno de los capítulos del famoso libro *Refactoring* de Martin Fowler. El concepto, como puedes imaginar, está relacionado con el de deuda técnica. En este caso los *code smells* hacen referencia a posibles indicios de que algo no está del todo bien planteado en nuestro código y que es probable que debamos refactorizarlo.

Existen múltiples *code smells*, nosotros vamos a centrarnos en algunos de ellos. Si quieres profundizar más en el tema te recomiendo que leas el capítulo de “Bad Smells” del libro *Refactoring*⁵⁰.

El patrón singleton

El patrón singleton es quizás uno de los patrones más conocidos y a la vez más denostados. La intención de este patrón es tratar de garantizar que una clase tenga

⁵⁰<https://amzn.to/33GqLj9>

una única instancia y proporcionar un acceso global a ella. Suele implementar creando en la clase una variable estática que almacena una instancia de si misma, dicha variable se inicializa por primera vez en el constructor o en un *named constructor*.

Patrón singleton

```
1 class Singleton {  
2     constructor(){  
3         if(Singleton.instance){  
4             return Singleton.instance;  
5         }  
6  
7         this.title = "my singleton";  
8         Singleton.instance = this;  
9     }  
10 }  
11  
12 let mySingleton = new Singleton()  
13 let mySingleton2 = new Singleton()  
14  
15 console.log("Singleton 1: ", mySingleton.title);  
16 mySingleton.title = "modified in instance 1"  
17 console.log("Singleton 2: ", mySingleton2.title);
```

Puedes acceder al ejemplo interactivo [desde aquí](#)⁵¹.

Una de las razones por las que emplear patrones Singleton se considera una mala práctica es porque generalmente expone la instancia de nuestro objeto al contexto global de la aplicación, pudiendo ser modificado en cualquier momento y perdiendo el control del mismo.

Otra de las razones es que hacer *unit test* con ellas puede llegar a ser un infierno porque cada test debe ser totalmente independiente al anterior y eso no se cumple, por lo que al mantener el estado, la aplicación se hace difícil de testear.

A pesar de que se pueda seguir usando, suele ser más recomendable separar la gestión del ciclo de vida de la clase de la propia clase.

⁵¹<https://repl.it/@SoftwareCrafter/STUPID-Singleton>

Alto acoplamiento

Seguramente habrás leído o escuchado que un alto acoplamiento entre clases dificulta la mantenibilidad y tolerancia al cambio de un proyecto *software*, y que lo ideal es tener un acoplamiento bajo y buena cohesión pero, ¿a qué se refieren exactamente con estos conceptos?

Acoplamiento y cohesión

La cohesión hace referencia a la relación entre los módulos de un sistema. En términos de clase, podemos decir que presenta alta cohesión si sus métodos están estrechamente relacionados entre sí. Un código con alta cohesión suele ser más *self-contained*, es decir contiene toda las piezas que necesita por lo tanto también suele ser más sencillo de entender. No obstante, si aumentamos demasiado la cohesión, podríamos tender a crear módulos con múltiples responsabilidades.

El acoplamiento, en cambio, hace referencia a la relación que guardan entre sí los módulos de un sistema y su dependencia entre ellos. Si tenemos muchas relaciones entre dichos módulos, con muchas dependencias unos de otros, tendremos un grado de acoplamiento alto. En cambio, si los módulos son independientes unos de otros, el acoplamiento será bajo. Si favorecemos el bajo acoplamiento, obtendremos módulos más pequeños y con responsabilidades más definidas, pero también más dispersos

En el equilibrio está la virtud, es por ello que debemos tratar de favorecer el bajo acoplamiento pero sin sacrificar la cohesión.

Código no testeable

La mayoría de las veces, el código no testeable o difícilmente testeable es causado por un alto acoplamiento y/o cuando no se inyectan las dependencias . Profundizaremos en este último concepto en el principio SOLID de inversión de dependencias.

Aunque hay técnicas específicas para lidiar con estas situaciones, lo ideal es que básicamente viene a decirnos que, para poder cumplir con los test, nuestro diseño debe de tenerlos en cuenta desde el inicio. Así conseguiremos que situaciones problemáticas como alto acomplamiento o dependencias de estado global se manifiestan de manera inmediata. Profundizaremos en esto en la sección de *Unit Testing* y TDD.

Optimizaciones prematuras

“Cuando lleguemos a ese río cruzaremos ese puente”

Mantener abiertas las opciones retrasando la toma de decisiones nos permite darle mayor relevancia a lo que es más importante en una aplicación: las reglas de negocio, donde realmente está el valor. Además, el simple hecho de aplazar estas decisiones nos permitirá tener más información sobre las necesidades reales del proyecto, lo que nos permitirá tomar mejores decisiones ya que estarán basadas en los nuevos requisitos que hayan surgido.

Donald Knuth⁵² decía que la optimización prematura es la raíz de todos los males. Con esto no quiere decir que debamos escribir *software* poco optimizado, sino que no debemos anticiparnos a los requisitos y desarrollar abstracciones innecesarias que puedan añadir complejidad accidental.

Complejidad esencial y complejidad accidental

El antipatrón de diseño accidental *complexity*, o complejidad accidental, es la situación a la que se llega cuando en el desarrollo de un producto *software* se implementa una solución de complejidad mayor a la mínima indispensable.

Lo ideal sería que la complejidad fuese la inherente al problema, dicha complejidad es conocida como complejidad esencial. Pero, lo que suele ocurrir es que acabamos introduciendo complejidad accidental por desconocimiento o por problemas de planificación del equipo, lo que hace que el proyecto se vuelva difícilmente mantenible y poco tolerante al cambio.

Si quieres seguir profundizando en estas ideas, te recomiendo el artículo [No hay balas de plata — Esencia y accidentes de la ingeniería del software](#)⁵³ (Título en inglés: *No Silver Bullet — Essence and Accidents of Software Engineering*). Su autor y ganador del premio Alan Turing, Fred Brooks, dividió las propiedades del *software* en esenciales y accidentales basándose en la descomposición que hizo Aristóteles del conocimiento.

⁵²<https://www-cs-faculty.stanford.edu/~knuth/>

⁵³<http://worrydream.com/refs/Brooks-NoSilverBullet.pdf>

Nombres poco descriptivos

El siguiente principio STUPID es el de *Indescriptive Naming* o nombres poco descriptivos. Básicamente viene a decirnos que los nombres de variables, métodos y clases deben seleccionarse con cuidado para que den expresividad y significado a nuestro código. Ya hemos profundizado en esto en el capítulo de nombres y variables.

Duplicidad de código

El último principio STUPID viene a hacernos referencia al principio DRY (don't repeat yourself), que ya comenté en el apartado de las funciones. Básicamente venía a decirnos que, por norma general, debemos evitar el código duplicado, aunque existen excepciones.

Duplicidad real

El código en la duplicidad real, además de ser idéntico, cumple la misma función. Por lo tanto, si hacemos un cambio, debemos propagarlo de forma manual a todos las partes de nuestro proyecto donde se encuentre dicho código, además, debemos cambiarlo de la misma manera, lo que incrementa las probabilidades de que se produzca un error humano. Este es el tipo de código duplicado que debemos evitar y que tendríamos que unificar.

Duplicidad accidental

Al contrario que en la duplicidad real, la duplicidad accidental es aquella en la que el código puede parecer el mismo, pero en realidad cumple funciones distintas. En este caso el código tiene un motivo para cambiar, ya que si tenemos que introducir un cambio es probable que solo sea necesario modificar alguno de los sitios donde esté dicho código. Este es el tipo de código que debemos evitar y que tendríamos que unificar.

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 Feathers⁵⁴](#), 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.

⁵⁴<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

El primero de los cinco principios, *single responsibility principle* (SRP), principio de responsabilidad única en castellano, viene a decir que una clase debe tener tan solo una única responsabilidad. A finales de los 80, Kent Beck y Ward Cunningham ya aplicaban este principio mediante tarjetas CRC (Class, Responsibility, Collaboration), con las que detectaban responsabilidades y colaboraciones entre módulos.

Tener más de una responsabilidad en nuestras clases o módulos hace que nuestro código sea difícil de leer, de testear y mantener. Es decir, hace que el código sea menos flexible, más rígido y, en definitiva, menos tolerante al cambio.

La mayoría de veces, los programadores aplicamos mal este principio, ya que solemos confundir “tener una única responsabilidad” con “hacer una única cosa”. Es más, ya vimos un principio como este último en el capítulo de las funciones: las funciones deben hacer una única cosa y hacerla bien. Este principio lo usamos para refactorizar funciones de gran tamaño en otras más pequeñas, pero esto no aplica a la hora de diseñar clases o componentes.

¿Qué entendemos por responsabilidad?

El principio de responsabilidad única no se basa en crear clases con un solo método, sino en diseñar componentes que solo estén expuestos a una fuente de cambio. Por lo tanto, el concepto de responsabilidad hace referencia a aquellos actores (fuentes de cambio) que podrían reclamar diferentes modificaciones en un determinado módulo dependiendo de su rol en el negocio. Veamos un ejemplo:

Principio de responsabilidad única

```
1  class UseCase{
2      doSomethingWithTaxes(){
3          console.log("Do something related with taxes ...")
4      }
5
6      saveChangesInDatabase(){
7          console.log("Saving in database ...")
8      }
9
10     sendEmail(){
11         console.log("Sending email ...")
12     }
13 }
14
15 function start(){
16     const myUseCase = new UseCase()
17
18     myUseCase.doSomethingWithTaxes();
19     myUseCase.saveInDatabase();
20     myUseCase.sendEmail();
21 }
22
23 start();
```

Puedes acceder al ejemplo interactivo [desde aquí⁵⁵](#).

En este ejemplo tenemos una clase *UseCase* que consta de tres métodos: *doSomethingWithTaxes()*, *sendEmail()* y *saveChangesInDatabase()*. A primera vista se puede detectar que estamos mezclando tres capas de la arquitectura muy diferenciadas: la lógica de negocio, la lógica de presentación y la lógica de persistencia. Pero, además, esta clase no cumple con el principio de responsabilidad única porque el funcionamiento de cada uno de los métodos son susceptibles a ser cambiados por tres actores diferentes.

⁵⁵<https://repl.it/@SoftwareCrafter/SOLID-SRP>

El método *doSomethingWithTaxes()* podría ser especificado por el departamento de contabilidad, mientras que *sendEmail()* podría ser susceptible a cambio por el departamento de *marketing* y, para finalizar, el método *saveChangesInDatabase()* podría ser especificado por el departamento encargado de la base de datos.

Es probable que no existan, de momento, estos departamentos en tu empresa, y que la persona encargada de asumir estos puestos por ahora seas tú, pero ten en cuenta que las necesidades de un proyecto van evolucionando. Es por esta razón que uno de los valores más importantes del *software* es la tolerancia al cambio. Por ello, aunque estemos trabajando en un proyecto pequeño, debemos hacer el ejercicio de diferenciar las responsabilidades para conseguir que nuestro *software* sea lo suficientemente flexible como para poder satisfacer las nuevas necesidades que puedan aparecer.

Aplicando el SRP

Volviendo al ejemplo, una forma de separar estas responsabilidades podría ser moviendo cada una de las funciones de la clase *UseCase* a otras, tal que así:

Principio de responsabilidad única refactorizado

```
1 class UseCase{
2     constructor(repo, notifier){
3         this.repo = repo;
4         this.notifier = notifier;
5     }
6
7     doSomethingWithTaxes(){
8         console.log("Do something related with taxes . . .")
9     }
10
11    saveChanges(){
12        this.repo.update();
13    }
14
15    notify(){
16        this.notifier.notify("Hi!")
17    }
18}
```

```
17     }
18 }
19
20
21 class Repository{
22     add(){
23         console.log("Adding in database");
24     }
25
26     update(){
27         console.log("Updating in database...");
28     }
29
30     remove(){
31         console.log("Deleting from database ...");
32     }
33
34     find(){
35         console.log("Finding from database ...");
36     }
37 }
38
39
40 class NotificationService{
41     notify(message){
42         console.log("Sending message ...");
43         console.log(message);
44     }
45 }
46
47
48 function start(){
49     const repo = new Repository()
50     const notifier = new NotificationService()
51     const myUseCase = new UseCase(repo, notifier)
52 }
```

```
53     myUseCase.doSomethingWithTaxes();  
54     myUseCase.saveChanges();  
55     myUseCase.notify();  
56 }  
57  
58 start();
```

Puedes acceder al ejemplo interactivo [desde aquí](#)⁵⁶.

La clase *UseCase* pasaría a representar un caso de uso con una responsabilidad más definida, ya que ahora el único actor relacionado con las clases es el encargado de la especificación de la operación *doSomethingWithTaxes()*. Para ello hemos extraído la implementación del resto de operaciones a las clases *Repository* y *NotificationService*.

La primera implementa un repositorio (como podrás comprobar es un repositorio *fake*) y se responsabiliza de todas las operaciones relacionadas con la persistencia. Por otro lado, la clase *NotificationService*, se encargaría de toda la lógica relacionada con las notificaciones al usuario. De esta manera ya tendríamos separadas las tres responsabilidades que habíamos detectado.

Ambas clases se inyectan vía constructor a la clase *UseCase*, pero, como puedes comprobar, se trata de implementaciones concretas, con lo cual el acoplamiento sigue siendo alto. Continuaremos profundizando en esto a lo largo de los siguientes capítulos, sobre todo en el de inversión de dependencias.

Detectar violaciones del SRP:

Saber si estamos respetando o no el principio de responsabilidad única puede ser, en ocasiones, un tanto ambiguo. A continuación veremos un listado de situaciones que nos ayudarán a detectar violaciones del SRP:

- **Nombre demasiado genérico.** Escoger un nombre excesivamente genérico suele derivar en un *God Object*, un objeto que hace demasiadas cosas.

⁵⁶<https://repl.it/@SoftwareCrafter/SOLID-SRP2>

- **Los cambios suelen afectar a esta clase.** Cuando un elevado porcentaje de cambios suele afectar a la misma clase, puede ser síntoma de que dicha clase está demasiado acoplada o tiene demasiadas responsabilidades.
- **La clase involucra múltiples capas de la arquitectura.** Si, como vimos en el caso del ejemplo, nuestra clase hace cosas como acceder a la capa de persistencia o notificar al usuario, además de implementar la lógica de negocio, está violando claramente el SRP.
- **Número alto de *imports*.** Aunque esto por sí mismo no implica nada, podría ser un síntoma de violación.
- **Cantidad elevada de métodos públicos.** Cuando una clase tiene una API con un número alto de métodos públicos, suele ser síntoma de que tiene demasiadas responsabilidades.
- **Excesivo número de líneas de código.** Si nuestra clase solo tiene una única responsabilidad, su número de líneas no debería, en principio, ser muy elevado.

OCP - Principio Abierto/Cerrado

“Todas las entidades software deberían estar abiertas a extensión, pero cerradas a modificación”. – Bertrand Meyer⁵⁷

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

⁵⁷https://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 aquí⁵⁸](#).

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.

⁵⁸<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();
```

Descargado en: www.detodoprogramacion.org

Puedes acceder al ejemplo completo [desde aquí](#).⁵⁹

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*:

⁵⁹<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

El tercer principio SOLID debe su nombre a la doctora **Barbara Jane Huberman**, más conocida como [Barbara Liskov⁶⁰](#). Esta reconocida ingeniera de *software* estadounidense, además de ser ganadora de un premio Turing (el Nobel de la informática), fue la primera mujer en Estados Unidos en conseguir un doctorado en Ciencias Computacionales.

Barbara Liskov y [Jeanette Wing⁶¹](#), de manera conjunta, definieron en 1994 dicho principio, el cual viene a decir algo así: siendo U un subtipo de T , cualquier instancia de T debería poder ser sustituida por cualquier instancia de U sin alterar las propiedades del sistema. En otras palabras, si una clase A es extendida por una clase B , debemos de ser capaces de sustituir cualquier instancia de A por cualquier objeto de B sin que el sistema deje de funcionar o se den comportamientos inesperados.

Este principio viene a desmentir la idea preconcebida de que las clases son una forma directa de modelar el mundo, pero nada más lejos de la realidad. A continuación veremos el por qué de esto con el típico ejemplo del rectángulo y del cuadrado.

Aplicando el LSP

Un cuadrado, desde el punto de vista matemático, es exactamente igual que un rectángulo, ya que un cuadrado es un rectángulo con todos los lados iguales. Por lo tanto, *a priori*, podríamos modelar un cuadrado extendiendo una clase rectángulo, tal que así:

⁶⁰https://en.wikipedia.org/wiki/Barbara_Liskov

⁶¹https://en.wikipedia.org/wiki/Jeanette_Wing

Principio de sustitución de Liskov

```
1 class Rectangle {  
2     constructor() {  
3         this.width = 0;  
4         this.height = 0;  
5     }  
6  
7     setWidth(width) {  
8         this.width = width;  
9     }  
10  
11    setHeight(height) {  
12        this.height = height;  
13    }  
14  
15    getArea() {  
16        return this.width * this.height;  
17    }  
18 }  
19  
20  
21 class Square extends Rectangle {  
22     setWidth(width) {  
23         this.width = width;  
24         this.height = width;  
25     }  
26  
27     setHeight(height) {  
28         this.width = height;  
29         this.height = height;  
30     }  
31 }
```

En el caso del cuadrado, el ancho es igual que el alto, es por ello que cada vez que llamamos a *setWidth* o a *setHeight*, establecemos el mismo valor para el ancho que

para el alto. *A priori*, esto podría parecer una solución válida. Vamos a crear un test unitario (profundizaremos en los test unitarios en la sección dedicada al *testing*) para comprobar que el método *getArea()* devuelve el resultado correcto:

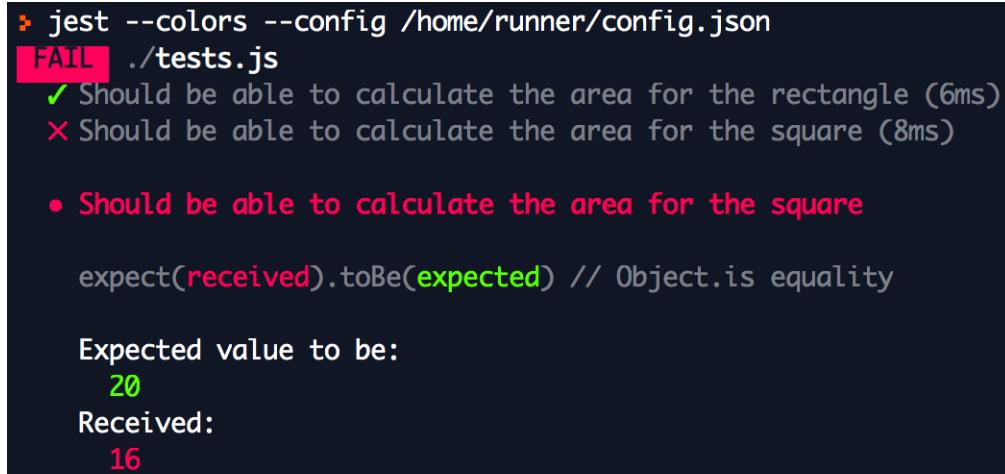
Principio de sustitución de Liskov

```

1 test('Should be able to calculate the area for the rectangle', ()=>{
2   let rectangle = new Rectangle()
3   rectangle.setHeight(5)
4   rectangle.setWidth(4)
5
6   expect(rectangle.getArea()).toBe(20)
7 })

```

Si ejecutamos el test, pasaría correctamente pero, ¿qué sucedería en el caso de reemplazar la clase *Rectangle* por *Square*? Pues directamente el test no pasaría, ya que el resultado esperado sería 16 en lugar de 20. Estaríamos, por tanto, violando el principio de sustitución de Liskov. Puedes probar el código del ejemplo desde aquí⁶²



```

> jest --colors --config /home/runner/config.json
FAIL ./tests.js
  ✓ Should be able to calculate the area for the rectangle (6ms)
  ✗ Should be able to calculate the area for the square (8ms)

  ● Should be able to calculate the area for the square

    expect(received).toBe(expected) // Object.is equality

      Expected value to be:
        20
      Received:
        16

```

Resultado del ejemplo

Como podemos observar, el problema reside en que nos vemos obligados a reimplementar los métodos públicos *setHeight* y *setWidth*. Estos métodos tienen sentido en

⁶²<https://repl.it/@SoftwareCrafter/SOLID-LSP>

la clase *Rectangle*, pero no lo tienen en la clase *Square*. Una posible solución para esto podría ser crear una jerarquía de clase diferentes, extrayendo una clase superior que tenga rasgos comunes y modelando cada clase hija acorde a sus especificaciones:

Principio de sustitución de Liskov

```
1 class Figure{
2     constructor() {
3         this.width = 0;
4         this.height = 0;
5     }
6
7     getArea() {
8         return this.width * this.height;
9     }
10 }
11
12
13 class Rectangle extends Figure {
14     constructor(width, height) {
15         super();
16         this.width = width;
17         this.height = height;
18     }
19 }
20
21
22 class Square extends Rectangle {
23     constructor(length) {
24         super();
25         this.width = length;
26         this.height = length;
27     }
28 }
29
30
31 test('Should be able to calculate the area for the rectangle', ()=>{
```

```

32   let rectangle = new Rectangle(5, 4)
33
34   expect(rectangle.getArea()).toBe(20)
35 })
36
37
38 test('Should be able to calculate the area for the square', ()=>{
39   let square = new Square(5)
40
41   expect(square.getArea()).toBe(25)
42 })

```

Hemos creado una clase *Figure* de la cual heredan las clases *Square* y *Rectangle*. En estas clases hijas ya no se exponen los métodos para establecer el ancho y el alto, con lo cual son perfectamente intercambiables unas por otras, por lo tanto cumple con el principio de LSP. Puedes acceder al ejemplo interactivo desde [aquí⁶³](#).

```

Jest v22.1.2 node v7.4.0 linux/amd64
> jest --colors --config /home/runner/config.json
PASS ./tests.js
  ✓ Should be able to calculate the area for the rectangle (5ms)
  ✓ Should be able to calculate the area for the square (1ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.306s
Ran all test suites.

```

Resultado del ejemplo

De todas formas, si lo piensas bien, este es un caso forzado de herencia, ya que el método de cálculo de área de la clase *Figure* solo funcionaría con cuadrados y rectángulos. En este caso, una mejor solución pasaría por utilizar las interfaces

⁶³<https://repl.it/@SoftwareCrafter/SOLID-LSP-2>

de TypeScript para definir un contrato y aplicar polimorfismo. Veremos esto en el capítulo sobre el principio de segregación de interfaces.

Detectar violaciones del LSP

Como acabamos de ver, la manera más sencilla para detectar violaciones en el principio de sustitución de Liskov es observando si los métodos sobrescritos en una clase hija tienen el comportamiento esperado. Una forma muy común de violación del LSP suele ser cuando los métodos sobreescritos de una clase hija devuelven un `null` o lanzan una excepción.

ISP - Principio de segregación de la interfaz

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

El principio de segregación de la interfaz fue definido por Robert C. Martin cuando trabajaba en Xerox como consultor. Este principio viene a decir que una clase no debería depender de métodos o propiedades que no necesita. Por lo tanto, cuando definimos el contrato de una interfaz, debemos centrarnos en las clases que la van a usar (las interfaces pertenecen a la clase cliente), no en las implementaciones que ya tenemos desarrolladas.

En lenguajes que no disponen de interfaces, como JavaScript, este principio no tiene demasiado sentido y se suele confiar en el buen hacer del propio desarrollador para que aplique el concepto de *duck typing*⁶⁴ de forma coherente. Dicho concepto viene a decir que los métodos y propiedades de un objeto determinan su validez semántica, en vez de su jerarquía de clases o la implementación de una interfaz específica.

En los ejemplos de este capítulo vamos a utilizar TypeScript⁶⁵ y sus interfaces. Estas, además de ayudarnos a comprender mejor este principio, son una herramienta muy poderosa a la hora de definir contratos en nuestro código.

Aplicando el ISP

Las interfaces, como ya sabemos, son abstracciones que definen el comportamiento de las clases que la implementan. La problemática surge cuando esas interfaces tratan de definir más métodos de los necesarios, ya que las clases que la implementan no necesitarán dichos métodos y nos veremos obligados a crear implementaciones

⁶⁴https://en.wikipedia.org/wiki/Duck_typing#In_Python

⁶⁵<https://softwarecrafters.io/typescript/typescript-javascript-introduccion>

forzosas para los mismos, siendo muy común lanzar una excepción, lo que nos llevará a incumplir el principio de sustitución de Liskov.

Veamos esto con un ejemplo: imagina que necesitamos diseñar un sistema que nos permita controlar de forma básica un automóvil independientemente del modelo, por lo que definimos una interfaz tal que así:

Principio de segregación de la interfaz

```
1 interface Car{  
2     accelerate: () => void;  
3     brake:() => void;  
4     startEngine: () => void;  
5 }
```

A continuación definimos una clase Mustang que implementa dicha interfaz:

Principio de segregación de la interfaz

```
1 class Mustang implements Car{  
2     accelerate(){  
3         console.log("Speeding up...")  
4     }  
5  
6     brake(){  
7         console.log("Stopping...")  
8     }  
9  
10    startEngine(){  
11        console.log("Starting engine... ")  
12    }  
13 }
```

Hasta aquí todo bien. Pero, de repente un día nuestro sistema llega a oídos de Elon Musk y quiere que lo adaptemos a su empresa, Tesla Motors. Como sabréis, Tesla, además del componente eléctrico de sus vehículos, tiene algunos elementos diferenciadores sobre el resto de compañías automovilísticas, como son el *auto pilot*

y el modo Ludicrous Speed. Nosotros como no podía ser menos, adaptamos nuestro sistema para controlar, además de los vehículos actuales, los del amigo Elon Musk.

Para ello añadimos a la interfaz *Car* el nuevo comportamiento asociado al nuevo cliente:

Principio de segregación de la interfaz

```
1 interface Car{  
2     accelerate: () => void;  
3     brake:() => void;  
4     startEngine: () => void;  
5     autoPilot: () => void;  
6     ludicrousSpeed: () => void;  
7 }
```

Implementamos la interfaz modificada en una nueva clase *ModelS*:

Principio de segregación de la interfaz

```
1 class ModelS implements Car{  
2     accelerate(){  
3         console.log("Speeding up...")  
4     }  
5  
6     brake(){  
7         console.log("Stopping...")  
8     }  
9  
10    startEngine(){  
11        console.log("Starting engine... ")  
12    }  
13  
14    ludicrousSpeed(){  
15        console.log("woooooooooow ...")  
16    }  
17  
18    autoPilot(){
```

```
19     console.log("self driving... ")
20 }
21 }
```

Pero, ¿que pasa ahora con la clase *Mustang*? Pues que el compilador de TypeScript nos obliga a implementar los métodos adicionales para cumplir con el contrato que hemos definido en la interfaz *Car*:

Ejemplo x: Principio de segregación de la interfaz

```
1 class Mustang implements Car{
2     accelerate(){
3         console.log("Speeding up...")
4     }
5
6     brake(){
7         console.log("Stopping...")
8     }
9
10    startEngine(){
11        console.log("Starting engine... ")
12    }
13
14    ludicrousSpeed(){
15        throw new Error("UnSupported operation")
16    }
17
18    autoPilot(){
19        throw new Error("UnSupported operation")
20    }
21 }
```

Ahora cumplimos con la interfaz, pero para ello hemos tenido que implementar los métodos *autoPilot()* y *ludicrousSpeed()* de manera forzosa. Al hacer esto estamos **violando** claramente el principio de segregación de interfaces, ya que estamos forzando a la clase *Cliente* a implementar métodos que no puede utilizar.

La solución es sencilla, podemos dividir la interfaz en dos trozos, una para los comportamientos básicos de cualquier vehículo (*Car*) y otra interfaz más específica (*Tesla*) que describa el comportamiento de los modelos de la marca. Puedes acceder al editor interactivo con el ejemplo completo desde [aquí](#)⁶⁶.

Principio de segregación de la interfaz

```
1 interface Car{
2     accelerate: () => void;
3     brake: () => void;
4     startEngine: () => void;
5 }
6
7 interface Tesla{
8     autoPilot: () => void;
9     ludicrousSpeed: () => void;
10 }
```

Por último, debemos refactorizar, por un lado, la clase *Mustang* para que solo implemente *Car* y por otro, la clase *ModelS*, para que implemente tanto la interfaz *Car*, como *Tesla*.

Principio de segregación de la interfaz

```
1 class Mustang implements Car{
2     accelerate(){
3         console.log("Speeding up...")
4     }
5
6     brake(){
7         console.log("Stopping...")
8     }
9
10    startEngine(){
11        console.log("Starting engine... ")
12    }
```

⁶⁶<https://repl.it/@SoftwareCrafter/SOLID-ISP3>

```
13 }
14
15 class ModelS implements Car, Tesla{
16     accelerate(){
17         console.log("Speeding up...")
18     }
19
20     brake(){
21         console.log("Stopping...")
22     }
23
24     startEngine(){
25         console.log("Starting engine... ")
26     }
27
28     ludicrousSpeed(){
29         console.log("woooooooooow . . .")
30     }
31
32     autoPilot(){
33         console.log("self driving... ")
34     }
35 }
```

Es importante ser conscientes de que dividir la interfaz no quiere decir que dividamos su implementación. Cuando se aplica la idea de que una única clase implemente varias interfaces específicas, a las interfaces se les suele denominar *role interface*⁶⁷.

Detectar violaciones del ISP

Como podrás intuir, este principio está estrechamente relacionado con el de responsabilidad única y con el de sustitución de Liskov. Por lo tanto, si las interfaces que diseñemos nos obligan a violar dichos principios, es muy probable que también te

⁶⁷<http://martinfowler.com/bliki/RoleInterface.html>

estés saltando el ISP. Mantener tus interfaces simples y específicas y, sobre todo, tener presente la clase cliente que las va a implementar te ayudará a respetar este principio.

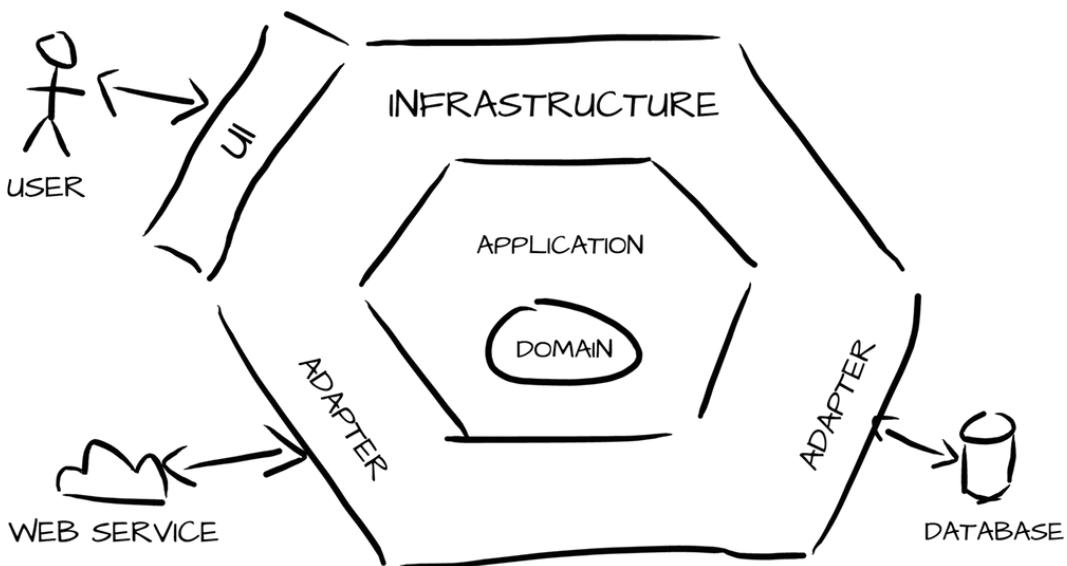
DIP - Principio de inversión de dependencias

“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

Cuando hablamos de abstracciones nos estamos refiriendo a clases abstractas o interfaces. Uno de los motivos más importantes por el cual las reglas de negocio o capa de dominio deben depender de estas y no de concreciones es que aumenta su tolerancia al cambio. Pero, ¿por qué obtenemos este beneficio?

Cada cambio en un componente abstracto implica un cambio en su implementación. Por el contrario, los cambios en implementaciones concretas, la mayoría de las veces, no requieren cambios en las interfaces que implementa. Por lo tanto, las abstracciones tienden a ser más estables que las implementaciones. Con lo cual, si nuestro dominio depende de interfaces, será más tolerante al cambio, siempre y cuando estas se diseñen respetando el principio de sustitución de Liskov y el de segregación de la interfaz.

Pero, ¿cómo escribimos nuestro código para depender de abstracciones y no de concreciones? No seas impaciente, aún debemos introducir un concepto más, la inyección de dependencias.

Inyección de dependencias

En programación nos referimos a **dependencia** cuando módulo o componente requiere de otro para poder realizar su trabajo. Decimos que un componente *A* tiene una dependencia con otro componente *B*, cuando *A* usa *B* para realizar alguna tarea. *Dicha dependencia se manifiesta porque el componente *A no puede funcionar sin el componente B.*

Las dependencias en el *software* son necesarias. La problemática con estas viene dada por el grado de acoplamiento que tiene la dependencia con el componente. Como vimos en el capítulo sobre dependencia y cohesión, debemos tratar de favorecer un grado de acoplamiento bajo, pero sin sacrificar la cohesión. Analicemos el siguiente ejemplo:

Código acoplado con dependencia oculta.

```
1 class UseCase{  
2     constructor(){  
3         this.externalService = new ExternalService();  
4     }  
5  
6     doSomething(){  
7         this.externalService.doExternalTask();  
8     }  
9 }  
10  
11 class ExternalService{  
12     doExternalTask(){  
13         console.log("Doing task...")  
14     }  
15 }
```

En este caso nos encontramos ante una situación de alto acoplamiento, ya que la clase *UseCase* tiene una **dependencia oculta** de la clase *ExternalService*. Si, por algún motivo tuviéramos que cambiar la implementación de la clase *ExternalService*, la funcionalidad de la clase *UseCase* podría verse afectada. ¿Te imaginas la pesadilla que esto supone a nivel de mantenimiento en un proyecto real? Para lidiar con esta problemática debemos empezar por aplicar el **patrón de inyección de dependencias**.

El término fue acuñado por primera vez por [Martin Fowler⁶⁸](#). Se trata de un patrón de diseño que se encarga de extraer la responsabilidad de la creación de instancias de un componente para delegarla en otro. Aunque puede sonar complejo es muy simple, veamos cómo aplicarlo en el ejemplo:

⁶⁸[<https://martinfowler.com/articles/injection.html>](https://martinfowler.com/articles/injection.html)

Código acoplado, con dependencia visible

```
1 class UseCase{
2     constructor(externalService: ExternalService){
3         this.externalService = externalService;
4     }
5
6     doSomething(){
7         this.externalService.doExternalTask();
8     }
9 }
10
11 class ExternalService{
12     doExternalTask(){
13         console.log("Doing task...")
14     }
15 }
```

Ya está, así de sencillo, esto es inyectar la dependencia vía **constructor**, que también se podría hacer vía método **setter**. Ahora, aunque seguimos teniendo un grado de acoplamiento alto, la **dependencia es visible**, con lo cual ya nos queda más clara la relación entre las clases.

**Diagrama UML de dependencias visible**

Como puedes ver, se trata de un concepto muy simple pero que los autores normalmente se complican a la hora de explicarlo.

Aplicando el DIP

En nuestro ejemplo seguimos teniendo un grado de acoplamiento alto, ya que la clase *UseCase* hace uso de una implementación concreta de *ExternalService*. Lo ideal aquí es que la clase cliente (*UseCase*) dependa de una abstracción (interfaz) que defina el contrato que necesita, en este caso *doExternalTask()*, es decir, la clase menos importante *ExternalService*, debe adaptarse a las necesidades de la clase más importante, *UseCase*.

Código desacoplado, con la dependencia invertida

```
1 interface IExternalService{
2     doExternalTask: () => void;
3 }
4
5 class UseCase{
6     externalService: IExternalService;
7
8     constructor(externalService: IExternalService){
9         this.externalService = externalService;
10    }
11
12    doSomething(){
13        this.externalService.doExternalTask();
14    }
15 }
16
17 class ExternalService implements IExternalService {
18     doExternalTask(){
19         console.log("Doing external task...")
20     }
21 }
22
23 const client = new UseCase(new ExternalService());
24
25 client.doSomething();
```

Puedes acceder al ejemplo interactivo desde [aquí⁶⁹](#).

Ahora el código de la clase *UseCase* está totalmente desacoplado de la clase *ExternalService* y tan solo depende de una interfaz creada en base a sus necesidades, con lo cual podemos decir que hemos invertido la dependencia.



Diagrama UML de dependencias invertidas

Detectando violaciones del DIP

Quizás el mejor consejo para detectar si estamos violando el principio de inversión de la dependencia es comprobar que, en la arquitectura de nuestro proyecto, los elementos de alto nivel no tienen dependencias de los elementos de bajo nivel. Desde el punto de vista de la arquitectura hexagonal, esto se refiere a que la capa de dominio no debe saber de la existencia de la capa de aplicación, y, de igual modo, los elementos de la capa de aplicación no deben de conocer nada de la capa de infraestructura. Además, las dependencias con librerías de terceros deben estar en esta última.

⁶⁹<https://repl.it/@SoftwareCrafter/SOLID-DIP>

SECCIÓN III: TESTING Y TDD

Introducción al testing

“El testing de software puede verificar la presencia de errores pero no la ausencia de ellos”. – Edsger Dijkstra⁷⁰

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-

⁷⁰https://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

Aunque en esta sección nos vamos a centrar en los test que escriben los desarrolladores, en concreto en los test unitarios, creo que es interesante realizar, a grandes rasgos, una clasificación de los diferentes tipos que existen. Para ello lo primero es contestar a la pregunta:

¿Qué entendemos por testing?

El *testing* de *software* se define como un conjunto de técnicas que se utilizan para verificar que el sistema desarrollado, ya sea por nosotros o por terceros, cumple con los requerimientos establecidos.

Test manuales vs automáticos

La primera gran diferenciación que podemos realizar es atendiendo a cómo se realiza su ejecución, es decir, a si se hace de forma manual o automática. El *testing* manual consiste en preparar una serie de casos y ejecutar a mano los elementos necesarios, como podrás imaginar tiene muchísimas limitaciones, ya que son lentos, difíciles de replicar, caros y además cubren muy pocos casos. Es por ello que la mayoría de los test deberían de ser automáticos.

Aunque debemos ser conscientes de que no todo es automatizable, ya que a veces se dan casuísticas en las que se vuelve muy costoso, o directamente imposible, automatizar ciertas situaciones y no queda más remedio que hacer ciertas pruebas de forma manual. De ahí la importancia de contar con un buen equipo de QA que complemente al equipo de desarrollo.

Test funcionales vs no funcionales

Una forma quizás más intuitiva de clasificar los [diferentes tipos de test de software que existen⁷¹](#) es agrupándolos en test funcionales y test no funcionales.

Tests funcionales

Los test funcionales hacen referencia a las pruebas que verifican el correcto comportamiento del sistema, subsistema o componente *software*. Es decir, validan que el código cumpla con las especificaciones que llegan desde negocio y que además esté libre de bugs. Dentro de este tipo de pruebas encontramos principalmente las siguientes:

- **Tests Unitarios:** Este tipo de pruebas comprueban elementos básicos de nuestro *software* de forma aislada. Son los test más importantes a la hora de validar las reglas de negocio que hemos desarrollado. Nos centraremos en este tipo de pruebas a lo largo de la sección de *testing*.
- **Tests de integración:** Los test de integración son aquellos que prueban conjuntos de elementos básicos, normalmente suelen incluirse en este tipo de pruebas algunos elementos de infraestructura, como base de datos o llamadas a APIs.
- **Tests de sistema:** Este tipo de test, también denominados end-to-end o de extremo a extremo, prueban múltiples elementos de nuestra arquitectura simulando el comportamiento de un actor con nuestro software.
- **Tests de regresión:** Este tipo de pruebas se encargan de verificar la funcionalidad ya entregada, es decir, son pruebas que se usan para detectar que en los cambios introducidos en el sistema no se genera un comportamiento inesperado. En definitiva, cualquier tipo de test funcional de los que hemos visto podría ser un test de regresión, siempre y cuando hayan pasado correctamente en algún momento y, tras realizar algún cambio en el sistema, empiecen a fallar.

Además de este tipo de test funcionales puedes encontrar algunos más con nomenclatura diferente como podrían ser: *sanity testing*, *smoke testing*, *UI testing*, *Beta/Acceptance testing*, etc. Todos ellos pertenecen a uno o a varios de los tipos de test funcionales anteriores.

⁷¹<https://www.softwaretestinghelp.com/types-of-software-testing/>

Tests no funcionales

El objetivo de los test no funcionales es la verificación de un requisito que especifica criterios que pueden usarse para juzgar la operación de un sistema, como por ejemplo la disponibilidad, accesibilidad, usabilidad, mantenibilidad, seguridad y/o rendimiento. Es decir, a diferencia de los funcionales, se centran en comprobar cómo responde el sistema, no en qué hace o debería hacer.

Podemos clasificar los test no funcionales según el tipo de requisito no funcional que abarcan, entre todos ellos destacan:

- **Tests de carga:** Son test mediante los que se observa el comportamiento de una sistema *software* bajo diferentes números de peticiones durante un tiempo determinado.
- **Tests de velocidad:** Comprueban si el sistema genera los resultados en un tiempo aceptable.
- **Tests de usabilidad:** Son pruebas en las que se trata de evaluar la UX del sistema.
- **Tests de seguridad:** Se trata de un conjunto de pruebas en las que se trata de evaluar si el sistema desarrollado está expuesto a vulnerabilidades conocidas.

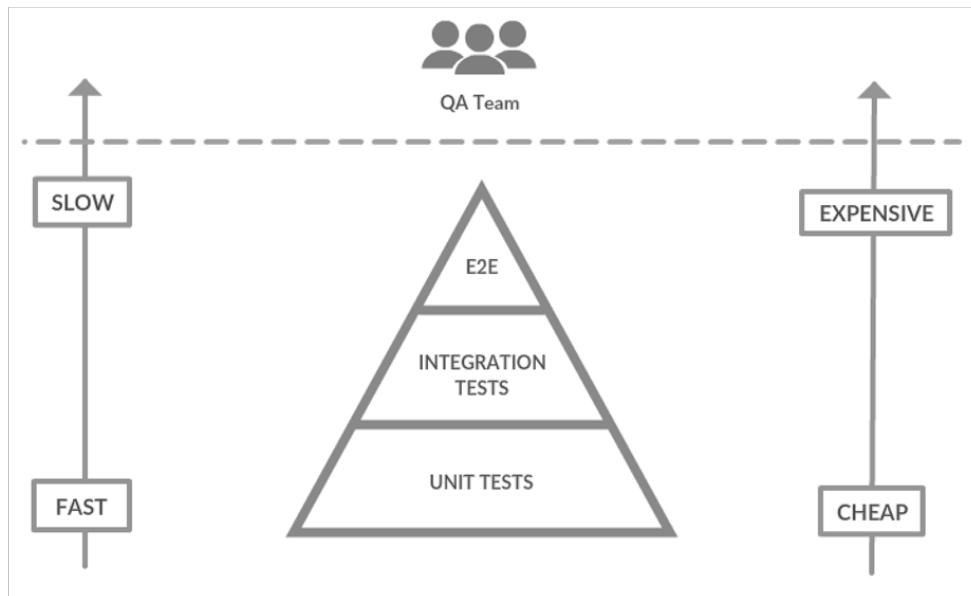
Tanto los test no funcionales como los funcionales siguen el mismo proceso a la hora de generarlos:

- **Escenario inicial:** Se crean una serie de datos de entrada para poder ejecutar las pruebas.
- **Ejecución del tests:** Se ejecuta la prueba con sus correspondientes datos de entrada sobre el sistema.
- **Evaluación del resultado:** El resultado obtenido se analiza para ver si coincide con lo esperado.

Normalmente los test no funcionales son generados por el equipo de QA, mientras que los test funcionales suelen ser creados por los desarrolladores, sobre todo los test unitarios, de integración y la mayoría de los test de sistema.

Pirámide de testing

La Pirámide de Testing, también conocida como Pirámide de Cohn (por su autor [Mike Cohn⁷²](#)), es una forma muy extendida y aceptada de organizar los test funcionales en distintos niveles, siguiendo una estructura con forma de pirámide:



Pirámide de Testing.

La pirámide es muy simple de entender, la idea es tratar de organizar la cantidad de test que tenemos en base a su velocidad de ejecución y al coste de crearlos y mantenerlos. Es por ello que los test unitarios aparecen en la base de la pirámide, ya que, si los diseñamos centrándonos en una sola unidad de *software* de forma aislada, son muy rápidos de ejecutar, fáciles de escribir y baratos de mantener.

En el otro extremo se encuentran los test *end-to-end* o de sistema. Como hemos mencionado. En estos test se prueba nuestro sistema de punta a punta, debido a esto entran en juego todos los elementos del sistema implicados en una acción concreta, con lo cual estos test se antojan lentos a la hora de ejecutar y complicados de crear

⁷²https://en.wikipedia.org/wiki/Mike_Cohn

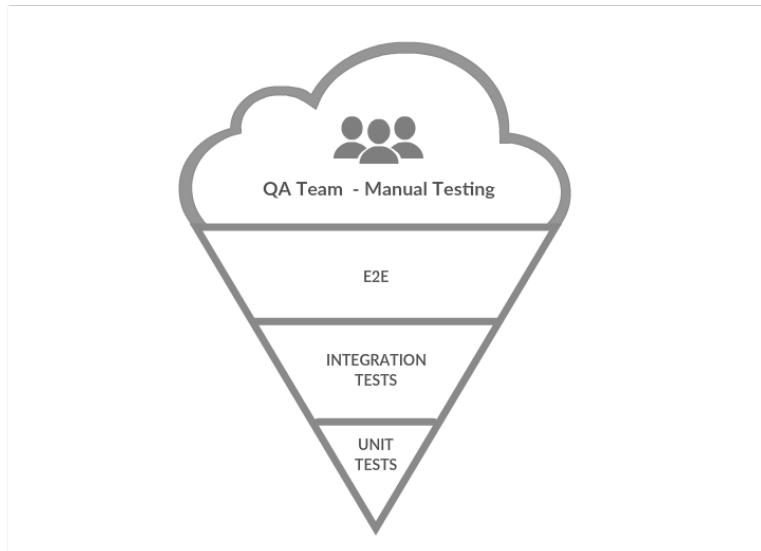
y mantener. Generalmente buscamos tener pocos de estos test debido a su fragilidad y alto costo de mantenimiento.

La parte media de la pirámide está constituida por los test de integración, el objetivo de este tipo de pruebas es comprobar si las diferentes unidades de software interactúan con ciertos elementos de infraestructura como APIs externas o base de datos de la forma prevista. Este tipo de test son más lentos de ejecutar y complejos de escribir y mantener que los test unitarios, aunque en muchos contextos también aportan mayor seguridad. Por otro lado, este tipo de test son bastante más baratos y rápidos que los de sistemas, por esta razón lo ideal es que tengamos una cantidad intermedia de estos.

Antipatrón del cono de helado

La Pirámide del Testing hay que entenderla como una recomendación y no tiene por qué encajar siempre con nuestro contexto, sobre todo en lo referente a la cantidad de test unitarios y de integración. Por ejemplo, podría darse el caso que el sistema software que estamos diseñando tenga pocas reglas de negocio, las cuales pueden ser cubiertas por unos cuantos test unitarios. En cambio, dicho sistema podría ser muy demandante a nivel de elementos externos, en ese caso es probable que nos interese tener más test de integración que unitarios.

Normalmente, contextos como el anterior suelen ser la excepción y, aun hoy en día, en muchos proyectos nos encontramos con el temido “antipatrón del cono de helado”:



Pirámide de Testing invertida.

Como puedes observar, es la pirámide de *testing* invertida. En este contexto se centra el foco en muchas pruebas manuales y *end-to-end*. Esto acarrea múltiples problemas, el principal es que el coste de probar el sistema se dispara, ya que con pocos test de integración y unitarios (y muchas veces ninguno) se vuelve tremadamente complejo determinar dónde están los problemas cuando los test de nivel superior fallan.

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 **Smalltalk⁷³**, 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.

⁷³<https://es.wikipedia.org/wiki/Smalltalk>

Características de los tests unitarios

Hace unos años, la mayoría de los desarrolladores no hacían *unit testing*, algo que sigue ocurriendo hoy en día, aunque en menor medida. Si es tu caso, no te preocupes, ya que en esta sección te pondrás al día. En el otro extremo tenemos a los dogmáticos del *unit testing*, los cuales tienen como objetivo el 100% de *code coverage*. Esto es un error, ya que la métrica de la cobertura de pruebas solo indica qué líneas de código (y cuáles no) se han ejecutado al menos una vez al pasar las pruebas unitarias. Esto suele derivar en test de poca calidad que no aportan demasiada seguridad, en este contexto suele primar la cantidad sobre la calidad. Por esta razón, el *code coverage* nunca debe de ser un objetivo en sí mismo.

Como casi siempre, en el equilibrio está la virtud. Por ello, es fundamental tener en cuenta la frase de Kent Beck con la que abrimos el capítulo: “Nos pagan por escribir código que funciona, no por hacer test”. Por esta razón debemos centrarnos en escribir test de calidad, que además de validar elementos útiles, sean:

- **Rápidos:** Los test tienen que ejecutarse lo más rápidamente posible, para no perder tiempo y poder ejecutarlos con la frecuencia adecuada.
- **Aislados:** Es muy importante que el estado de cada test sea totalmente independiente y no afecte a los demás.
- **Atómicos:** cada test debe probar una cosa y ser lo suficientemente pequeño.
- **Fáciles de mantener y extender:** Como dice Edsger Dijkstra, la simplicidad es un prerequisito de la fiabilidad. Mantener simples los test es fundamental para mantenerlos y extenderlos.
- **Deterministas:** Los resultados deben ser consistentes, es decir, deben producir siempre la misma salida a partir de las mismas condiciones de partida.
- **Legibles:** Una prueba legible es aquella que revela su propósito o intención de forma clara. Para ello la elección de un nombre autoexplicativo es fundamental.

Anatomía de un test unitario

Como norma general, los test deben tener una estructura muy simple basada en las siguientes tres partes:

- **Preparación (Arrange):** en esta parte del test preparamos el contexto para poder realizar la prueba. Por ejemplo, si probamos un método de una clase, primero tendremos que instanciar dicha clase para probarlo. Además, una parte la preparación puede estar contenida en el método *SetUp* (*Before* en el caso de Jest), si es común a todos los test de la clase.
- **Actuación (Act):** ejecutamos la acción que queremos probar. Por ejemplo, invocar un método con unos parámetros.
- **Aserción (Assert):** verificamos si el resultado de la acción es el esperado. Por ejemplo, el resultado de la invocación del método anterior tiene que devolver un valor determinado.

En inglés estas tres partes se identifican con las siglas AAA lo que ayuda a recordarlas, **Arrange**, **Act**, **Assert**.

Veamos un ejemplo:

Anatomía de un test unitario.

```
1 test('return zero if receive one', () => {
2   //Arrange
3   const n = 1;
4
5   //Act
6   const result = fibonacci(n);
7
8   //Assert
9   expect(result).toBe(0);
10});
```

En el ejemplo hemos definido un test tal y como lo haríamos con el *framework* Jest (lo veremos en el siguiente capítulo). Para ello hemos usado la función `test`, la cual recibe dos parámetros. El primero es un *string* con la descripción del test y el segundo un *callback*.

El *callback* contiene la lógica del propio test en sí. En la primera parte del *callback* tenemos el *arrange* o preparación. En este caso, inicializamos una constante con el parámetro que le vamos a pasar a la función que queremos probar. A continuación,

tenemos la actuación donde ejecutamos la función *fibonacci* para el valor de n, y lo almacenamos en la variable *result*. Por último, tenemos la parte de la aserción donde verificamos, a través del método *expect* de Jest y el *matcher toBe*, si el valor del resultado es el que esperamos, en este caso cero. Profundizaremos en las diferentes aserciones que nos provee Jest en el siguiente capítulo.

No siempre vamos a tener las tres partes del test claramente diferenciadas, muchas veces, en tests tan sencillos como este nos lo podemos encontrar todo en una sola linea:

Ejemplo x: Anatomía de un test unitario.

```
1 test('return zero if receive one', () => {  
2     expect(fibonacci(1)).toBe(0);  
3 });
```

Personalmente, aunque sean test simples, te recomiendo que trates de respetar la estructura de la triple A, ya que te ayudará a mantener una buena legibilidad de tus test.

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 [Mocha⁷⁴](#), [Jasmine⁷⁵](#) y [Jest⁷⁶](#), 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 [React⁷⁷](#), 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

Entre sus principales características destacan:

- Fácil instalación
- Retroalimentación inmediata con modo ‘watch’
- Plataforma de *testing* de configuración simple.
- Ejecución rápida y con entorno aislado.

⁷⁴<https://mochajs.org/>

⁷⁵<https://jasmine.github.io/>

⁷⁶<https://facebook.github.io/jest/>

⁷⁷<https://softwarecrafters.io/react/tutorial-react-js-introduccion>

- Herramienta de *code coverage* integrada.
- Introduce el concepto de *Snapshot testing*.
- Potente librería de *mocking*.
- Funciona con TypeScript además de ES6

Instalación y configuración

Como en los demás ejemplos del libro, vamos a seguir usando [Repl.it⁷⁸](#) para mostrar los ejemplos interactivos. No obstante, creo que es importante que tengamos claro cómo instalar y configurar Jest, por si queremos poner en marcha los ejemplos en nuestro entorno local.

Al igual que sucede con cualquier otro proyecto JavaScript, podemos instalar Jest mediante [Yarn⁷⁹](#) o [NPM⁸⁰](#). Para nuestros ejemplos usaremos NPM. Lo primero que debemos hacer es crear un directorio para el proyecto y, a continuación, ejecutar *npm init* para inicializar el proyecto:

Comandos de configuración

```
1 mkdir testing_1
2 npm init
```

Una vez inicializado el proyecto, instalamos las dependencias. En este caso, además de las dependencias de Jest, vamos a instalar ts-jest y TypeScript, ya que simplifica el uso de Jest con ES6 y superior. De esta manera, el proyecto queda funcionando también para los que prefieran TypeScript.

Instalación de dependencias

```
1 npm install --save-dev jest typescript ts-jest @types/jest
```

Tras instalar las dependencias debemos ejecutar el comando de configuración de *ts-jest*:

⁷⁸<http://repl.it/>

⁷⁹<https://yarnpkg.com/lang/en/>

⁸⁰<https://www.npmjs.com/>

Comando de inicialización de ts-jest

```
1 npx ts-jest config:init
```

Con todo el *tooling* preparado, si queremos ejecutar los test desde npm, tan solo debemos añadir a la sección de *scripts* del fichero *package.json* lo siguiente:

Configuración de npm para ejecutar jest

```
1 "scripts": {  
2   "test": "jest",  
3   "test:watch": "jest --watchAll"  
4 }
```

El primer script (npm test) ejecutará la suite de test de forma normal, es decir, ejecutará los test y Jest se cerrará. El segundo, npm run test:watch, ejecutará los test pero se mantendrá en modo “watcher” y cada vez que hagamos un cambio en el código Jest volverá a ejecutar los test de forma automática.

Por supuesto, si tratamos de ejecutar algunos de estos *scripts*, la ejecución fallará, ya que aún no hemos creado ningún test en el proyecto.

```
PROBLEMS TERMINAL ... 2: node + □ └─ ×  
  
No tests found, exiting with code 1  
Run with `--passWithNoTests` to exit with code 0  
In /Users/miguelghz/Dropbox/Blog/LeanPub/cleancodejavascript/examples/testing/testing_1  
  5 files checked.  
  testMatch: **/_tests_/**/*.[jt]s?(x), **/?(*.)+(spec|test).[tj]s?(x) - 0 matches  
  testPathIgnorePatterns: /node_modules/ - 5 matches  
  testRegex: - 0 matches  
  Pattern: - 0 matches  
  
Watch Usage: Press w to show more.
```

Ejecución fallida de Jest

Puedes descargar la configuración lista para ser usada desde nuestro repositorio⁸¹ de GitHub.

⁸¹<https://github.com/softwarecrafters-io/typescript-jest-minimal>

Nuestro primer test

Antes de crear nuestro primer tests, vamos a crear la siguiente estructura de carpetas:

Ejemplo x: Framework Jest

```
1 src
2   |--core
3   |--tests
```

Dentro de *core*, crearemos el código con nuestras “reglas de negocio” y, en el directorio *tests*, los test asociados.

A continuación, vamos a crear un ejemplo sencillo que nos permita comprobar que hemos realizado correctamente la configuración. Para ello, creamos el fichero *sum.ts* dentro del directorio *core* con el siguiente código:

Ejemplo x: Framework Jest

```
1 export const sum = (a, b) =>
2   a + b;
```

Una vez creado el código a probar, vamos a crear el test asociado. Para ello añadimos el fichero *sum.test.ts* dentro del directorio de test con lo siguiente:

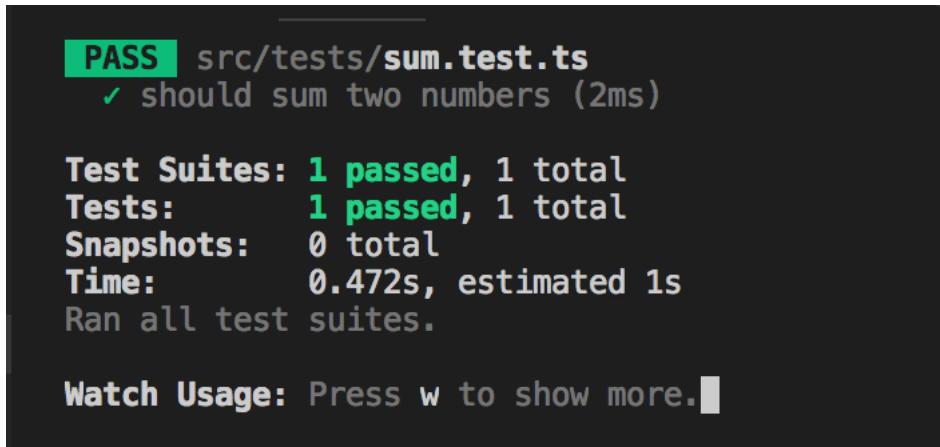
Ejemplo x: Framework Jest

```
1 import {sum} from '../core/sum'
2
3 test('should sum two numbers', () => {
4   //Arrange
5   const a = 1;
6   const b = 2;
7   const expected = 3;
8
9   //Act
10  const result = sum(a,b)
```

```
12  //Assert  
13  expect(result).toBe(expected);  
14});
```

En el test, simplemente ejecutamos nuestra función *sum* con unos valores de entrada y esperamos que nos devuelva un resultado de salida. Sencillo, ¿verdad?

Vamos a ejecutar los test de nuevo para ir familiarizándonos con la salida:



```
PASS  src/tests/sum.test.ts
  ✓ should sum two numbers (2ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.472s, estimated 1s
Ran all test suites.

Watch Usage: Press w to show more. █
```

Ejecución fallida de Jest

¡Genial! ¡Ya tenemos nuestro primer test pasando!

Aserciones

Las aserciones, *asserts* o *matchers* en inglés, son funciones que nos proveen los *frameworks* de *testing*, Jest en nuestro caso, para verificar si el valor esperado por la prueba automática coincide realmente con el obtenido. Evidentemente, en el caso de que el valor coincida, el test pasará, en caso contrario fallará.

En el test que vimos en el ejemplo utilizamos la aserción *toBe* la cual verifica que dos valores sean iguales, para ello se basa en [Object.is\(\)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is)⁸². Es importante que tengas esto en cuenta porque, en el caso de comparar objetos o colecciones, obtendrás resultados inesperados.

⁸²https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is

Ejemplo x: Framework Jest

```
1 expect(1 + 2).toBe(3)
```

En el caso de querer comprobar que son distintos, podemos utilizar el *matcher toBe* precedido de *not*. Esta aproximación es interesante ya que añade semántica a nuestros test.

Ejemplo x: Framework Jest

```
1 expect(1 + 2).not.toBe(4)
```

Para verificar objetos, debes utilizar la aserción *toEqual*, ya que realiza una comparación profunda que verifica correctamente cada campo de la estructura.

Ejemplo x: Framework Jest

```
1 test('equality of objects', () => {
2   const data = { one: 1 };
3   data['two'] = 2;
4   const expected = { one: 1, two: 2 }
5
6   expect(data).toEqual(expected);
7 });
```

Además de los *asserts* de igualdad que hemos visto, Jest nos ofrece otros interesantes como:

- **toBeNull**: valida que el valor sea *null*.
- **toBeGreaterThan**: valida que un valor numérico sea mayor a un número especificado. Además de este tenemos otros muchos comparadores para valores numéricos que pueden aportar semántica a nuestros test.
- **toMatch**: valida un *string* a través de una expresión regular.
- **toContain**: verifica que un *array* contenga un valor específico.
- **toThrow**: verifica que se haya lanzado una excepción.

La referencia a la lista completa de las aserciones la puedes encontrar en [este enlace⁸³](https://jestjs.io/docs/en/expect).

⁸³<https://jestjs.io/docs/en/expect>

Organización y estructura

Estructurar bien nuestros test es un aspecto básico que condiciona su legibilidad y mantenibilidad. Como norma general, los test se suelen agrupar en base a un contexto común, como podría ser, por ejemplo, un determinado caso de uso. A dichas agrupaciones se las conoce con el nombre de *suites*.

Jest nos permite definir estas suites tanto a nivel de fichero como a nivel de contexto. A nivel de fichero debemos tener en cuenta que el nombre ha de respetar uno de estos dos formatos: *.spec.* o *.test.* Esto es necesario para que Jest los detecte sin tener que modificar la configuración. Además, para mantener una buena legibilidad, lo ideal sería que el sufijo que lo precede sea el nombre del fichero que contiene el código que estamos probando.

Por otro lado, dentro de los propios ficheros podemos agrupar los test en contextos o *describes*. Además de permitirnos tener varios contextos, Jest nos permite anidarlos. Aunque como norma general, deberíamos evitar la anidación en más de dos niveles, ya que dificulta la legibilidad de los test.

Definición de contextos

```
1 describe('Use Case', () => {
2   test('Should able to do something...', () => {});
3 });
```

Gestión del estado: *before* y *after*

En muchas ocasiones, cuando tenemos varios test para un mismo componente, nos encontramos en un contexto en el que debemos de inicializar y/o finalizar el estado dicho componente. Para ello, los frameworks basados en *RSpec*⁸⁴ nos proveen de métodos de *setup* y *tearDown*, los cuales se ejecutan antes y después, respectivamente. A su vez, pueden ser ejecutados antes o después de cada uno de los test, o al principio y al final de la suite entera. En el caso de Jest estos métodos son:

⁸⁴<https://en.wikipedia.org/wiki/RSpec>

Ejemplo x: Framework Jest

```
1 describe('Use Case', () => {
2     beforeEach(() => {
3         //se ejecuta antes de cada test
4     });
5
6     afterEach(() => {
7         //se ejecuta después de cada test
8     });
9
10    beforeEach(() => {
11        //se ejecuta antes de todos los tests
12    });
13
14    afterEach(() => {
15        //se ejecuta después de todos los tests
16    });
17
18    test('Should able to do something...', () => {
19
20    });
21});
```

Code coverage

Como mencionamos en el capítulo anterior, el *code coverage*, o cobertura de test en castellano, no es más que una métrica que indica cuál es el porcentaje de nuestro código que ha sido ejecutado por los test unitarios. A pesar de que puede ser una métrica peligrosa, sobre todo si es usada como indicador de calidad (ya que podría derivar en lo contrario), puede ser interesante a nivel orientativo.

Obtener esta información con Jest es muy sencillo, ya que tan solo debemos añadir el *flag -coverage*. Si actualizamos la sección de *scripts* de nuestro *package.json* quedaría tal que así:

```
1 "scripts": {  
2     "test": "jest",  
3     "test:watch": "jest --watchAll",  
4     "test:coverage": "jest --coverage"  
5 }
```

Si lo ejecutamos a través del comando `npm run test:coverage`, Jest nos mostrará una tabla a modo de resumen con un aspecto similar al siguiente:

The terminal window shows the following output:

```
PASS  src/tests/sum.test.ts  
  ✓ should sum two numbers (4ms)  
  
-----|-----|-----|-----|-----|-----|  
File   | % Stmt | % Branch | % Funcs | % Lines | Uncovered Line #s |  
-----|-----|-----|-----|-----|-----|  
All files | 100  | 100    | 100    | 100    | 100    |  
sum.ts   | 100  | 100    | 100    | 100    | 100    |  
-----|-----|-----|-----|-----|-----|  
Test Suites: 1 passed, 1 total  
Tests:       1 passed, 1 total  
Snapshots:   0 total  
Time:        1.297s
```

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 los 90 y forma parte de la metodología *extreme programming*⁸⁵. 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 acostumbres a aplicarlo, promete una mayor velocidad a la hora de desarrollar.

Las tres leyes del TDD

Robert C. Martin describe la esencia del TDD como un proceso que atiende a las siguientes tres reglas:

- No escribirás código de producción sin antes escribir un test que falle.
- No escribirás más de un test unitario suficiente para fallar (y no compilar es fallar).
- No escribirás más código del necesario para hacer pasar el test.

Estas tres leyes derivan en la repetición de lo que se conoce como el ciclo *Red-Green-Refactor*. Veamos en qué consiste:

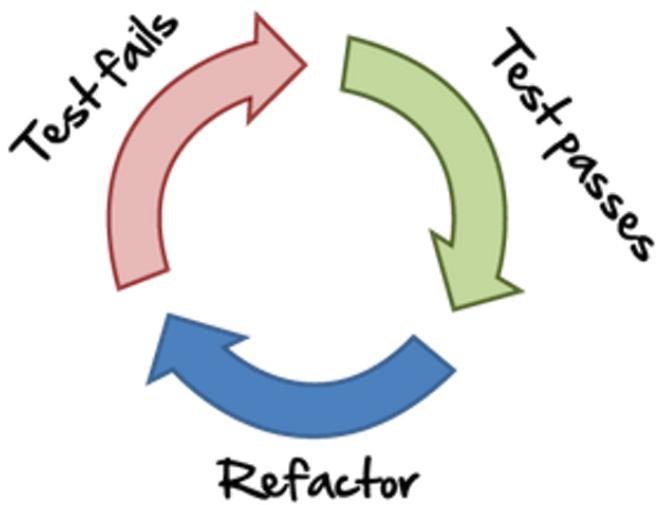
El ciclo Red-Green-Refactor

El ciclo *Red-Green-Refactor*, también conocido como algoritmo del TDD, se basa en:

⁸⁵https://en.wikipedia.org/wiki/Extreme_programming

- **Red:** Escribir un test que falle, es decir, tenemos que realizar el test antes de escribir la implementación. Normalmente se suelen utilizar test unitarios, aunque en algunos contextos puede tener sentido hacer TDD con test de integración.
- **Green:** Una vez creado el test que falla, implementaremos el mínimo código necesario para que el test pase.
- **Refactor:** Por último, tras conseguir que nuestro código pase el test, debemos examinarlo para ver si hay alguna mejora que podamos realizar.
- Una vez que hemos cerrado el ciclo, empezamos de nuevo con el siguiente requisito.

Descargado en: www.detodoprogramacion.org



Ciclo Red-Green-Refactor

Esta forma de programar ofrece dos beneficios principales. El primero y más obvio es que obtenemos un código con una buena cobertura de test, lo que es positivo hasta cierto punto. Recuerda, nos pagan por escribir código que funciona, no por hacer test.

El segundo beneficio es que escribir primero las pruebas nos ayuda a diseñar la API que va a tener nuestro componente, ya que nos obliga a pensar en cómo queremos

utilizarlo. Esto suele acabar derivando en componentes con responsabilidades bien definidas y bajo acoplamiento.

TDD como herramienta de diseño

Cuando Kent Beck desarrolló esta metodología lo hizo centrándose en el segundo de los beneficios que describimos en el apartado anterior, es decir, en TDD como una herramienta de diseño de software que nos ayuda a obtener mejor código, no a obtener más test. Para ello, una vez que tenemos una lista con los primeros requisitos que debe satisfacer el producto, debemos seguir los siguientes pasos:

1. Escogemos un requisito.
2. Escribimos un test que falla.
3. Creamos la implementación mínima para que el test pase.
4. Ejecutamos todos los tests.
5. Refactorizamos.
6. Actualizamos la lista de requisitos.

En el último paso, cuando actualizamos la lista de requisitos, además de marcar como completado el requisito implementado, debemos añadir los nuevos requisitos que hayan podido aparecer.

Normalmente, cuando desarrollamos un producto *software*, los requisitos no están completamente definidos desde el principio, o estos sufren cambios a corto y medio plazo, bien porque son descartados, modificados o porque surgen otros nuevos. TDD encaja muy bien con este tipo de escenarios ya que, además de ir añadiendo test que evalúan que nuestro diseño cumple con los requisitos especificados, ayuda a descubrir nuevos casos que no se habían detectado previamente. A esto último se le conoce como **diseño emergente**.

Esta es la razón por la que para muchos de sus seguidores la última “D” de TDD debería significar *design* en vez de *development*.

Estrategias de implementación, de rojo a verde.

Quizás uno de los puntos más delicados a la hora de aplicar TDD como herramienta de diseño es en el paso en el que ya tenemos un test que falla y debemos crear la implementación mínima para que el test pase. Para ello Kent Beck, en su libro *Test Driven Development by Example*⁸⁶, expone un conjunto de estrategias, también conocidas como patrones de barra verde, que nos van a permitir avanzar en pasos pequeños hacia la solución del problema.

Implementación falsa

Una vez que tenemos el test fallando, la forma más rápida de obtener la primera implementación es creando un fake que devuelva una constante. Esto nos ayudará a ir progresando poco a poco en la resolución del problema, ya que al tener la prueba pasando estamos listos para afrontar el siguiente caso.

La mejor forma de entender el concepto es con un ejercicio práctico. El ejercicio es simple, vamos a construir una función que reciba como parámetro un número entero n y devuelva el n -ésimo número de Fibonacci. Recuerda la sucesión de Fibonacci comienza con 0 y 1, los siguientes términos siempre son la suma de los dos anteriores:

n	0	1	2	3	4	5	6	7	8	9	10	11	12
F_n	0	1	1	2	3	5	8	13	21	34	55	89	144

Sucesión de Fibonacci.

Observando la tabla anterior, podemos darnos cuenta de que los casos *edge* son 0 y 1, además de los más sencillos de implementar. Vamos a empezar por crear el test para $n = 0$:

⁸⁶<https://amzn.to/2mkC2Vt>

Fibonacci, primer test.

```
1 describe('Fibonacci should', () => {
2   it('return zero if receive zero', () => {
3     expect(fibonacci(0)).toBe(0);
4   });
5 });
```

La implementación *fake* más obvia que permite que el test pase es hacer que la función fibonacci devuelva 0 como una constante:

Fibonacci, primera implementación.

```
1 function fibonacci(n) {
2   return 0;
3 }
```

Puedes acceder al ejemplo interactivo [desde aquí](#).⁸⁷

Una vez que tenemos el primer test pasando, la idea es transformar gradualmente la constante en una expresión. Veámoslo en el ejemplo, para ello primero debemos crear un test para el siguiente caso obvio, n = 1;

Fibonacci, segundo test.

```
1 it('return one if receive one', () => {
2   expect(fibonacci(1)).toBe(1);
3 });
```

Ya tenemos el siguiente test fallando. El siguiente paso obvio es escribir una pequeña expresión con un condicional para una entrada con n = 0 devuelva 0 y para n = 1 devuelva 1:

⁸⁷<https://repl.it/@SoftwareCrafter/TDD-Fibonacci>

Fibonacci, segunda implementación.

```
1 function fibonacci(n) {  
2     if(n ==0)  
3         return 0;  
4     else  
5         return 1;  
6 }
```

Puedes acceder al ejemplo interactivo [desde aquí.⁸⁸](#)

Como puedes observar, la técnica de la implementación falsa nos ayuda a progresar poco a poco. Principalmente tienes dos ventajas inherentes, la primera es a nivel psicológico, ya que se hace más llevadero tener algunos test en verde, en vez de en rojo, que nos permitan ir dando pasos pequeños hacia la solución. La segunda tiene que ver con el control del alcance, ya que esta práctica nos permite mantener el foco en el problema real, evitando caer en optimizaciones prematuras.

Triangular

Triangular, o la técnica de la triangulación, es el paso natural que sigue a la técnica de la implementación falsa. Es más, en la mayoría de los contextos, forma parte de la triangulación, basándose en lo siguiente:

1. Escoger el caso más simple que debe resolver el algoritmo.
2. Aplicar *Red-Green-Refactor*.
3. Repetir los pasos anteriores cubriendo las diferentes casuísticas.

Para comprender cómo funciona la triangulación, vamos a continuar desarrollando el ejemplo de Fibonacci, el cual, en parte, ya hemos empezado a triangular. El siguiente caso que podríamos cubrir es para $n = 2$.

⁸⁸<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-1>

Fibonacci, tercer test.

```
1 it('return one if receive two', () => {
2   expect(fibonacci(2)).toBe(1);
3 });
```

Puedes acceder al ejemplo interactivo [desde aquí](#).⁸⁹

En esta ocasión el test pasa, por lo tanto, nuestro algoritmo también funciona para $n = 2$. El siguiente paso sería comprobar qué ocurre para $n = 3$.

Fibonacci, test para $n = 3$.

```
1 it('returns two if receive three', () => {
2   expect(fibonacci(3)).toBe(2);
3 });
```

Como suponíamos, el test falla. Este paso nos ayudará a aproximarnos a la implementación de una solución más genérica. Ya que podríamos crear una implementación falsa para $n = 3$ y añadir otro condicional que devuelva 1 para $n = 1$ y $n = 2$.

Fibonacci, implementación para $n = 3$

```
1 function fibonacci(n) {
2   if(n == 0)
3     return 0;
4   if(n == 1 || n == 2)
5     return 1;
6   else
7     return 2;
8 }
```

Puedes ver el ejemplo interactivo [desde aquí](#)⁹⁰.

Ahora que tenemos los test pasando, vamos a comprobar qué sucede para $n = 4$:

⁸⁹<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-2>

⁹⁰<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-3>

Fibonacci, test para n = 4.

```
1 it('returns three if receive four', () => {
2   expect(fibonacci(4)).toBe(3);
3 });
```

Al llegar a este punto, ya te habrás dado cuenta de que sería más fácil escribir la implementación obvia que seguir haciendo ramas de decisión:

Fibonacci, implementación para n = 4

```
1 function fibonacci(n) {
2   if(n == 0)
3     return 0;
4
5   if(n == 1 || n == 2)
6     return 1;
7
8   else
9     return fibonacci(n - 1) + fibonacci(n - 2);
10 }
```

En este paso, nuestro algoritmo funciona para cualquier valor de n, aunque aún podemos refactorizarlo para eliminar duplicidades y darle un aspecto más funcional:

Fibonacci, refactor funcional

```
1 function fibonacci(n) {
2   const partialFibonacci = (n) =>
3     n == 1
4       ? 1
5       : fibonacci(n - 1) + fibonacci(n - 2)
6
7   return n == 0
8     ? 0
9     : partialFibonacci(n)
10 }
```

Puedes acceder al ejemplo interactivo [desde aquí⁹¹](#).

Con este último paso hemos resuelto el algoritmo de Fibonacci aplicando un enfoque funcional y utilizando la triangulación. Quizás en un hipotético siguiente paso deberíamos eliminar los test para $n=3$, $n=4$ y $n=5$, ya que en este punto no aportan demasiado valor, y crear un test que compruebe el algoritmo generando un número aleatorio mayor que 2 cada vez que se ejecuta.

Como puedes observar, la triangulación es una técnica muy conservadora para aplicar TDD, su uso tiene sentido cuando no tenemos clara la implementación obvia de la solución.

Implementación obvia

Cuando la solución parece muy sencilla, lo ideal es escribir la implementación obvia en las primeras iteraciones del ciclo *Red-Green-Refactor*.

La problemática con esto surge cuando nos precipitamos, creyendo que se trata de un problema sencillo, cuando en realidad no lo es, porque tiene, por poner un ejemplo, algún caso *edge* sobre el que no habíamos reflexionado.

Limitaciones del TDD

Por muchos beneficios inherentes que tenga (o que nos prometan), la técnica del TDD no debe entenderse como una religión ni como una fórmula mágica que vale para todo. Seguir TDD a rajatabla y en todos los contextos no garantiza que tu código vaya a ser más tolerante al cambio, robusto o seguro, ni siquiera te asegura que vayas a ser más productivo a la hora de diseñar *software*.

Desde mi punto de vista, aplicar TDD no encaja bien en todos los contextos. Por ejemplo, si existe una implementación obvia para un caso de uso, directamente la escribo y luego hago las pruebas. En el caso de estar trabajando en el *frontend* tampoco me planteo hacer TDD para diseñar componentes de la *UI*. Incluso es discutible si se deberían hacer test unitarios para probar elementos de la *UI*, desarrolladores de la talla de Ward Cunningham han comentado repetidas veces que no conviene

⁹¹<https://repl.it/@SoftwareCrafter/TDD-Fibonacci-4>

hacer test automatizados sobre esta, ya que es muy cambiante y los test quedan desactualizados con demasiada frecuencia.

Mi consejo es que pruebes, trata de aplicarlo en tu día a día durante una temporada y luego decidas por ti mismo. En los siguientes capítulos vamos a ver algunas *katas* para que puedas seguir practicando.

TDD Práctico: La kata FizzBuzz

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

Pasar de la teoría a la práctica es fundamental. En el mundo del desarrollo si estudias un concepto y no lo pones en práctica durante unos días con la suficiente **repetición espaciada**⁹² es probable que nunca lo llegues a interiorizar. Por esta razón, es muy recomendable que trates de practicar todo lo que puedas, realizar *katas* de código te ayudará con este objetivo.

Las katas de código

En artes marciales, una *kata* es un conjunto de movimientos establecidos. Normalmente se realizan en solitario, con el fin de perfeccionar las bases del conocimiento de un arte marcial en concreto. Aunque estas tengan un componente estético, su propósito no es el de representarlas en un escenario, sino el de entrenar la mente y el cuerpo para saber cómo reaccionar en una situación de combate determinada.

El concepto de *kata* de código no es más que un ejercicio de programación en el que se plantea un problema (con o sin restricciones), que el desarrollador debe tratar de resolver, idealmente, utilizando diferentes técnicas. El término fue acuñado por primera vez por [Dave Thomas](#),⁹³ coautor del mítico libro [The Pragmatic Programmer](#)⁹⁴, del cual, por cierto, van a sacar una [versión](#)⁹⁵ para conmemorar su vigésimo aniversario.

Dave Thomas decía que los músicos, cuando tratan de mejorar, no están siempre tocando con la banda, sino que suelen realizar ejercicios en solitario que les permiten refinar su técnica. Los desarrolladores, en el trabajo, estamos todo el día programando en proyectos reales, en los cuales, normalmente, solemos resolver el mismo tipo de

⁹²https://es.wikipedia.org/wiki/Repaso_espaciado

⁹³<https://twitter.com/pragdave>

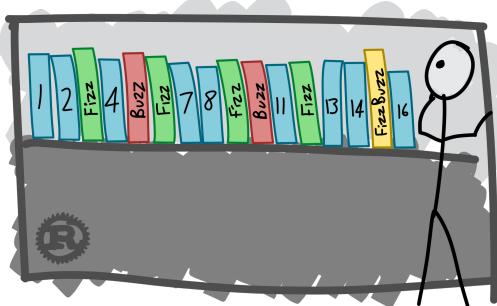
⁹⁴<https://amzn.to/2LG4Joy>

⁹⁵<https://pragprog.com/book/tpp20/the-pragmatic-programmer-20th-anniversary-edition>

problemas. Usando el símil con los músicos, podríamos decir que estamos todo el día tocando con la banda. Después de cierto tiempo, esto limita nuestra capacidad de aprendizaje, por lo que realizar *katas* en solitario o acudir a *coding dojos*, nos ayudará a adquirir nuevas habilidades.

La kata FizzBuzz

La *kata* FizzBuzz, además de ser un gran ejercicio para practicar TDD, es una de las pruebas más típicas a las que te vas a enfrentar en una entrevista de trabajo para un puesto de desarrollador. Este ejercicio tiene su origen en un juego infantil cuyo objetivo era que los niños practicasen la división. Esta *kata* empezó a volverse popular entre los desarrolladores después de que [Emily Bache y Michael Feathers⁹⁶](#) la presentaran en la competición *Programming with the stars* en la “Agile Conference” de 2008.



Kata FizzBuzz

Descripción del problema

El enunciado de la *kata* es el siguiente: Escribe un programa que muestre en pantalla los números del 1 al 100, sustituyendo los múltiplos de 3 por la palabra Fizz y, a su vez, los múltiplos de 5 por Buzz. Para los números que a su vez son múltiplos de 3 y 5, utiliza el combinado FizzBuzz.

Salida de ejemplo:

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16, 17, Fizz, 19, Buzz...

⁹⁶<https://emilybache.blogspot.com/2008/08/onwards-with-stars.html?m=0>

Diseño de la primera prueba

Antes de escribir el primer test, me gusta comprobar que todas las piezas están correctamente configuradas. Para ello escribo un test sencillo con el cual simplemente compruebo que el framework de *testing* funciona correctamente.

FizzBuzz

```
1 describe('FizzBuzz', () => {
2   it('', () => {
3     expect(true).toBe(true);
4   });
5});
```

Puedes acceder al ejemplo [desde aquí⁹⁷](#).

Una vez que estamos seguros de que nuestra *suite* funciona como esperamos, podemos tratar de escoger el primer test. En TDD esto a veces presenta cierta dificultad, por ello es interesante hacer una pequeña lista, como si de una caja negra se tratase, con los diferentes casos que deberían ir cubriendo nuestros test:

- Para el número uno el resultado debe ser uno
- Para el número tres el resultado debe ser “fizz”
- Para el número cinco el resultado debe ser “buzz”
- Para el número quince el resultado debe ser “fizzbuzz”
- Para cualquier número divisible entre tres el resultado debe ser “fizz”
- Para cualquier número divisible entre cinco el resultado debe ser “buzz”
- Para cualquier número divisible entre quince el resultado debe de ser “fizzbuzz”
- Para el resto de números el resultado debería ser el propio número recibido.

En el caso de que emergan requisitos durante el proceso de TDD, es importante que vayamos actualizando nuestra lista.

Una vez que tengamos la lista con los diferentes casos a cubrir, estaremos en disposición de abordar el problema. Para ello comenzaremos con un diseño inicial

⁹⁷<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-Green>

muy simple, en el que tendremos una función que recibe un número entero y devuelve un cero. El fichero *fizzBuzz.js* contendrá dicha función, quedando de esta manera:

FizzBuzz, primera implementación. Rojo

```
1 function fizzBuzz(n) {  
2     return 0;  
3 }  
4  
5 module.exports = fizzBuzz;
```

A continuación estaremos en disposición de poder escribir el primer test. Para ello vamos a comenzar por el primer caso de nuestra lista, en el que para el número uno el resultado debería ser uno:

FizzBuzz, primer test (rojo)

```
1 describe('FizzBuzz', () => {  
2     it('should return one if receive one', () => {  
3         const expected = 1;  
4         const result = fizzBuzz(1)  
5  
6         expect(result).toBe(expected);  
7     });  
8});
```

Ejecutamos y... ¡rojo!

Ejecutamos la suite de pruebas y el test falla como esperábamos. Este primer test en rojo es importante, ya que nos puede ayudar a detectar posibles errores en la construcción del test. Suele ser bastante común que el test falle, no porque el código de la implementación sea incorrecto, sino porque nos hemos equivocado a la hora de implementarlo.

Puedes acceder al ejemplo interactivo [desde aquí⁹⁸](#).

```
• FizzBuzz > should return one if receive one

expect(received).toBe(expected) // Object.is equality

Expected: 1
Received: 0

  6 |   const result = fizzBuzz(1)
  7 |
>  8 |     expect(result).toBe(expected);
    |     ^
  9 |   });
10 | }

at Object.toBe (fizzBuzz-test.js:8:20)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
```

Ejecutamos y ... ¡rojo!

Descargado en: www.detodoprogramacion.org

Pasamos a verde

A continuación, vamos a aplicar el concepto de implementación falsa que vimos en el capítulo anterior para conseguir que el test pase lo antes posible:

FizzBuzz, primera implementación. Rojo

```
1 function fizzBuzz(n) {
2   return 1;
3 }
4
5 module.exports = fizzBuzz;
```

Puedes acceder al ejemplo interactivo [desde aquí⁹⁹](#).

Ejecutamos el test y pasa correctamente. En estos primeros test tendremos implementaciones muy concretas. A medida que vayamos avanzando en la solución iremos generalizando la solución.

⁹⁸<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-0-RED>

⁹⁹<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-1-Green>

Añadiendo nuevas pruebas

Si revisamos nuestra lista, el siguiente caso a comprobar es: “Para el número tres el resultado debe ser fizz”. Vamos a ello:

FizzBuzz, segundo test (rojo)

```
1 describe('FizzBuzz', () => {
2   it('should return one if receive one', () => {
3     const expected = 1;
4     const result = fizzBuzz(1);
5
6     expect(result).toBe(expected);
7   });
8
9   it('should return fizz if receive three', () => {
10    const expected = "fizz";
11    const result = fizzBuzz(3)
12
13    expect(result).toBe(expected);
14  });
15});
```

Puedes acceder al ejemplo [desde aquí¹⁰⁰](#).

Ejecutamos la *suite* de test y el nuevo test falla. Es importante ejecutar todos los test para ir verificando la integridad de la implementación que vamos realizando.

¹⁰⁰<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-2-Red>

FizzBuzz

```
✓ should return one if receive one (5ms)
✗ should return fizz if receive three (5ms)
```

- **FizzBuzz > should return fizz if receive three**

```
expect(received).toBe(expected) // Object.is equality
```

```
Expected: "fizz"
```

```
Received: 1
```

Segundo test en rojo

A continuación escribimos la implementación mínima necesaria para que el nuevo test pase sin romper el anterior. Para ello podemos hacer uso de un condicional que devuelva el valor fizz en caso de que n sea igual a tres:

FizzBuzz, segundo caso.

```
1 function fizzBuzz(n) {
2   if(n==3)
3     return "fizz";
4
5   return 1;
6 }
```

Descargado en: www.detodoprogramacion.org

Una vez que tengamos el segundo test en verde, vamos abordar el tercer caso: “Para el número cinco el resultado debe ser buzz”. Escribimos el test:

FizzBuzz, tercer test (rojo)

```
1 it('should return buzz if receive five', () => {
2   const expected = "buzz";
3   const result = fizzBuzz(5)
4
5   expect(result).toBe(expected);
6 });
```

Rojo. Ahora debemos crear la implementación para este caso, al igual que para el caso anterior, podemos hacer uso de un condicional, en esta ocasión que retorne “fizz” cuando n sea igual a 5:

FizzBuzz, tercer caso.

```
1 function fizzBuzz(n) {  
2     if(n == 3)  
3         return "fizz";  
4  
5     if(n == 5)  
6         return "buzz";  
7  
8     return n;  
9 }
```

Puedes acceder a este paso [desde aquí](#).¹⁰¹

En este punto, antes de afrontar el siguiente caso y dado que hemos ido madurando nuestro conocimiento sobre el problema, podría ser interesante refactorizar para empezar a generalizar un poco la solución. Para ello, en los condicionales, en lugar de comprobar si se trata de un tres o un cinco, podríamos comprobar si el número es divisible entre tres o cinco y devolver fizz o buzz, respectivamente. Además, en lugar de retornar 1 para cualquier valor de n, podríamos retornar el propio valor de n:

FizzBuzz, refactorizando.

```
1 function fizzBuzz(n) {  
2     if(n % 3 == 0)  
3         return "fizz";  
4  
5     if(n % 5 == 0)  
6         return "buzz";  
7  
8     return n;  
9 }
```

Como puedes observar en el [ejemplo interactivo](#)¹⁰² los tests continúan en verde:

¹⁰¹<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-3-Green>

¹⁰²<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-3-Refactor>

```
PASS ./fizzBuzz-test.js
FizzBuzz
  ✓ should return one if receive one (50ms)
  ✓ should return fizz if receive three (1ms)
  ✓ should return buzz if receive five (1ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
```

Test en verde

Lo siguiente que debemos verificar es el caso en el que para n igual a quince la función devuelva *fizzbuzz*:

FizzBuzz, cuarto test (rojo)

```
1 it('should return fizzbuzz if receive fifteen', () => {
2   const expected = "fizzbuzz";
3   const result = fizzBuzz(15)
4
5   expect(result).toBe(expected);
6 });
```

Si ejecutamos el [ejemplo interactivo](#)¹⁰³ podemos observar que este último test está en rojo mientras que el resto continúa verde. Vamos a corregirlo: para ello, de forma similar a lo que hicimos en el refactor anterior, vamos a comprobar a través de un condicional que evalúe si el número recibido por parámetro es divisible entre quince:

¹⁰³<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-4-Red>

FizzBuzz, cuarto caso.

```
1 function fizzBuzz(n) {  
2     if(n % 15 == 0)  
3         return "fizzbuzz";  
4  
5     if(n % 3 == 0)  
6         return "fizz";  
7  
8     if(n % 5 == 0)  
9         return "buzz";  
10  
11    return n;  
12 }
```

Volvemos a ejecutar la suite de test y los test pasan correctamente¹⁰⁴. Probablemente ya te habrás dado cuenta de que hemos llegado a una solución válida, que resuelve el ejercicio, pero aún no lo hemos verificado. Para comprobarlo vamos a escribir en el mismo paso el resto de test que faltan:

FizzBuzz, resto de casos (verde)

```
1 it('should return fizz if receive any number divisible by three', () => {  
2     {  
3         const expected = "fizz";  
4         const result = fizzBuzz(9)  
5  
6         expect(result).toBe(expected);  
7     });  
8  
9     it('should return buzz if receive any number divisible by five', () => {  
10        const expected = "buzz";  
11        const result = fizzBuzz(25)  
12  
13        expect(result).toBe(expected);  
14    });
```

¹⁰⁴<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-4-Green>

```
15
16 it('should return buzz if receive any number divisible by fifteen', () \ 
17 => {
18   const expected = "fizzbuzz";
19   const result = fizzBuzz(30)
20
21   expect(result).toBe(expected);
22 });
23
24 it('should return the same number that receives', () => {
25   const expected = 4;
26   const result = fizzBuzz(4)
27
28   expect(result).toBe(expected);
29 });
```

Si ejecutamos el resto de casos en la [consola interactiva](#)¹⁰⁵ podemos comprobar que efectivamente nuestra implementación cumple para todos los casos que hemos enumerado en nuestra lista:

```
PASS ./fizzBuzz-test.js
FizzBuzz
✓ should return one if receive one (5ms)
✓ should return fizz if receive three (2ms)
✓ should return buzz if receive five (1ms)
✓ should return fizzbuzz if receive fifteen
✓ should return fizz if receive any number divisible by three
✓ should return buzz if receive any number divisible by five (1ms)
✓ should return buzz if receive any number divisible by fifteen
✓ should return the same number that receives (1ms)
```

Resto de casos en verde

¹⁰⁵<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-5-Green>

Refactorizando la solución, aplicando pattern matching.

Una vez que hemos cubierto todos los posibles escenarios, es el momento ideal de refactorizar para hacer limpieza y añadir legibilidad en el código que hemos obtenido. En este caso, podríamos añadirle algo de semántica, extrayendo una función lambda que compruebe si n es divisible por un número dado:

FizzBuzz, refactorizando.

```
1 function fizzBuzz(n) {
2     const divisibleBy = (divider, n) => n % divider == 0;
3
4     if(divisibleBy(15, n))
5         return "fizzbuzz";
6
7     if(divisibleBy(3, n))
8         return "fizz";
9
10    if(divisibleBy(5, n))
11        return "buzz";
12
13    return n;
14 }
15
16 module.exports = fizzBuzz;
```

Puedes acceder al ejemplo [desde aquí](#).¹⁰⁶

Tal y como comentamos en el capítulo de funciones de la sección de *Clean Code*, me gusta priorizar el estilo declarativo frente al imperativo, ya que nos permite obtener funciones mucho más expresivas. En nuestro problema encaja muy bien aplicar (pattern matching)[<https://softwarecrafters.io/typescript/union-types-pattern-matching-typescript>]. El pattern matching es una estructura típicamente

¹⁰⁶<https://repl.it/@SoftwareCrafter/TDD-Fizzbuzz-5-Final>

usada por lenguajes de programación funcionales y estáticamente tipados como Haskell, Scala o F#.

Dicha estructura nos permite comprobar un valor contra una serie de casos. Cuando un caso se cumple, se ejecuta la expresión asociada y se termina. Idealmente, los casos permiten especificar no solo valores constantes, si no también tipos, tipos con propiedades concretas o condiciones complejas. Conceptualmente, se parece a un switch mejorado.

A pesar de que JavaScript todavía no soporta pattern matching en su sintaxis ([ya existe una propuesta para ello¹⁰⁷](#)), podemos recurrir a bibliotecas, libraries en inglés, para suplir su carencia. En nuestro caso vamos a usar el paquete de npm llamado [x-match-expression¹⁰⁸](#).

FizzBuzz pattern matching.

```

1 import { match } from "x-match-expression";
2
3 export function fizzBuzz(n) {
4   const divisibleBy = divider => n => n % divider === 0;
5
6   return match(n)
7     .case(divisibleBy(15), "fizzbuzz")
8     .case(divisibleBy(5), "buzz")
9     .case(divisibleBy(3), "fizz")
10    .default(n);
11 }
```

Puedes acceder al ejemplo interactivo completo [desde aquí.¹⁰⁹](#)

Como podemos observar el código ahora se ve más conciso y expresivo. Aunque, si no estás acostumbrado al estilo funcional, puede resultar un poco confuso, así que vamos a explicarlo detalladamente.

En primer lugar, importamos la función *match*, la cual nos va a permitir hacer el pattern matching en sí. A continuación hemos [currificado¹¹⁰](#) la función *divisibleBy*

¹⁰⁷<https://github.com/tc39/proposal-pattern-matching>

¹⁰⁸<https://www.npmjs.com/package/x-match-expression>

¹⁰⁹<https://codesandbox.io/s/fizzbuzz-pattern-matching-tdd-p7oby>

¹¹⁰<https://es.wikipedia.org/wiki/Currificaci%C3%B3n>

creada en el paso anterior, para, aplicando composición, pasársela a cada uno de los *case*. Finalmente, se evalúa el valor de *n* a través de *match*, cada uno de los *cases* se irán ejecutando hasta que alguno cumpla la condición pasada en la expresión. Si no se cumple ninguno de los tres casos devolverá *n* por defecto.

Si quieres seguir practicando te recomiendo que visites la página [kata-log.rocks¹¹¹](https://kata-log.rocks), donde encontrarás muchos ejercicios con los que seguir practicando. Recuerda: ¡La práctica hace al maestro!

¹¹¹<https://kata-log.rocks/>

Siguientes pasos

En este *e-book* me he centrado en hacer hincapié en algunas buenas prácticas que hacen que nuestro código JavaScript sea legible, mantenible y testeable y, en definitiva, más tolerante al cambio. En futuras publicaciones profundizaremos en algunos conceptos por los que hemos pasado de puntillas, como por ejemplo el de arquitectura hexagonal. Probablemente trataremos este tema a través de ejemplos prácticos con NodeJS y TypeScript.

Por otro lado, una lectura que en mi opinión es muy recomendable y que además podemos entender como una continuación muy natural de los conceptos expuestos en la sección de “Testing”, es la nueva versión del libro de Carlos Blé, *Diseño ágil con TDD*¹¹². He tenido el placer de leer los primeros capítulos antes de que Carlos lo publicara y he de decir que está haciendo un gran trabajo. Tal y como él mismo dice, se trata de una edición totalmente nueva, actualizada, más práctica y fácil de leer.

Si te ha gustado el *e-book*, valóralo y compártelo en tus redes sociales. No dudes en plantear preguntas, aportes o sugerencias. ¡Estaré encantado de responder!

¹¹²<https://leanpub.com/tdd-en-castellano>

Referencias

- Clean Code: A Handbook of Agile Software Craftsmanship de Robert C. Martin¹¹³
- Clean Architecture: A Craftsman's Guide to Software Structure and Design de Robert C Martin¹¹⁴
- The Clean Coder: A Code of Conduct for Professional Programmers de Robert C. Martin¹¹⁵
- Test Driven Development. By Example de Kent Beck¹¹⁶
- Extreme Programming Explained de Kent Beck¹¹⁷
- Implementation Patterns de Kent Beck¹¹⁸
- Refactoring: Improving the Design of Existing Code de Martin Fowler¹¹⁹
- Design patterns de Erich Gamma, John Vlissides, Richard Helm y Ralph Johnson¹²⁰
- Effective Unit Testing de Lasse Koskela¹²¹
- The Art of Unit Testing: with examples in C# de Roy Osherove¹²²
- JavaScript Allonge de Reg “raganwald” Braithwaite¹²³
- You Don’t Know JS de Kyle Simpson¹²⁴
- Diseño Ágil con TDD de Carlos Blé¹²⁵
- Testing y TDD para PHP de Fran Iglesias¹²⁶
- Cursos de Codely.TV¹²⁷

¹¹³<https://amzn.to/2TUywwB>

¹¹⁴<https://amzn.to/2ph2wrZ>

¹¹⁵<https://amzn.to/2q5xgws>

¹¹⁶<https://amzn.to/2j1zWSH>

¹¹⁷<https://amzn.to/2VHQkNg>

¹¹⁸<https://amzn.to/2Hnh7cC>

¹¹⁹<https://amzn.to/2MGmeFy>

¹²⁰<https://amzn.to/2EW7MXv>

¹²¹<https://amzn.to/2VCcsbP>

¹²²<https://amzn.to/31ahpK6>

¹²³<https://leanpub.com/javascript-allonge>

¹²⁴<https://amzn.to/2OJ24xu>

¹²⁵<https://www.carlosble.com/libro-tdd/?lang=es>

¹²⁶<https://leanpub.com/testingtddparaphp>

¹²⁷<https://codely.tv/pro/cursos>

Descargado en: www.detodoprogramacion.org

- [Repository de Ryan McDermott¹²⁸](https://github.com/ryanmcdermott/clean-code-javascript)
- [Guía de estilo de Airbnb¹²⁹](https://github.com/airbnb/javascript)
- [El artículo “From Stupid to SOLID code” de Willian Durand¹³⁰](https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/)
- [Blog Koalite¹³¹](http://blog.koalite.com/)
- [El artículo de Software Crafters: “Programación Funcional en JavaScript” de Jose Manuel Lucas¹³²](https://softwarecrafters.io/javascript/introduccion-programacion-funcional-javascript)
- Conversaciones con los colegas [Carlos Blé¹³³](https://twitter.com/carlosble), [Dani García](https://twitter.com/DaniGarcia), [Patrick Hertling¹³⁴](https://twitter.com/PatrickHertling) y [Juan M. Gómez¹³⁵](https://twitter.com/_jmgomez_).

Descargado en: www.detodoprogramacion.org

¹²⁸<https://github.com/ryanmcdermott/clean-code-javascript>

¹²⁹<https://github.com/airbnb/javascript>

¹³⁰<https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/>

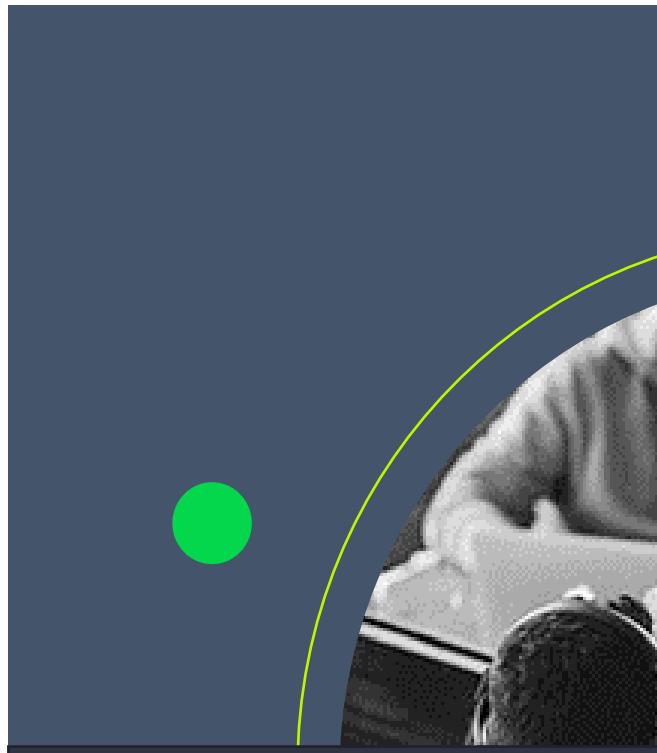
¹³¹<http://blog.koalite.com/>

¹³²<https://softwarecrafters.io/javascript/introduccion-programacion-funcional-javascript>

¹³³<https://twitter.com/carlosble>

¹³⁴<https://twitter.com/PatrickHertling>

¹³⁵https://twitter.com/_jmgomez_



El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la
Programación Java,
C/C++/C#, Visual.Net, SQL,
Python, Javascript, Oracle,
Algoritmos, CSS, Desarrollo
Web, Joomla, jquery, Ajax y
Mucho Mas...

VISITA

www.detodoprogramacion.org
www.detodopython.com
www.gratiscodigo.com

