SOFTWARE CRAFTERS

Clean JavaScript

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

Índice general

DEMO

No dejes escapar la oportunidad, estoy convencido de que te puede aportar algún detalle de mucho valor.

Recuerda que si no es lo que esperas, te devolvemos tu dinero. Durante los primeros 15 días de compra, puedes obtener un reembolso del 100%. El riesgo es cero y el beneficio podría ser muy elevado.

Puedes adquirir el e-book desde aquí.[1]

[1] https://softwarecrafters.io/cleancode-solid-testing-js

Prefacio

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

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

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

Qué no es este libro

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

Agradecimientos

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

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

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

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

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

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

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

Sobre Software Crafters

Software Crafters[6] es una web sobre **artesanía del software**, **DevOps y tecnologías software** con aspiraciones a plataforma de formación y consultoría.

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

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

[6] https://softwarecrafters.io/

Sobre el autor

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

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

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

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

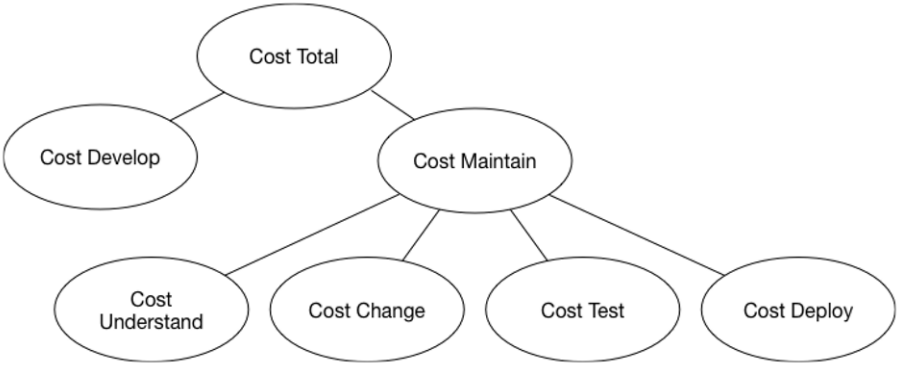
[7] https://www.haskell.org/
[8] https://en.wikipedia.org/wiki/Functional_reactive_programming
[9] https://softwarecrafters.io/typescript/typescript-javascript-introduccion
[10] https://softwarecrafters.io/javascript/introduccion-programacion-reactiva-rxjs
[11] https://softwarecrafters.io/react/tutorial-react-js-introduccion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

Introducción

"La fortaleza y la debilidad de JavaScript reside en que te permite hacer cualquier cosa, tanto para bien como para mal". – Reginald Braithwaite[15]

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[16] (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*[17], 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

[15] https://twitter.com/raganwald
[16] https://stackshare.io/javascript
[17] https://amzn.to/2BHRU8P

Además de los costes mencionados, Dan North[18] famoso por ser uno de los creadores de BDD[19], también hace hincapié en el coste de oportunidad y en el coste por el retraso en las entregas. Aunque en este libro no voy a entrar en temas relacionados con la gestión de proyectos, si que creo que es importante ser conscientes de cuales son los costes que generamos los desarrolladores y sobre todo en qué podemos hacer para minimizarlos.

En la primera parte del libro trataré de exponer algunas maneras de minimizar el coste relacionado con la parte de entender el código, para ello trataré de sintetizar y ampliar algunos de los conceptos relacionados con esto que exponen Robert C. Martin[20], Kent Beck[21], Ward Cunningham[22] 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.

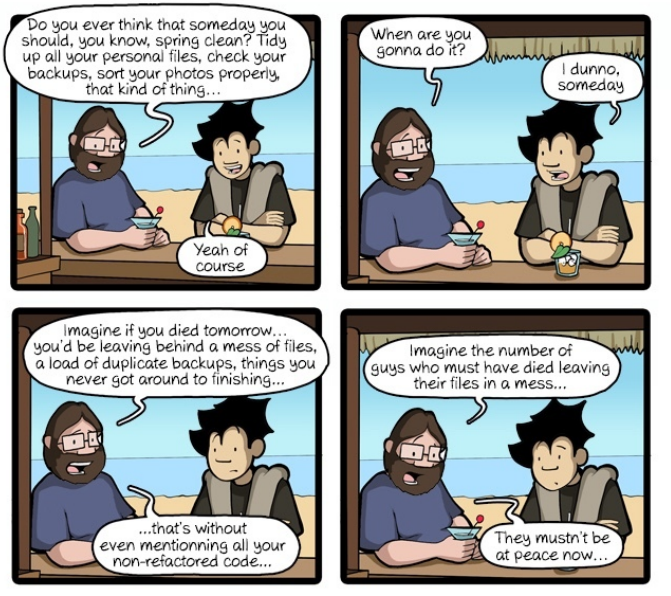
En la tercera y última parte, trataremos de ver cómo nos pueden ayudar los test unitarios y el diseño dirigido por test (TDD) a escribir código de mayor calidad y robustez, lo cual nos ayudará, además de a prevenir la deuda técnica, a minimizar el coste relacionado con testear el software.

[18] https://twitter.com/tastapod?lang=es
[19] https://en.wikipedia.org/wiki/Behavior-driven_development
[20] https://twitter.com/unclebobmartin
[21] https://twitter.com/KentBeck
[22] https://twitter.com/WardCunningham

Deuda técnica

"**Un Lannister siempre paga sus deudas**". – Game of Thrones

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



La última deuda técnica. Viñeta de CommitStrip

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

Tipos de deuda

...

Refactoring, las deudas se pagan

...

Mejor prevenir que curar, las reglas del diseño simple

...

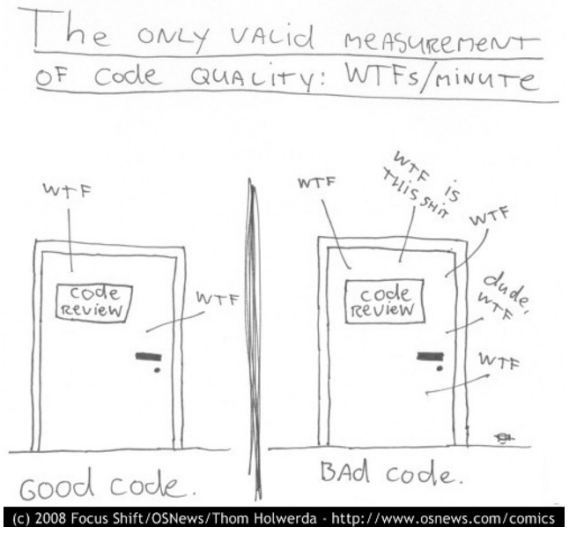
SECCIÓN I: CLEAN CODE

¿Qué es Clean Code?

"Programar es el arte de decirle a otro humano lo que quieres que el ordenador haga". – Donald Knuth[23]

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[24], también conocido como Uncle Bob, publicó su libro "Clean Code: A Handbook of Agile Software Craftsmanship[25]" 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

[23] https://es.wikipedia.org/wiki/Donald_Knuth
[24] https://twitter.com/unclebobmartin
[25] 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*[26]

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.

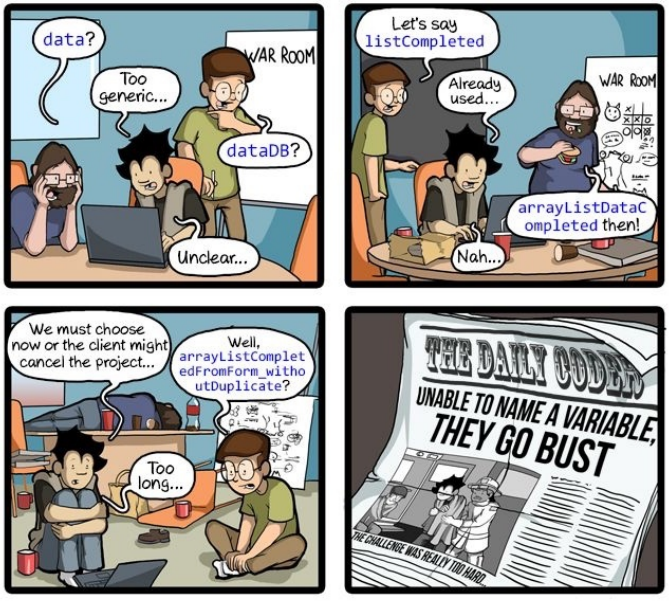
A continuación veremos algunas de las secciones del libro de Uncle Bob que más relacionadas están con la legibilidad del código. Si conoces el libro o lo has leído, podrás observar que he añadido algunos conceptos y descartado otros, además de incluir ejemplos sencillos aplicados a JavaScript.

[26] 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 Booch[27]

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

[27] https://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

```
var variable = 5;
{
  console.log('inside', variable); // 5
  var variable = 10;
}

console.log('outside', variable); // 10
variable = variable * 2;
console.log('changed', variable); // 20
```

Puedes acceder al ejemplo interactivo desde aquí[28]

[28] https://repl.it/@SoftwareCrafter/CLEAN-CODE-var

Ejemplo de uso de let

```
let variable = 5;

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

console.log('outside', variable); // 5
variable = variable * 2;
console.log('changed', variable); // 10
```

Puedes acceder al ejemplo interactivo desde aquí[29]

Ejemplo de uso de const

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

Puedes acceder al ejemplo interactivo desde aquí[30]

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.

[29] https://repl.it/@SoftwareCrafter/CLEAN-CODE-let
[30] https://repl.it/@SoftwareCrafter/CLEAN-CODE-const

Nombres pronunciables y expresivos

```
//bad
const yyyymmdstr = moment().format('YYYY/MM/DD');

//better
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 húngara[31], 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

```
//bad
class AbstractUser(){...}

//better
class User(){...}
```

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 tremendamente importante a la hora de obtener un léxico coherente.

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

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

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

//better
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

```
//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.

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* o *total*:

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 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

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

//better
createUser(...)
updateUser(...)
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

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

```
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name;
  }

  set name(newName) {
    this._name = newName;
  }
}

let person = new Person('Miguel');
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:

Ámbito global

```
let greeting = 'hello world!';

function greet(){
        console.log(greeting);
}

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

```
{
        let greeting = “Hello world!”;
        var lang = “English”;
        console.log(greeting); //Hello world!
}

console.log(lang);//”English”
console.log(greeting);//// Uncaught ReferenceError: greeting is not def\
ined
```

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

```
const number = 10;
function printNumber() {
  console.log(number);
}

function app() {
  const number = 5;
  printNumber();
}

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

```
var greet = "Hi";
(function () {
    console.log(greet);// "undefined"
    var greet = "Hello";
    console.log(greet); //”Hello”
})();
```

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

```
var greet = "Hi";
(function () {
    var greet;
    console.log(greet);// "undefined"
    greet = "Hello";
    console.log(greet); //”Hello”
})();
```

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 Cunningham[32]

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

...

Expresión de una función

...

Expresiones con funciones flecha (*arrow functions*)

...

Funcionamiento del objeto *this* en *arrow functions*

...

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

Funciones autoejecutadas IIFE

...

Parámetros y argumentos

...

Limita el número de argumentos

...

Parámetros por defecto

...

Parámetro *rest* y operador *spread*

...

Tamaño y niveles de indentación

...

Cláusulas de guarda

...

Evita el uso de *else*

...

Prioriza las condiciones asertivas

…

Estilo declarativo frente al imperativo

…

Funciones anónimas

…

Transparencia referencial

…

Principio DRY

…

Command–Query Separation (CQS)

…

Algoritmos eficientes

…

Notación O grande (*big O*)

…

Clases

“Si quieres ser un programador productivo esfuérzate en escribir código legible”.
– Robert C. Martin[33]

…

Prototype y *ECMAScript* moderno

…

Constructores y funciones constructoras

…

Métodos

…

Herencia y cadena de prototipos

…

Tamaño reducido

…

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

Organización

...

Prioriza la composición frente a la herencia

...

Comentarios y formato

Evita el uso de comentarios

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

…

Formato coherente

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

…

Problemas similares, soluciones simétricas

…

Densidad, apertura y distancia vertical

…

Lo más importante primero

…

Indentación

…

[34] https://es.wikipedia.org/wiki/Brian_Kernighan
[35] https://twitter.com/mfeathers?lang=es

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.

- **S**ingleton: patrón singleton
- **T**ight Coupling: alto acoplamiento
- **U**ntestability: código no testeable
- **P**remature optimization: optimizaciones prematuras
- **I**ndescriptive Naming: nombres poco descriptivos
- **D**uplication: duplicidad de código

¿Qué es un *code smell*?

...

El patrón singleton

...

Alto acoplamiento

...

Acoplamiento y cohesión

...

Código no testeable

...

Optimizaciones prematuras

...

Complejidad esencial y complejidad accidental

...

Nombres poco descriptivos

...

Duplicidad de código

...

Duplicidad real

...

Duplicidad accidental

...

Principios SOLID al rescate

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

El acrónimo SOLID fue creado por Michael Feathers[36], y, cómo 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.

- **S**ingle Responsibility: Responsabilidad única.
- **O**pen/Closed: Abierto/Cerrado.
- **L**iskov substitution: Sustitución de Liskov.
- **I**nterface segregation: Segregación de interfaz.
- **D**ependency 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.

[36] https://michaelfeathers.silvrback.com/

SRP - Principio de responsabilidad única

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

...

¿Qué entendemos por responsabilidad?

...

Aplicando el SRP

...

Detectar violaciones del SRP:

...

OCP - Principio Abierto/Cerrado

"Todas las entidades software deberían estar abiertas a extensión, pero cerradas a modificación". – Bertrand Meyer[37]

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 habría 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

[37] https://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

```
const axios = require('axios');

class TodoExternalService{

  requestTodoItems(callback){
    const url = 'https://jsonplaceholder.typicode.com/todos/';

    axios
      .get(url)
      .then(callback)
  }
}

new TodoExternalService()
  .requestTodoItems(response => console.log(response.data))
```

Puedes acceder al ejemplo interactivo desde aquí[38].

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 quisiéramos 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.

[38] 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

```
class ClientWrapper{
  makeGetRequest(url, callback){
    return axios
      .get(url)
      .then(callback);
  }
}
```

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

```
//infrastructure/ClientWrapper
const axios = require('axios');

export class ClientWrapper{
  makeGetRequest(url, callback){
    return axios
      .get(url)
      .then(callback);
  }
}

//domain/TodoService
export class TodoService{
  client;

```

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

Puedes acceder al ejemplo completo desde aquí.[39]

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 tremendamente positivo para nosotros, ya que si quisieramos cambiar la librería *axios* por *fetch*, por ejemplo, tan solo tendríamos que hacerlo en nuestra clase *ClientWrapper:*

[39] https://repl.it/@SoftwareCrafter/SOLID-OCP1

Patrón adaptador

```
export class ClientWrapper{
  makeGetRequest(url, callback){
    return fetch(url)
      .then(response => response.json())
      .then(callback)
  }
}
```

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

Detectar violaciones del OCP

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

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

LSP - Principio de sustitución de Liskov

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

…

Aplicando el LSP

…

Detectar violaciones del LSP

…

ISP - Principio de segregación de la interfaz

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

...

Aplicando el ISP

...

Detectar violaciones del ISP

...

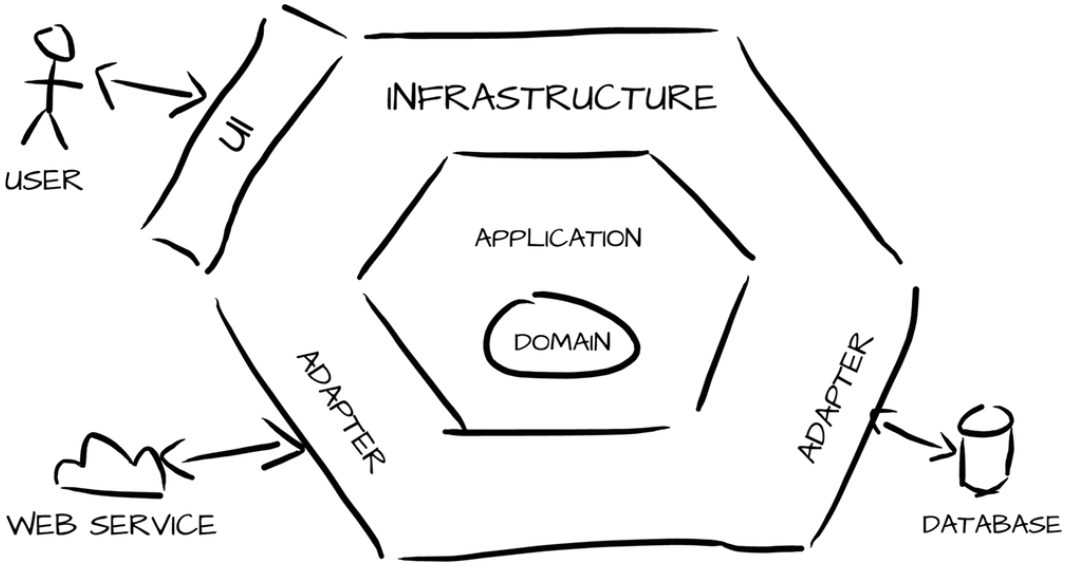
DIP - Principio de inversión de dependencias

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

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

Módulos de alto nivel y módulos de bajo nivel

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



Esquema arquitectura hexagonal

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

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

Depender de abstracciones

...

Inyección de dependencias

...

Aplicando el DIP

...

Detectando violaciones del DIP

...

Introducción al testing

"El testing de software puede verificar la presencia de errores pero no la ausencia de ellos". – Edsger Dijkstra[40]

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-

[40] https://es.wikipedia.org/wiki/Edsger_Dijkstra

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

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

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

Tipos de tests de software

...

¿Qué entendemos por testing?

...

Test manuales vs automáticos

...

Test funcionales vs no funcionales

...

Tests funcionales

...

Tests no funcionales

...

Pirámide de testing

...

Antipatrón del cono de helado

...

Tests unitarios

"**Nos pagan por hacer software que funcione, no por hacer tests**". – Kent Beck.

El *unit testing*, o test unitarios en castellano, no es un concepto nuevo en el mundo del desarrollo de software. Ya en la década de los años 70, cuando surgió el lenguaje Smalltack[41], 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.

[41] https://es.wikipedia.org/wiki/Smalltalk

Características de los tests unitarios

...

Anatomía de un test unitario

...

Jest, el framework de testing JavaScript definitivo

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

Históricamente JavaScript ha sido uno de los lenguajes con más *frameworks* y librerías de test, pero a la vez es uno de los lenguajes con menos cultura de *testing* entre los miembros de su comunidad. Entre dichos *frameworks* y librerías de *testing* automatizado destacan Mocha[42], Jasmine[43] y Jest[44], 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 React[45], es un framework de testing generalista que podemos utilizar en cualquier situación. Se trata de un framework flexible, rápido y con un output sencillo y comprensible, que nos permite completar un ciclo de feedback rápido y con la máxima información en cada momento.

Características

...

Instalación y configuración

...

[42] https://mochajs.org/
[43] https://jasmine.github.io/
[44] https://facebook.github.io/jest/
[45] https://softwarecrafters.io/react/tutorial-react-js-introduccion

Nuestro primer test

...

Aserciones

...

Organización y estructura

...

Gestión del estado: before y after

...

Code coverage

...

TDD - Test Driven Development

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

El TDD fue desarrollada por Kent Beck a finales de la década de los 90 y forma parte de la metodología *extreme programming*[46]. Su autor y los seguidores del TDD aseguran que con esta técnica se consigue un código más tolerante al cambio, robusto, seguro, más barato de mantener e, incluso, una vez que te acostumbras a aplicarlo, promete una mayor velocidad a la hora de desarrollar.

Las tres leyes del TDD

...

El ciclo Red-Green-Refactor

...

TDD como herramienta de diseño

...

Estrategias de implementación, de rojo a verde.

...

[46] https://en.wikipedia.org/wiki/Extreme_programming

Implementación falsa

...

Triangular

...

Implementación obvia

...

Limitaciones del TDD

...

TDD Práctico: La kata FizzBuzz

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

...

Las katas de código

...

La kata FizzBuzz

...

Descripción del problema

...

Diseño de la primera prueba

...

Ejecutamos y... ¡rojo!

...

Pasamos a verde

...

Añadiendo nuevas pruebas

...

Refactorizando la solución, aplicando pattern matching.

...

Referencias

- Clean Code: A Handbook of Agile Software Craftsmanship de Robert C. Martin[47]
- Clean Architecture: A Craftsman's Guide to Software Structure and Design de Robert C Martin[48]
- The Clean Coder: A Code of Conduct for Professional Programmers de Robert C. Martin[49]
- Test Driven Development. By Example de Kent Beck[50]
- Extreme Programming Explained de Kent Beck[51]
- Implementation Patterns de Kent Beck[52]
- Refactoring: Improving the Design of Existing Code de Martin Fowler[53]
- Design patterns de Erich Gamma, John Vlissides, Richard Helm y Ralph Johnson[54]
- Effective Unit Testing de Lasse Koskela[55]
- The Art of Unit Testing: with examples in C# de Roy Osherove[56]
- JavaScript Allonge de Reg "raganwald" Braithwaite[57]
- You Don't Know JS de Kyle Simpson[58]
- Diseño Ágil con TDD de Carlos Blé[59]
- Testing y TDD para PHP de Fran Iglesias[60]
- Cursos de Codely.TV[61]

[47] https://amzn.to/2TUywwB
[48] https://amzn.to/2ph2wrZ
[49] https://amzn.to/2q5xgws
[50] https://amzn.to/2J1zWSH
[51] https://amzn.to/2VHQkNg
[52] https://amzn.to/2Hnh7cC
[53] https://amzn.to/2MGmeFy
[54] https://amzn.to/2EW7MXv
[55] https://amzn.to/2VCcsbP
[56] https://amzn.to/31ahpK6
[57] https://leanpub.com/javascript-allonge
[58] https://amzn.to/2OJ24xu
[59] https://www.carlosble.com/libro-tdd/?lang=es
[60] https://leanpub.com/testingytddparaphp
[61] https://codely.tv/pro/cursos

- Repositorio de Ryan McDermott[62]
- Guía de estilo de Airbnb[63]
- El artículo "From Stupid to SOLID code" de Willian Durand[64]
- Blog Koalite[65]
- El artículo de Software Crafters: "Programación Funcional en JavaScript" de Jose Manuel Lucas[66]
- Conversaciones con los colegas Carlos Blé[67], Dani García, Patrick Hertling[68] y Juan M. Gómez[69].

[62] https://github.com/ryanmcdermott/clean-code-javascript
[63] https://github.com/airbnb/javascript
[64] https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/
[65] http://blog.koalite.com/
[66] https://softwarecrafters.io/javascript/introduccion-programacion-funcional-javascript
[67] https://twitter.com/carlosble
[68] https://twitter.com/PatrickHertling
[69] https://twitter.com/_jmgomez_

DEMO

No dejes escapar la oportunidad, estoy convencido de que te puede aportar algún detalle de mucho valor.

Recuerda que si no es lo que esperas, te devolvemos tu dinero. Durante los primeros 15 días de compra, puedes obtener un reembolso del 100%. El riesgo es cero y el beneficio podría ser muy elevado.

Puedes adquirir el e-book desde aquí.[70]

[70] https://softwarecrafters.io/cleancode-solid-testing-js