SOFTWARE CRAFTERS
CRAFTSMANSHIP, DEVOPS & TECH

CLEAN CODE

APLICADO A JAVASCRIPT

"PROGRAMAR ES EL ARTE DE DECIRLE A OTRO HUMANO LO QUE QUIERES QUE EL ORDENADOR HAGA"

MIGUEL A. GÓMEZ

Clean Code aplicado a Javascript

Programar es el arte de decirle a otro humano lo que quieres que el ordenador haga

Software Crafters

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

Esta versión se publicó en 2019-06-29

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

Índice general

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 o código limpio para aprender a escribir código JavaScript más legible e intuitivo. 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 *clean code* para mejorar tu código JavaScript. {sample: false}

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.

[1] https://twitter.com/_jmgomez_
[2] https://twitter.com/ramonesteban78
[3] https://twitter.com/PatrickHertling
[4] https://twitter.com/carlosble

Sobre Software Crafters

Software Crafters 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[5], 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[6].

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://en.wikipedia.org/wiki/Software_craftsmanship
[6] https://es.wikipedia.org/wiki/Educaci%C3%B3n_permanente

Sobre el autor

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

Actualmente trabajo como **Senior Software Engineer** en Bellefield Systems[8], una empresa estadounidense dedicada al desarrollo de soluciones software para el sector de la abogacía, en la cual he participado como desarrollador principal en diferentes proyectos.

Entre los puestos más importantes destacan **desarrollador móvil multiplataforma con Xamarin y C#** y el de desarrollador **fullStack**, el puesto que desempeño actualmente. En este último, aplico un estilo de programación híbrido entre orientación a objetos y programacion funcional reactiva (FRP)[9], tanto para el **frontend** con Typescript[10], RxJS[11] y ReactJS[12], como para el **backend** con **Typescript**, **NodeJS**, **RxJS y MongoDB**, además de gestionar los procesos DevOps[13] con **Docker** y **Azure**.

Por otro lado, soy cofundador de la start-up Omnirooms.com[14], 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[15], una

[7] https://www.haskell.org/
[8] https://www.bellefield.com/
[9] https://en.wikipedia.org/wiki/Functional_reactive_programming
[10] https://softwarecrafters.io/typescript/typescript-javascript-introduccion
[11] https://softwarecrafters.io/javascript/introduccion-programacion-reactiva-rxjs
[12] https://softwarecrafters.io/react/tutorial-react-js-introduccion
[13] https://es.wikipedia.org/wiki/DevOps
[14] https://www.omnirooms.com
[15] https://softwarecrafters.io/

web sobre artesanía del software, DevOps y tecnologías software con aspiraciones a plataforma de formación y consultoría.

Otros intereses

Además del desarrollo de software y el emprendimiento, estoy muy interesado en el mundo de las finanzas y de los mercados. Mi filosofía de inversión se basa en el value investing[16] combinado con principios de la escuela austriaca de economía[17].

Por otro lado, siento devoción por el estoicismo[18], una filosofía que predica ideas vitales como vivir conforme a la naturaleza del ser humano, profesar el agradecimiento, ignorar las opiniones de los demás, centrarse en el presente o mantenerse firme ante las adversidades, siendo consciente de lo que está bajo nuestro control y lo que no. Aunque, como Séneca, no siempre predico con el ejemplo.

También practico entrenamientos de fuerza con patrones de movimiento en los que se involucra todo el cuerpo de manera coordinada. Soy aficionado al powerlifting[19] y al culturismo natural[20], motivo por el cual le doy mucha importancia a la nutrición. He tratado de incorporar a mis hábitos alimenticios un enfoque evolutivo, priorizando la comida real (procedente de ganadería y agricultura sostenible siempre que sea posible) y evitando los alimentos ultraprocesados.

[16] https://en.wikipedia.org/wiki/Value_investing
[17] https://es.wikipedia.org/wiki/Escuela_austriaca
[18] https://es.wikipedia.org/wiki/Estoicismo
[19] https://en.wikipedia.org/wiki/Powerlifting
[20] https://en.wikipedia.org/wiki/Natural_bodybuilding

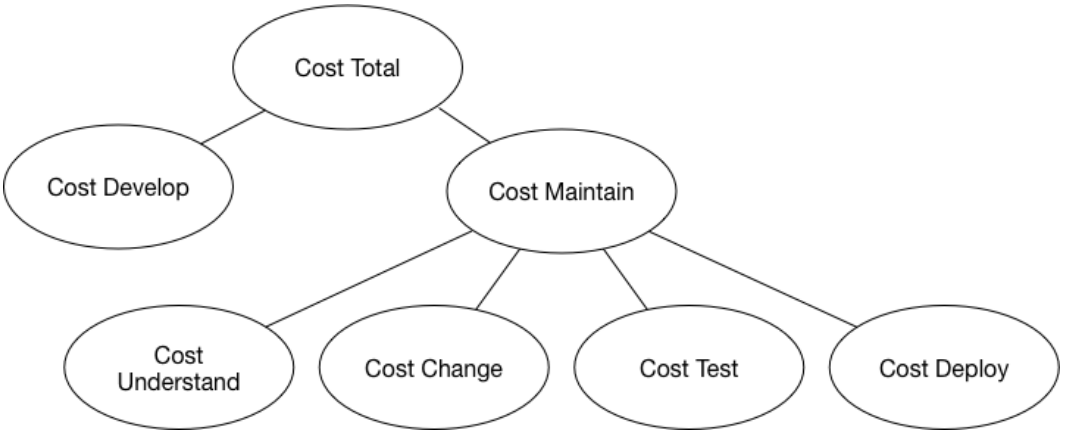
Introducción

"La fortaleza y la debilidad de JavaScript reside en que te permite hacer cualquier cosa, tanto para bien como para mal." – Reginald Braithwaite[21]

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 importantes[22] (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 Patterns*[23], el coste de mantenimiento viene dado por la suma de los costes de entender el código, cambiarlo, testearlo y desplegarlo.

[21] https://twitter.com/raganwald
[22] https://stackshare.io/javascript
[23] https://amzn.to/2BHRU8P



Esquema de costes de Kent Beck

La idea de este libro es tratar de exponer algunas maneras de minimizar el coste relacionado con la parte de entender el código, para ello trataré de sintetizar y ampliar algunos de los conceptos relacionados con esto que exponen Robert C. Martin[24], Kent Beck[25], Ward Cunningham[26] y otros autores aplicándolos a JavaScript.

[24] https://twitter.com/unclebobmartin
[25] https://twitter.com/KentBeck
[26] https://twitter.com/WardCunningham

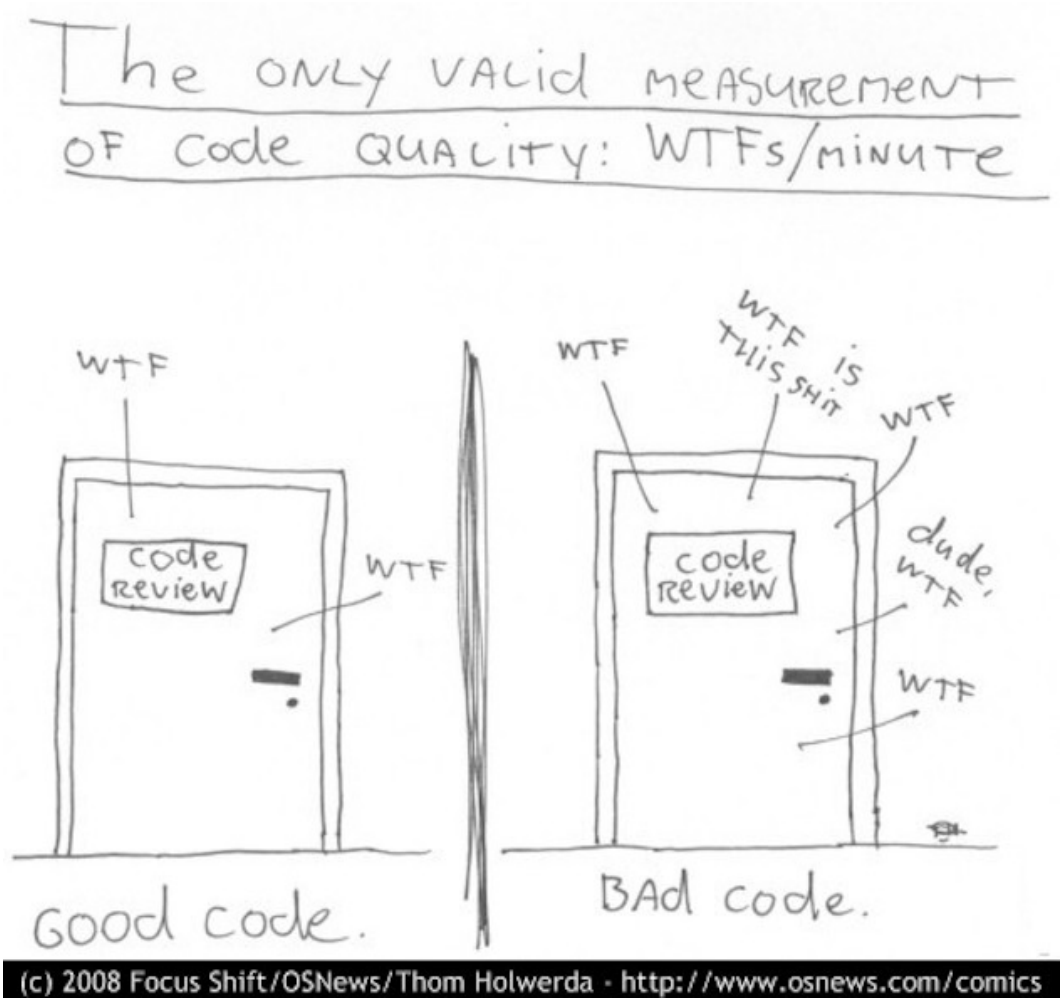
¿Qué es Clean Code?

"Programar es el arte de decirle a otro humano lo que quieres que el ordenador haga." – Donald Knuth[27]

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. Martin[28], también conocido como Uncle Bob, publicó su libro "Clean Code: A Handbook of Agile Software Craftsmanship[29]" 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.

[27] https://es.wikipedia.org/wiki/Donald_Knuth
[28] https://twitter.com/unclebobmartin
[29] https://amzn.to/2TUywwB



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

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*[30]

Los desarrolladores solemos escribir código sin la intención explícita

[30] https://twitter.com/carlosble

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.

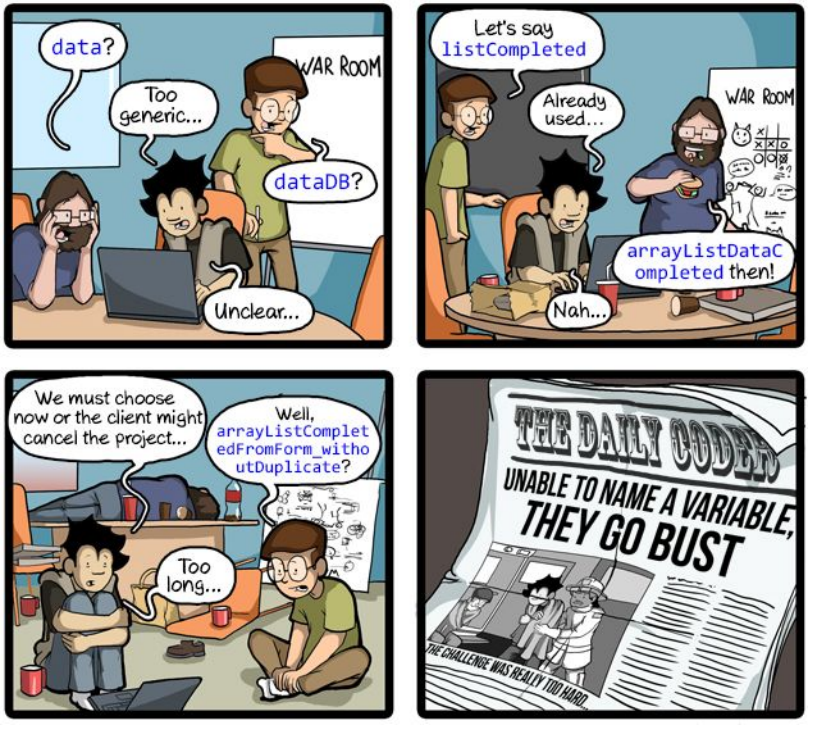
Tratar de entender el código de un tercero o incluso el que escribimos nosotros mismos hace tan solo unas semanas, se puede volver una tarea realmente difícil. Es por ello que hacer un esfuerzo extra para que nuestra solución sea legible e intuitiva es la base para reducir los costes de mantenimiento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

Variables y nombres

"Nuestro código tiene que ser simple y directo, debería leerse con la misma facilidad que un texto bien escrito" – Grady Booch[31]

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



Viñeta de Commit Strip sobre el nombrado de variables.

[31] https://es.wikipedia.org/wiki/Grady_Booch

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

Nombres pronunciables y expresivos

Los nombres, imprescindiblemente en inglés, deben ser pronunciables. Esto quiere decir que no deben ser abreviaturas ni llevar guion 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.

Ejemplo 1: Nombres pronunciables y expresivos

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

Uso correcto de var, let y const

Debemos evitar a toda costa el uso de var, ya que define las variables con alcance global. Esto no ocurre con las variables definidas con let y const, ya que se definen para un ámbito en concreto.

La diferencia entre let y const radica en que a esta última no se le puede reasignar su valor (aunque sí modificarlo). Es por ello que usar const en variables a las que no tengamos pensado cambiar su valor puede ayudarnos a mejorar la intencionalidad de nuestro código.

Ejemplo 2: Uso correcto de var, let y const

```
// old school JavaScript
var variable = 5;
{
  console.log('variable'); // 5
  var variable = 10;
}

console.log(variable); // 10
variable = variable*2;
console.log(variable); // 20

// modern JavaScript (let)
let variable = 5;

{
   console.log(variable); // error
   let variable = 10;
}

console.log(variable); // 5
variable = variable*2;
console.log(variable); // 10

// modern JavaScript (const)
const variable = 5;
variable = variable*2; // error
console.log(variable); // doesn't get here
```

Evitar que los nombres contengan información técnica

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 húngara[32], el tipo de clase, etc. Esto sí se admite en desarrollo de software horizontal o librerías de propósito general.

Ejemplo 3: Evitar que los nombres contengan información técnica

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

Léxico coherente

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.

[32] https://es.wikipedia.org/wiki/Notaci%C3%B3n_h%C3%BAngara

Ejemplo 4: Léxico coherente

```
//bad
getUserInfo();
getClientData();
getCustomerRecord();

//better
getUser()
```

Usa el nombre adecuado 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:

Ejemplo 5: Arrays

```
//bad
const fruit = ['manzana', 'platano', 'fresa'];
// regular
const fruitList = ['manzana', 'platano', 'fresa'];
// good
const fruits = ['manzana', 'platano', 'fresa'];
// better
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.

Ejemplo 6: Booleanos

```
//bad
const open = true;
const write = true;
const fruit = true;

// good
const isOpen = true;
const canWrite = true;
const hasFruit = true;
```

Números

Para los números es interesante escoger palabras que describan números, como “min”, “max”, “total”:

Ejemplo 7: Números

```
//bad
const fruits = 3;

//better
const maxFruits = 5;
const minFruits = 1;
const totalFruits = 3;
```

Funciones

Los nombres de las funciones deben representar acciones, por ello 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.

Ejemplo 8: Funciones

```
//bad
createUserIfNotExists()
updateUserIfNotEmpty()
sendEmailIfFieldsValid()

//better
createUser(...)
updateUser(...)
sendEmail()
```

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

Ejemplo 9: Funciones de acceso, modificación o predicado

```
getUser()
setUser(...)
isValidUser()
```

Clases

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

Funciones

"Sabemos que estamos desarrollando código limpio cuando cada función hace exactamente lo que su nombre indica" – Ward Cunningham[33]

Las funciones son la entidad organizativa más básica en cualquier programa. Es por ello que deben resultar sencillas de leer y de entender, además de transmitir claramente su intención. A continuación veremos algunas pautas que creo que nos pueden ser de ayuda a la hora de escribir buenas funciones.

Tamaño reducido y hacer una única cosa

La simplicidad es un pilar fundamental a la hora de tratar de escribir buen código, es por ello que la primera recomendación es que nuestras funciones **deben de tener un tamaño reducido**. Normalmente suelo escribir funciones de 4 o 5 líneas, en algunas ocasiones puedo llegar a 15 o 20 líneas, pero no me excedo nunca de esa cantidad.

Si te sueles exceder de esas 15 o 20 líneas es que tu función hace demasiadas cosas, lo que nos lleva a la segunda recomendación y quizás la más importante: las funciones deben hacer una única cosa y hacerla bien.

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

Limita el número de argumentos

Otra 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:

Ejemplo 10: Limita el número de argumentos

```
function createMenu(title, body, buttonText, cancellable) {
  // ...
}

function createMenu({ title, body, buttonText, cancellable }) {
  // ...
}

createMenu({
  title: 'Foo',
  body: 'Bar',
  buttonText: 'Baz',
  cancellable: true
});
```

Prioriza el estilo declarativo frente al imperativo

Aunque JavaScript no es un lenguaje funcional puro, sí que nos ofrece algunos elementos de la programación funcional 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 nos permitirá obtener funciones mucho más expresivas y de tamaño más reducido.

Ejemplo 11: Prioriza el estilo declarativo frente al imperativo

```
//worse
var orders = [
    { productTitle: "Product 1", amount: 10 },
    { productTitle: "Product 2", amount: 30 },
    { productTitle: "Product 3", amount: 20 },
    { productTitle: "Product 4", amount: 60 }
];

var totalAmount = 0;

for (var i = 0; i < orders.length; i++) {
    totalAmount += orders[i].amount;
}

console.log(totalAmount); // 120

//better
let shoppingCart = [
    { productTitle: "Product 1", amount: 10 },
    { productTitle: "Product 2", amount: 30 },
    { productTitle: "Product 3", amount: 20 },
    { productTitle: "Product 4", amount: 60 }
];

const sumAmount = (currentAmount, order) => currentAmount + order.amoun\
t;

function getTotalAmount(shoppingCart) {
    return shoppingCart.reduce(sumAmount, 0);
}

getTotalAmount(shoppingCart); // 120
```

Usa 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:

Ejemplo 12: Usa funciones anónimas

```
const stuffList = [
    { isEnabled: true, name: 'justin' },
    { isEnabled: false, name: 'lauren' },
    { isEnabled: false, name: 'max' },
];

const filteredStuff = stuffList.filter(stuff => !stuff.isEnabled);
```

La funcion stuff ⇒ !stuff.isEnabled es un predicado tan simple que extraerlo no tiene demasiado sentido.

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.

Ejemplo 12: Transparencia referencial

```
//bad
let counter = 1;

function increaseCounter(value) {
  counter = value + 1;
}

increaseCounter(counter);
console.log(counter); // 2

//better
let counter = 1;

function increaseCounter(value) {
  return value + 1;
}

increaseCounter(counter); // 2
console.log(counter); // 1
```

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.

Ejemplo 13: Principio DRY

```
//worse
function showDeveloperList(developers) {
  developers.forEach((developer) => {
    const expectedSalary = developer.calculateExpectedSalary();
    const experience = developer.getExperience();
    const githubLink = developer.getGithubLink();
    const data = {
      expectedSalary,
      experience,
      githubLink
    };

    render(data);
  });
}

function showManagerList(managers) {
  managers.forEach((manager) => {
    const expectedSalary = manager.calculateExpectedSalary();
    const experience = manager.getExperience();
    const portfolio = manager.getMBAProjects();
    const data = {
      expectedSalary,
      experience,
      portfolio
    };

    render(data);
  });
}
```

```
31
32  //better
33  function showEmployeeList(employees) {
34    const getCVLink = (employee) =>
35      employee.type == 'manager'
36        ? employee.getMBAProjects()
37        : employee.getGithubLink()
38
39    employees.forEach(employee => render({
40      employee.calculateExpectedSalary(),
41      employee.getExperience(),
42      getCVLink(employee)
43    });
44  };
```

Evita el uso de comentarios

"No comentes el código mal escrito, reescríbelo" – Brian W. Kernighan[34]

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.

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

Viñeta de Commit Strip sobre los comentarios.

Formato coherente

"El buen código siempre parece estar escrito por alguien a quien le importa." – Michael Feathers[35]

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:

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.

Tamaño de los archivos

Evita crear archivos excesivamente grandes o archivos demasiado cortos (de 5 a 6 líneas). Lo ideal sería movernos en un intervalo de entre 200 y 500 líneas.

[35] https://twitter.com/mfeathers?lang=es

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 identación, directamente ni te lo corregía.

Clases

"Si quieres ser un programador productivo esfuérzate en escribir código legible" – Robert C. Martin[36]

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.

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.

Veamos un ejemplo:

[36] https://twitter.com/unclebobmartin

Ejemplo 14: Principio de responsabilidad única

```
class UserSettings {
    private user: User;
    private settings: Settings;

    constructor(user) {
        this.user = user;
    }

    changeSettings(settings) {
        if (this.verifyCredentials()) {
        // ...
        }
     }

     verifyCredentials() {
        // ...
     }
}
```

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*.

Ejemplo 15: Principio de responsabilidad única refactorizado

```
class UserAuth{
    private user: User;

    constructor(user: User){
        this.user = user
    }

    verifyCredentials(){
        //...
    }
}

class UserSettings {
    private user: User;
    private settings: Settings;
    private auth: UserAuth;

    constructor(user: User, auth:UserAuth) {
        this.user = user;
        this.auth = auth;
    }

    changeSettings(settings) {
        if (this.auth.verifyCredentials()) {
        // ...
        }
    }
}
```

Esta forma de diseñar las clases nos permite mantener la responsabilidades bien definidas, además de contener el tamaño de las mismas.

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.

Ejemplo 16: Organización de clases

```
class Post {
    private title : string;
    private content: number;
    private createdAt: number;

    static create(title:string; content:string){
        return new Post(title, content)
    }

    private constructor(title:string; content:string){
        this.setTitle(title);
        this.setContent(content);
        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:

Ejemplo 17: composición frente a la herencia

```
class Employee {
    private this.name: string;
    private this.email: string;

    constructor(name:string, email:string) {
        this.name = name;
        this.email = email;
    }

  // ...
}

class EmployeeTaxData extends Employee {
    private this.ssn: string;
    private this.salary: number;

    constructor(ssn:string, salary:number) {
        super();
        this.ssn = ssn;
        this.salary = salary;
     }
```

```
  //...
}
```

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:

Ejemplo 18: composición frente a la herencia

```
class EmployeeTaxData{
    private this.ssn: string;
    private this.salary: number;

    constructor(ssn:string, salary:number) {
        super();
        this.ssn = ssn;
        this.salary = salary;
     }
  //...
}

class Employee {
    private this.name: string;
    private this.email: string;
    private this.taxData: EmployeeTaxData;

    constructor(name:string, email:string) {
        this.name = name;
        this.email = email;
    }

    setTaxData(taxData:EmployeeTaxData){
        this.taxData = taxData;
    }
  // ...
}
```

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.

Siguientes pasos

Aunque en este ebook me he centrado en la parte de clean code sobre la legibilidad del código, he dejado para futuras publicaciones otras cuestiones fundamentales relacionadas con cómo conseguir que nuestro código sea más intuitivo, no solo más legible, como los principios SOLID, patrones de diseño, arquitectura limpia, límites, testing, etc.

El término clean code realmente abarca mucho más de lo expuesto en este artículo e incluso de lo que expone Uncle Bob en su libro. Creo que si miramos más allá del código, clean code se convierte en una actitud, un deseo de hacer las cosas bien, de seguir buenas prácticas que nos conviertan en mejores profesionales.

Si te ha gustado el ebook, valora y comparte en tus redes sociales. No dudes en plantear preguntas, aportes o sugerencias. ¡Estaré encantado de responder!

Referencias

- Clean Code: A Handbook of Agile Software Craftsmanship de Robert C. Martin[37]
- Design patterns de Erich Gamma, John Vlissides, Richard Helm y Ralph Johnson[38]
- Implementation Patterns de Kent Beck[39]
- Repositorio de Ryan McDermott[40]
- Guía de estilo de Airbnb[41]
- Conversaciones con los colegas Carlos Blé[42], Dani García, Patrick Hertling[43] y Juan M. Gómez[44].

[37] https://amzn.to/2TUywwB
[38] https://amzn.to/2EW7MXv
[39] https://amzn.to/2Hnh7cC
[40] https://github.com/ryanmcdermott/clean-code-javascript
[41] https://github.com/airbnb/javascript
[42] https://twitter.com/carlosble
[43] https://twitter.com/PatrickHertling
[44] https://twitter.com/_jmgomez_