Breve introducción

{dev/talles}

Este curso:

- No es para aprender a programar.
- No es para aprender JavaScript o TypeScript.

Breve introducción

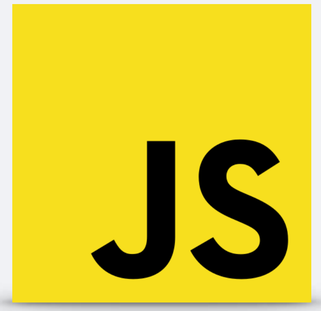
{dev/talles}

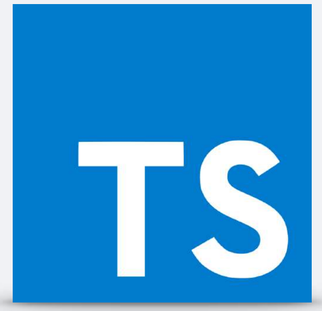
Este curso:

Es para practicar y aprender buenas prácticas para el manejo de nuestro código y reducir la deuda técnica.

Breve introducción

{dev/talles}





Lo que aprenderemos aquí nos servirá en cualquier lenguaje de programación

Deuda Técnica

¿Qué es?

La falta de calidad en el código, que genera una deuda que repercutirá en costos futuros.

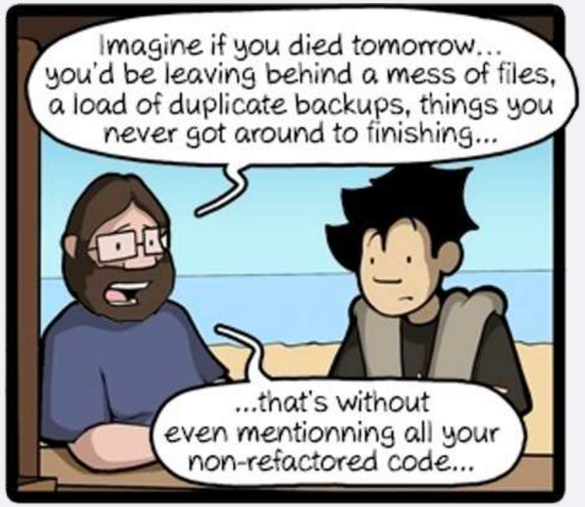
La falta de calidad en el código, que genera una deuda que repercutirá en costos futuros

Costos económicos

- Tiempo en realizar mantenimientos.
- Tiempo en refactorizar código.
- Tiempo en comprender el código.
- Tiempo adicional en la transferencia del código.


Do you ever think that someday you should, you know, spring clean? Tidy up all your personal files, check your backups, sort your photos properly, that kind of thing...
Yeah of course


When are you gonna do it?
I dunno, someday


Imagine if you died tomorrow... you'd be leaving behind a mess of files, a load of duplicate backups, things you never got around to finishing...
...that's without even mentionning all your non-refactored code...

CommitStrip.com

Esquema de deuda técnica de Martin Fowler

Imprudente

No hay tiempo, sólo copia y pega eso de nuevo

Prudente

Tenemos que entregar rápido, ya refactorizaremos

Inadvertido

“¿Qué son patrones de diseños?”

“Ahora sabemos cómo lo deberíamos haber hecho”

Deuda Técnica

Caer en deuda técnica es normal y a menudo es inevitable.

¿Cómo se paga una deuda técnica?

Refactorización

Es simplemente un proceso que tiene como objetivo mejorar el código sin alterar su comportamiento para que sea más entendible y tolerante a cambios.

Refactorización

Usualmente para que una refactorización fuerte tenga el objetivo esperado, es imprescindible contar con pruebas automáticas.

Usualmente esto produce el famoso: **“Si funciona, no lo toques”**

Deuda Técnica

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.

Clean Code

{dev/talles}

“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

Clean Code

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

Clean Code

“Programar es el arte de decirle a otro humano lo que quieres que la computadora haga”. – Donald Knuth

A continuación iniciaremos nuestro camino para escribir código limpio

// Mejor

```
const numberOfUnits = 53;

const tax = 0.15;

const category = 'T-Shirts';

const birthDate = new Date('August 15, 1965 00:00:00');
```

// Mal

```
const n = 53;

const tx = 0.15;

const cat = 'T-Shirts';

const ddmmyyyy = new Date('August 15, 1965 00:00:00');
```

Nombres pronunciables y expresivos

```
// Mal
class AbstractUser { };
class UserMixin { };
class UserImplementation { };
interface UserInterface { };
```

```
// Mejor
class User { };
interface User { };
```

Ausencia de información técnica en nombres

Arreglos - Arrays


```
// malo
const fruit= ['manzana','platano','fresa'];
```

```
// regular
const fruitList=['manzana','platano','fresa'];
```

```
// bueno
const fruits=['manzana','platano','fresa'];
```

```
// mejor
const fruitNames=['manzana','platano','fresa'];
```

Nombres según el tipo de dato

Booleanos - Booleans


```
//mal
const open = true;
const write = true;
const fruit = true;
const active = false;
const noValues = true;
const notEmpty = true;
```

```
//mejor
const isOpen = true;
const canWrite = true;
const hasFruit = true;
const isActive = false;
const hasValues = false;
const isEmpty = false;
```

números

```
//bad
const fruits = 3;
const cars = 10;
```

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

const totalOfCars = 5;
```

Nombres según el tipo de dato

Funciones

```
//mal
createUserIfNotExists();
updateUserIfNotEmpty();
sendEmailIfFieldsValid();
```

```
//mejor
createUser();
updateUser();
sendEmail();
```

Nombres según el tipo de dato

Clases

- El nombre es lo más importante de la clase.
- Formados por un sustantivo o frases de sustantivo.
- No deben de ser muy genéricos.
- Usar UpperCamelCase

```
// Malos
class Manager {};
class Data {};
class Info {}
class Individual{};
class Processor{};
class SpecialMonsterView {};
```

Clases

3 preguntas para determinar saber si es un buen nombre.

- ¿Qué exactamente hace la clase?
- ¿Cómo exactamente esta clase realiza cierta tarea?
- ¿Hay algo específico sobre su ubicación?

Si algo no tiene sentido, remuévelo o refactoriza.

Clases

Más palabra !== mejor nombre

```
class SpecialViewingCaseMonsterManagerEventsHandlerActivitySingleton {};
```

Funciones

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

– Ward Cunningham

Funciones


```ts
function sendEmail( toWhom: string ): boolean {

  // Verificar correo
  if ( !toWhom.includes('@') ) return false;

  // Construir el cuerpo o mensaje

  // Enviar correo

  // Si todo sale bien
  return true;
}
```

```ts
function sendEmail(): boolean {

  // Verificar si el usuario existe

  // Revisar contraseña

  // Crear usuario en Base de datos

  // Si todo sale bien
  return true;
}
```

Parámetros y argumentos

Parámetros

```ts
function sendEmail( toWhom: string ): boolean {

  // Verificar correo
  if ( !toWhom.includes('@') ) return false;

  // Construir el cuerpo o mensaje

  // Enviar correo

  // Si todo sale bien
  return true;
}
```

Argumentos

```ts
sendEmail('fernando@google.com');
```

Parámetros y argumentos


```
// Bien
function sendEmail( toWhom: string, from: string, body: string ): boolean {
```

```
// No muy bien
function sendEmail( toWhom: string, from: string, body: string, subject: string,apikey: string ): boolean {
```

Limitar a 3 parámetros posicionales

```
// No muy bien
function sendEmail(
  toWhom : string,
  from   : string,
  body   : string,
  subject: string,
  apikey : string,
): boolean {
```

Parámetros y argumentos


```ts
interface SendEmailOptions {
  toWhom : string;
  from   : string;
  body   : string;
  subject: string;
  apikey : string;
}
```

```ts
// Mejor
function sendEmail({ toWhom, from, body, subject, apikey }: SendEmailOptions ): boolean {
```

```ts
// No muy bien
function sendEmail( toWhom: string, from: string, body: string, subject: string,apikey: string ): boolean {
```

Funciones

Otras recomendaciones

- Simplicidad es fundamental.
- Funciones de tamaño reducido.
- Funciones de una sola línea sin causar complejidad.
- Menos de 20 líneas.
- Evita el uso del “else”.
- Prioriza el uso de la condicional ternaria.

Principio DRY

Don’t Repeat Yourself

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

– Robert C. Martin

Principio DRY

Don’t Repeat Yourself

- Simplemente es evitar tener duplicidad de código.
- Simplifica las pruebas.
- Ayuda a centralizar procesos.
- Aplicar el principio DRY, usualmente lleva a refactorizar.

Estructura de clases

“El buen código parece estar escrito por alguien a quien le importa”.

– Michael Feathers

```
class HtmlElement {

    public static domReady: boolean = false;

    private _id: string;
    private type: string;
    private updatedAt: number;

    static createInput( id: string ) {
        return new HtmlElement(id, 'input');
    }

    constructor( id: string, type: string ) {
        this._id = id;
        this.type = type;
        this.updatedAt = Date.now();
    }

    setType( type: string ) {
        this.type = type;
        this.updatedAt = Date.now();
    }

    get id(): string {
        return this.id;
    }
}
```

Comenzar con lista de propiedades.

1. Propiedades estáticas.
2. Propiedades públicas de último.

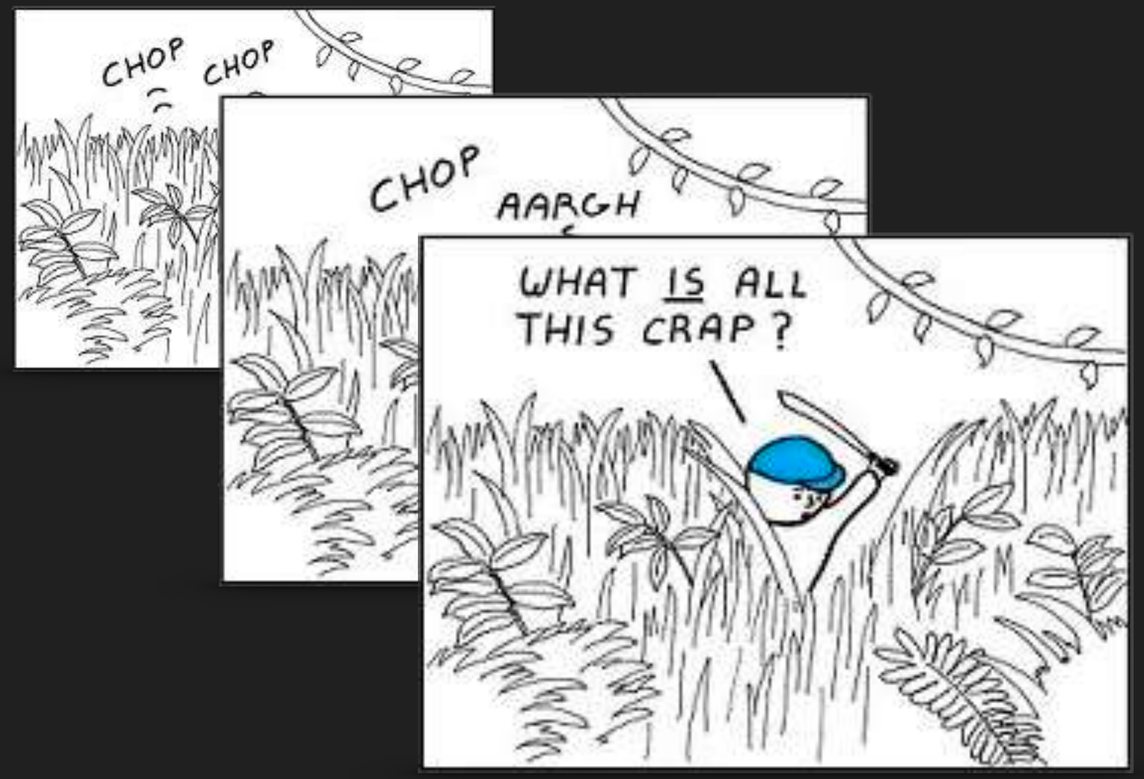
Métodos

1. Empezando por los constructores estáticos.
2. Luego el constructor.
3. Seguidamente métodos estáticos.
4. Métodos privados después.
5. Resto de métodos de instancia ordenados de mayor a menor importancia.
6. Getters y Setters al final.

Comentarios

Muchos de los comentarios que verás en el código lucen así

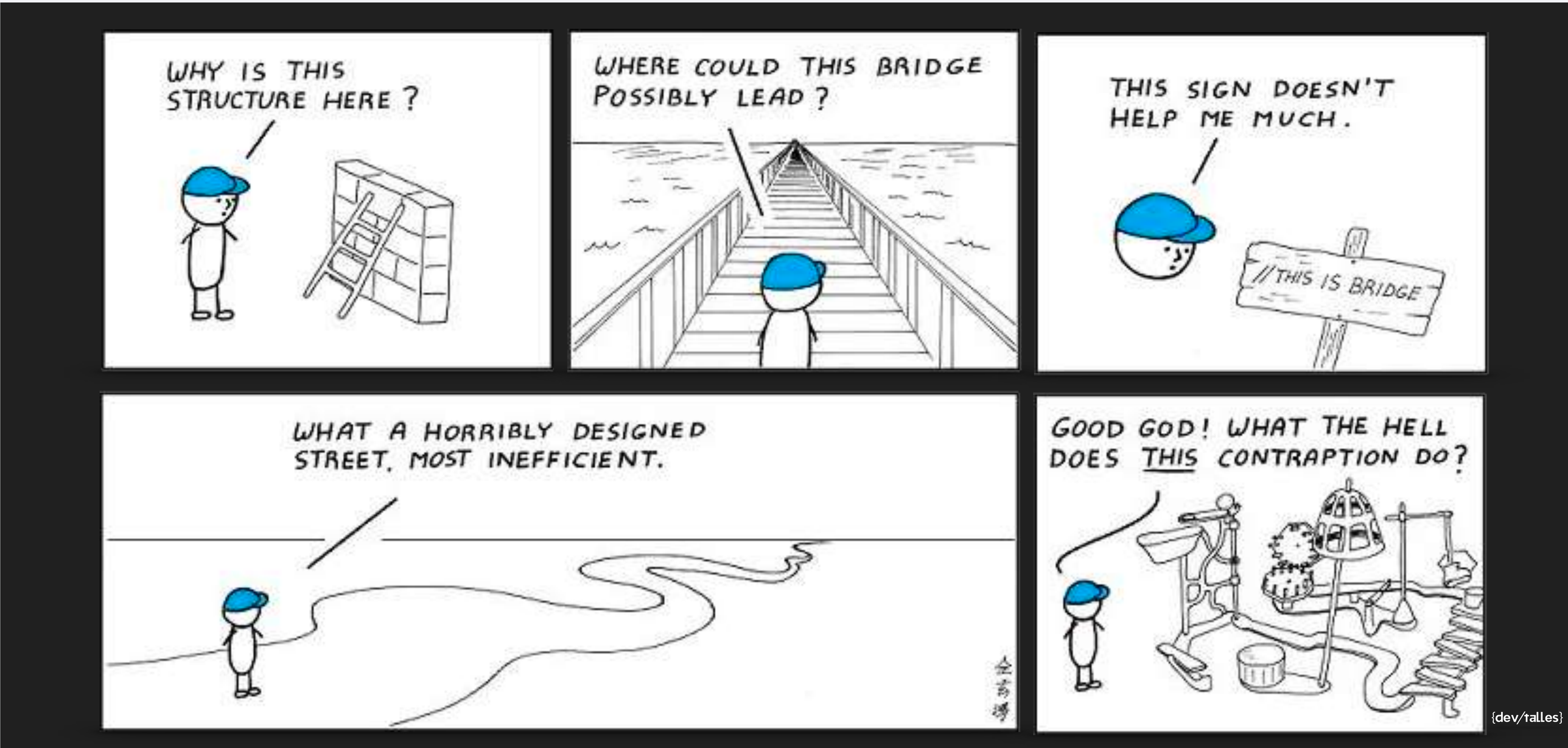
Comentarios



{dev/talles}

Comentarios

Es horrible leer código de otros



Comentarios

```
const name = 'John Doe';

// Si name es igual a 'John Doe'
if ( name === 'John Doe' ) {
    // entonces....
}
```

Comentarios

Evita usar comentarios, pero…

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

Los comentarios deberían de ser la excepción, no la regla.

Comentarios

“No comentes el código mal escrito, reescríbelo”.

– Brian W. Kernighan

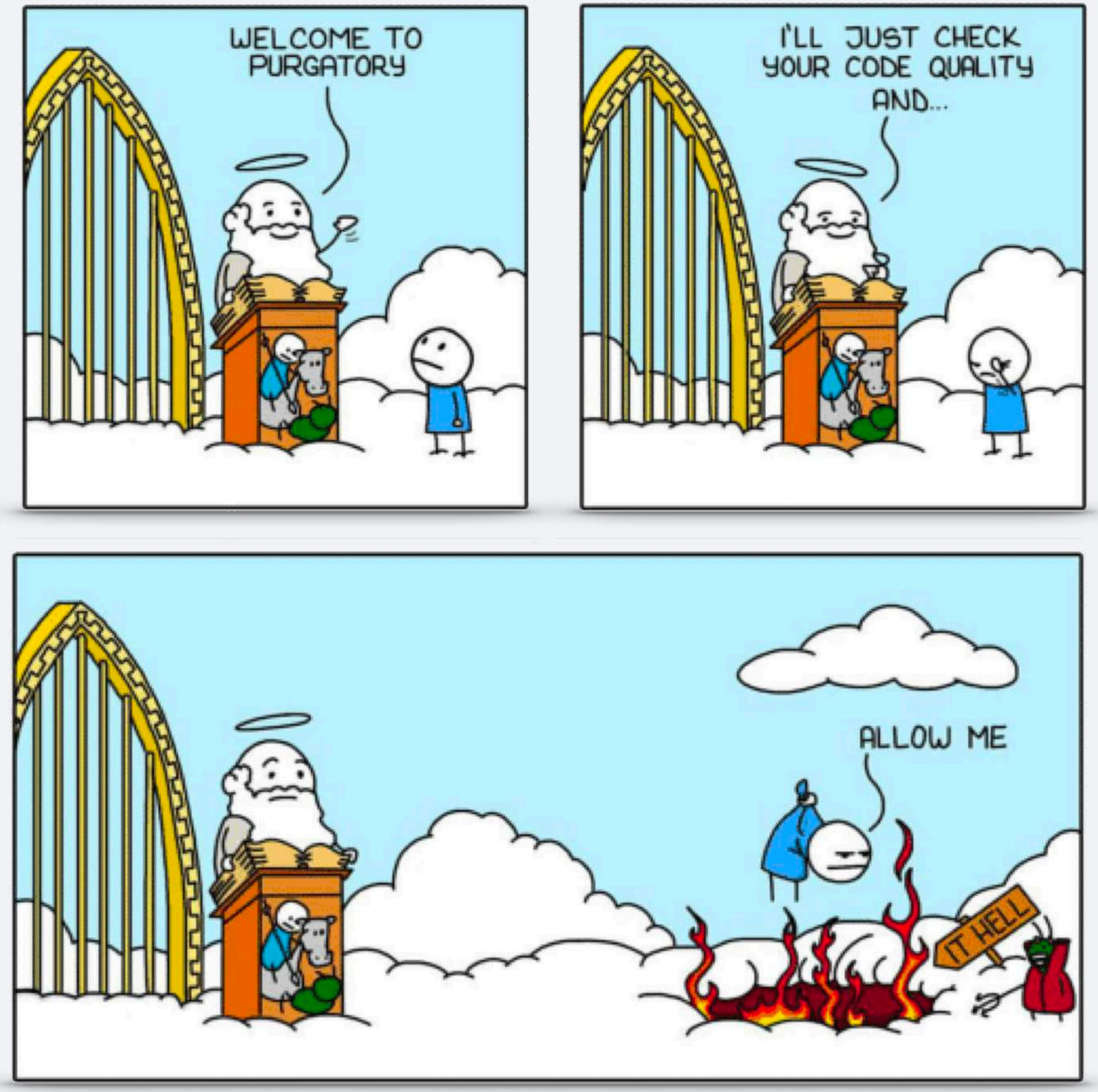
Comentarios

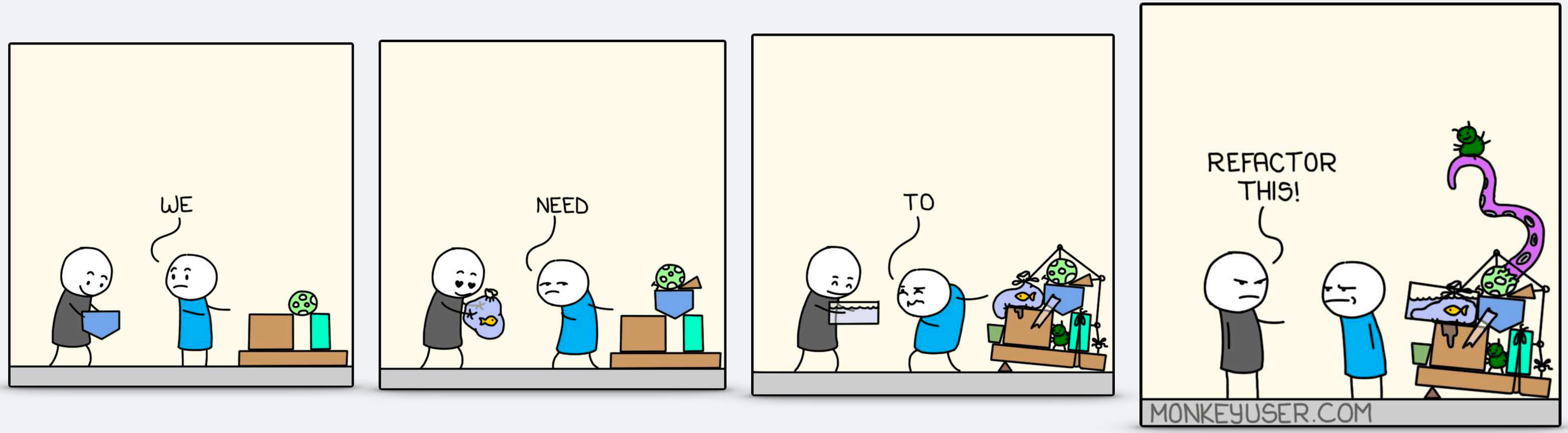
Recuerda:
Nuestro código debe de ser suficientemente auto explicativo.

Pero a veces es necesario comentarlo

¿El por qué?
En lugar del ¿qué? o ¿cómo?

{dev/talles}




WE
NEED
TO
REFACTOR THIS!
MONKEYUSER.COM

Uniformidad en el proyecto

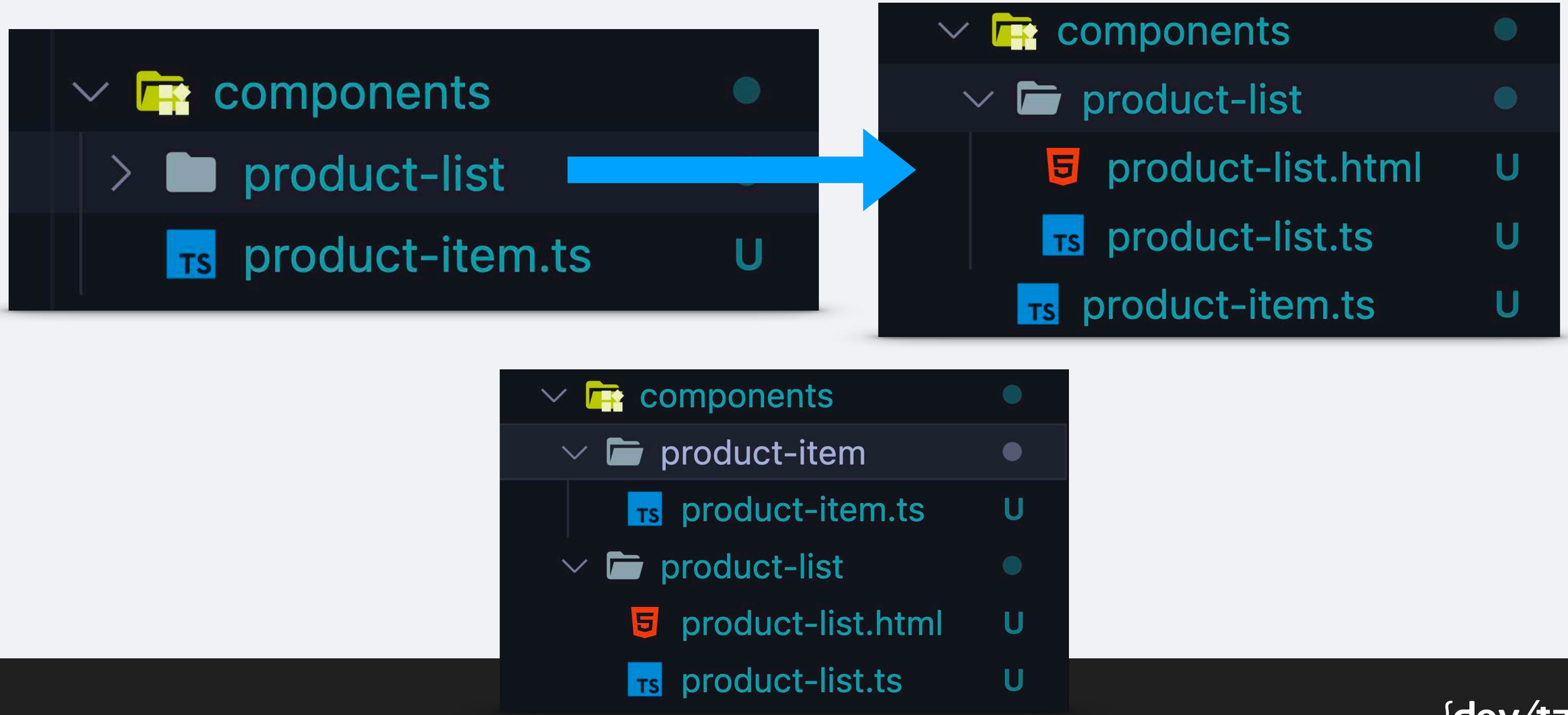
Problemas similares, soluciones similares

```
const createProduct = () => {

}

const updateProduct = () => {

}

const deleteProduct = () => {

}
```

```
const createUser = () => {

}

const updateUser = () => {

}

const deleteUser = () => {

}
```

Uniformidad en el proyecto

Problemas similares, soluciones similares



Uniformidad en el proyecto

Indentación

```
class UserSettings extends User {
constructor(
    public workingDirectory: string,
    public lastFolderOpen: string,
    email     : string,
    role      : string,
    name      : string,
    gender    : Gender,
    birthdate : Date,
){
super(
    email,
    role,
    new Date(),
    name,
    gender,
    birthdate
)
}
}
```

```
class UserSettings extends User {
    constructor(
        public workingDirectory: string,
        public lastFolderOpen: string,
        email     : string,
        role      : string,
        name      : string,
        gender    : Gender,
        birthdate : Date,
    ){
        super(
            email,
            role,
            new Date(),
            name,
            gender,
            birthdate
        )
    }
}
```

Acrónimo STUPID

6 Code Smells que debemos de evitar.

- **S**ingleton: patrón singleton.
- **T**ight Coupling: alto acoplamiento.
- **U**ntestability: código no probable (unit test).
- **P**remature optimization: optimizaciones prematuras.
- **I**ndescriptive Naming: nombres poco descriptivos.
- **D**uplication: duplicidad de código, no aplicar el principio DRY.

Patrón Singleton
Pros

Garantiza una única instancia de la clase a lo largo de toda la aplicación.

¿Por qué code smell?

Cons

- Vive en el contexto global.
- Puede ser modificado por cualquiera y en cualquier momento.
- No es rastreable.
- Difícil de testar debido a su ubicación.

Patrón Singleton

Ejemplo

Acrónimo STUPID

6 Code Smells que debemos de evitar.

- **S**ingleton: patrón singleton.
- **T**ight Coupling: alto acoplamiento.
- **U**ntestability: código no probable (unit test).
- **P**remature optimization: optimizaciones prematuras.
- **I**ndescriptive Naming: nombres poco descriptivos.
- **D**uplication: duplicidad de código, no aplicar el principio DRY.

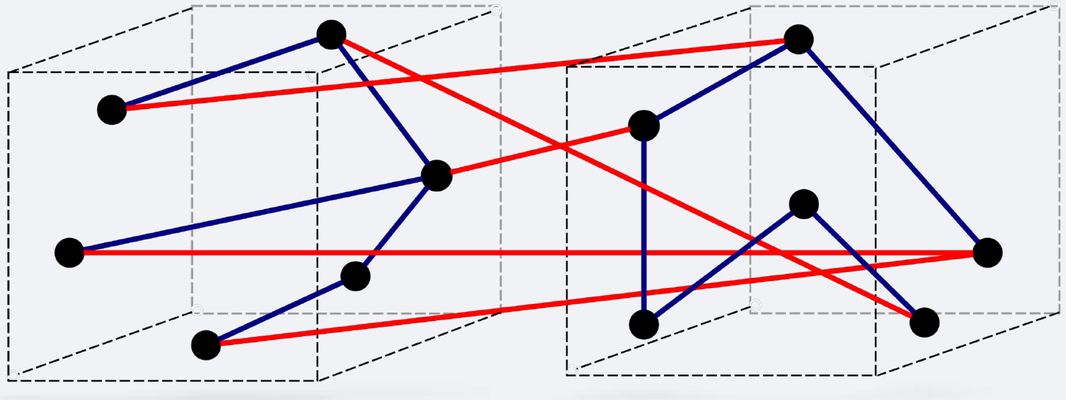
Alto acoplamiento

Lo ideal es tener bajo acoplamiento y buena cohesión.

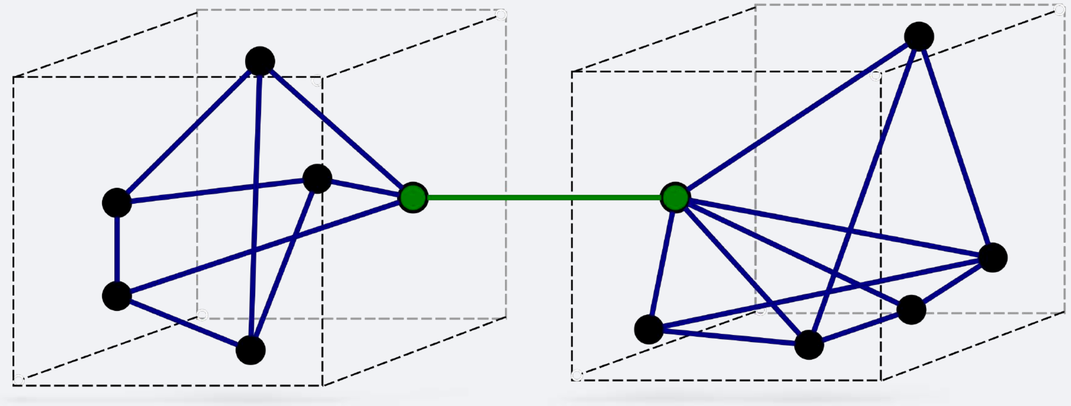
¿Pero qué significa eso?

Alto acoplamiento

Lo ideal es tener bajo acoplamiento y buena cohesión.



Algo acoplamiento y baja cohesión



Bajo acoplamiento y alta cohesión

Alto acoplamiento

Desventajas

- Un cambio en un módulo por lo general provoca un efecto dominó de los cambios en otros módulos.
- El ensamblaje de módulos puede requerir más esfuerzo y/o tiempo debido a la mayor dependencia entre módulos.
- Un módulo en particular puede ser más difícil de reutilizar y/o probar porque se deben incluir módulos dependientes.

Alto acoplamiento

Posibles soluciones

- **“A”** tiene un atributo que se refiere a **“B”**.
- **“A”** llama a los servicios de un objeto **“B”**.
- **“A”** tiene un método que hace referencia a **“B”** (a través del tipo de retorno o parámetro).
- **“A”** es una subclase de (o implementa) la clase **“B”**.

“Queremos diseñar componentes que sean auto-contenidos, auto suficientes e independientes. Con un objetivo y un propósito bien definido.”

— The Pragmatic Programmer

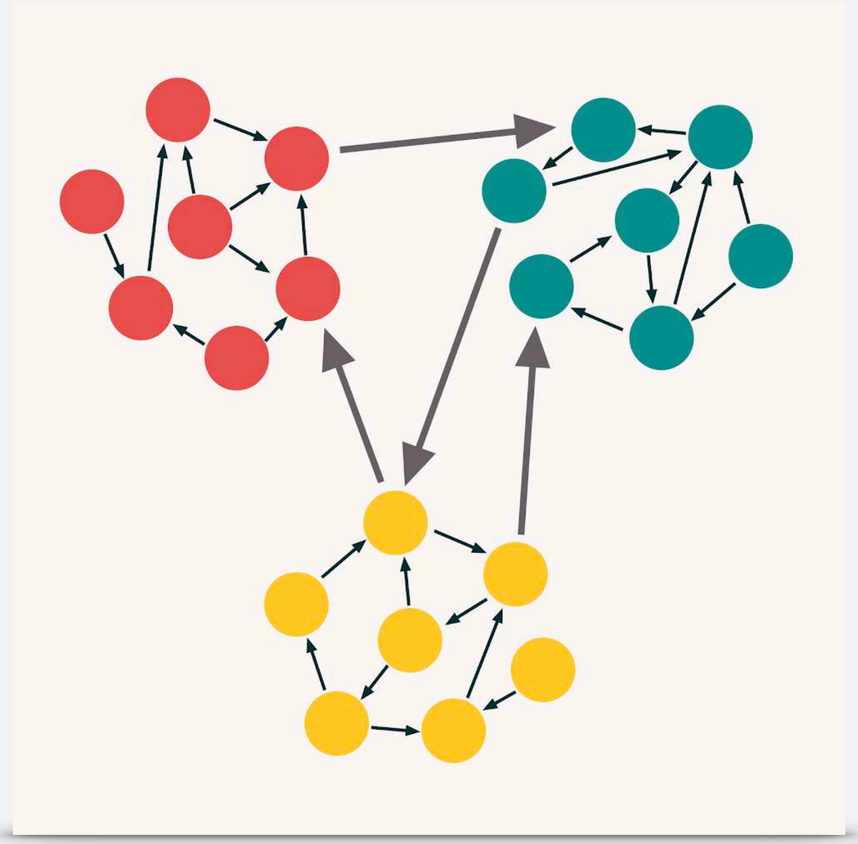
Cohesión

Lo ideal es tener bajo acoplamiento y buena cohesión.

- **La cohesión se refiere a lo que la clase (o módulo) puede hacer.**
- La baja cohesión significaría que la clase realiza una gran variedad de acciones: es amplia, no se enfoca en lo que debe hacer.
- Alta cohesión significa que la clase se enfoca en lo que debería estar haciendo, es decir, solo métodos relacionados con la intención de la clase.

Ideal

Clases A, B y C



Reference: enterprisecraftsmanship.com/posts/cohesion-coupling-difference

Acoplamiento

Lo ideal es tener bajo acoplamiento y buena cohesión.

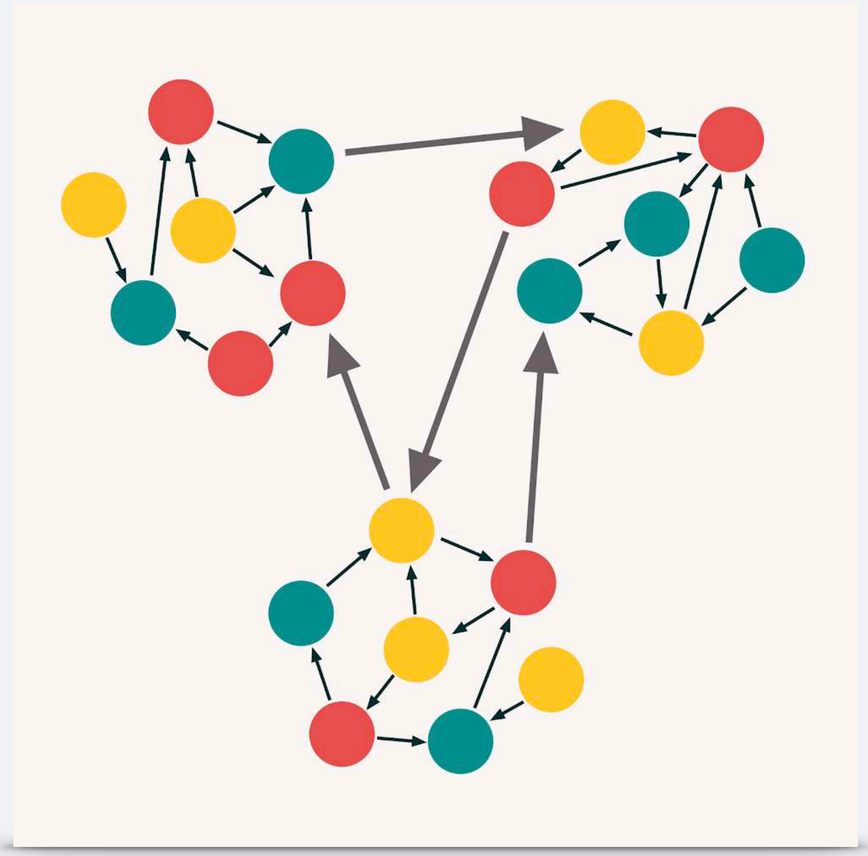
Se refiere a cuán relacionadas o dependientes son dos clases o módulos entre sí.

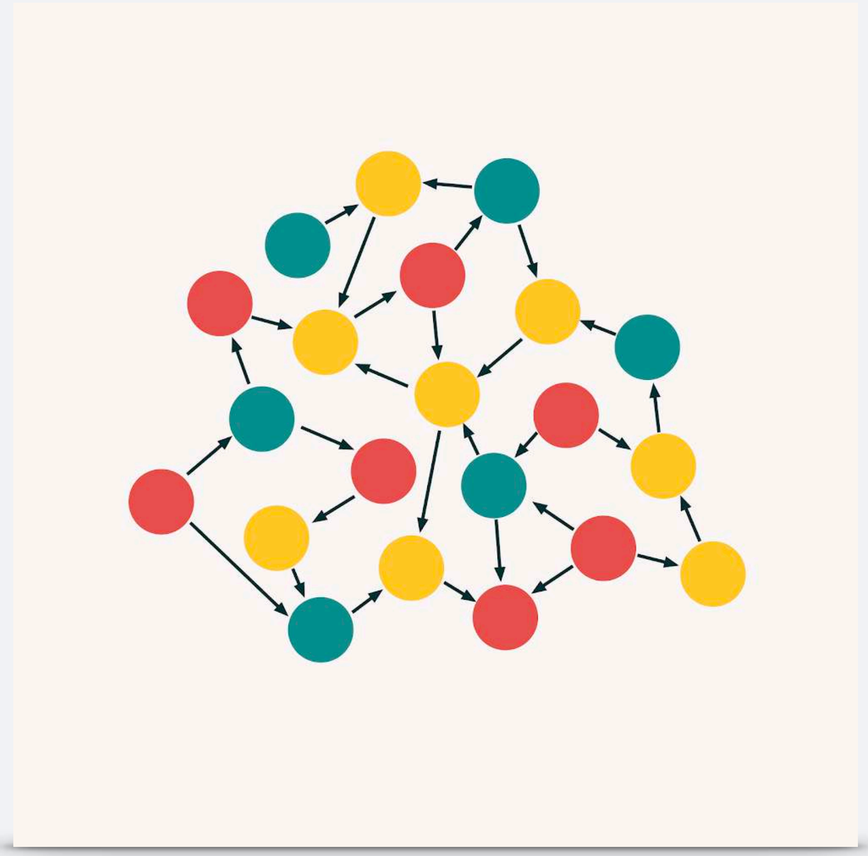
- **En bajo acoplamiento**, cambiar algo importante en una clase no debería afectar a la otra.

- **En alto acoplamiento,** dificultaría el cambio y el mantenimiento de su código; dado que las clases están muy unidas, hacer un cambio podría requerir una renovación completa del sistema.

Un buen diseño de software tiene alta cohesión y bajo acoplamiento.

Evitar





Reference: enterprisecraftsmanship.com/posts/cohesion-coupling-difference

Ejemplo

Acrónimo STUPID

6 Code Smells que debemos de evitar.

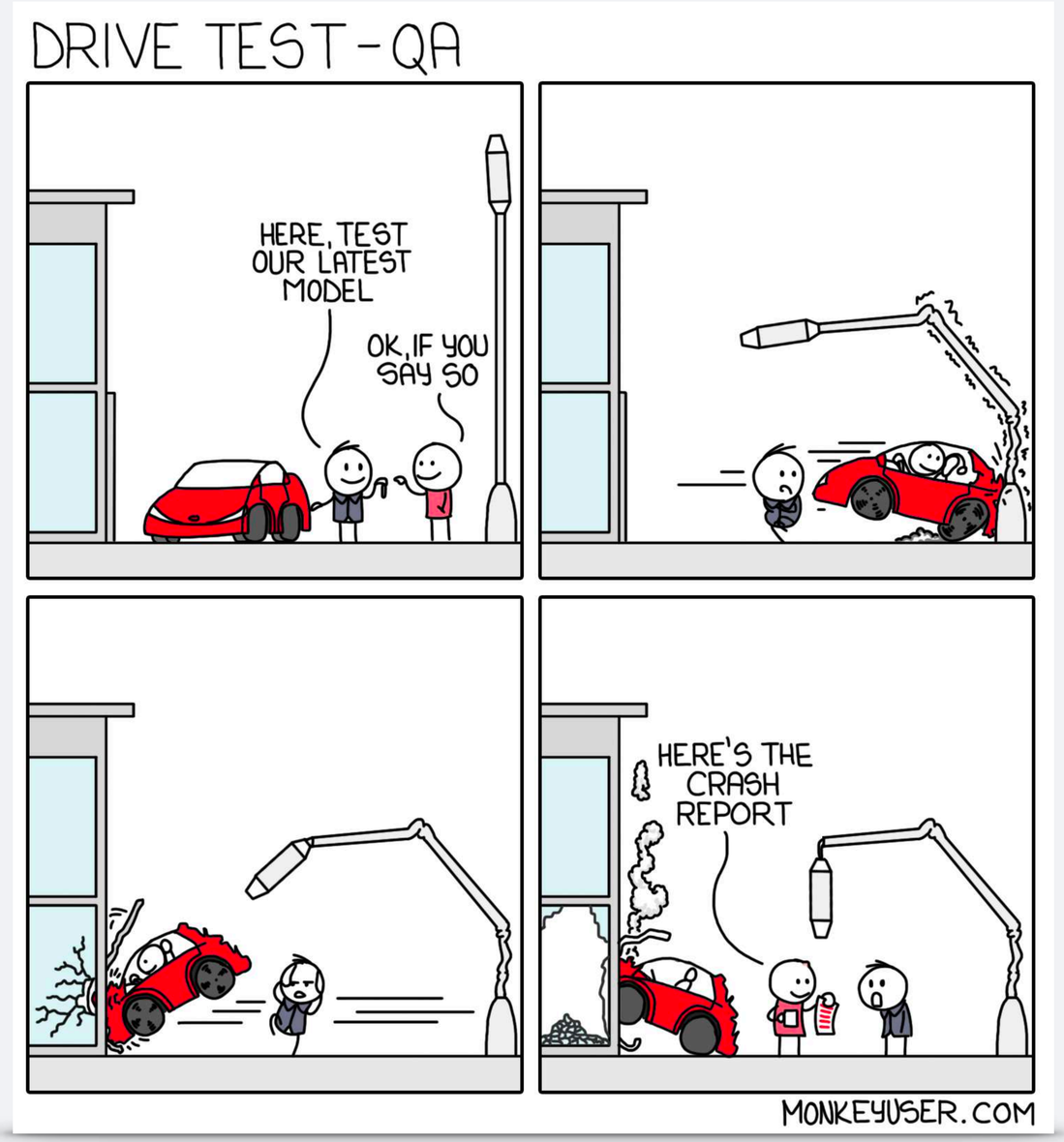
- **S**ingleton: patrón singleton.
- **T**ight Coupling: alto acoplamiento.
- **U**ntestability: código no probable (unit test).
- **P**remature optimization: optimizaciones prematuras.
- **I**ndescriptive Naming: nombres poco descriptivos.
- **D**uplication: duplicidad de código, no aplicar el principio DRY.

Código no probable

Código dificilmente testeable

- Código con alto acoplamiento.
- Código con muchas dependencias no inyectadas.
- Dependencias en el contexto global (Tipo Singleton)

Debemos de tener en mente las pruebas desde la creación del código.


DRIVE TEST - QA
HERE, TEST OUR LATEST MODEL
OK, IF YOU SAY SO
HERE'S THE CRASH REPORT
MONKEYUSER.COM

Optimizaciones prematuras

Mantener abiertas las opciones retrasando la toma de decisiones nos permite darle mayor relevancia a lo que es más importante en una aplicación.

No debemos anticiparnos a los requisitos y desarrollar abstracciones innecesarias que puedan añadir complejidad accidental.

Complejidad esencial

La complejidad es inherente al problema.

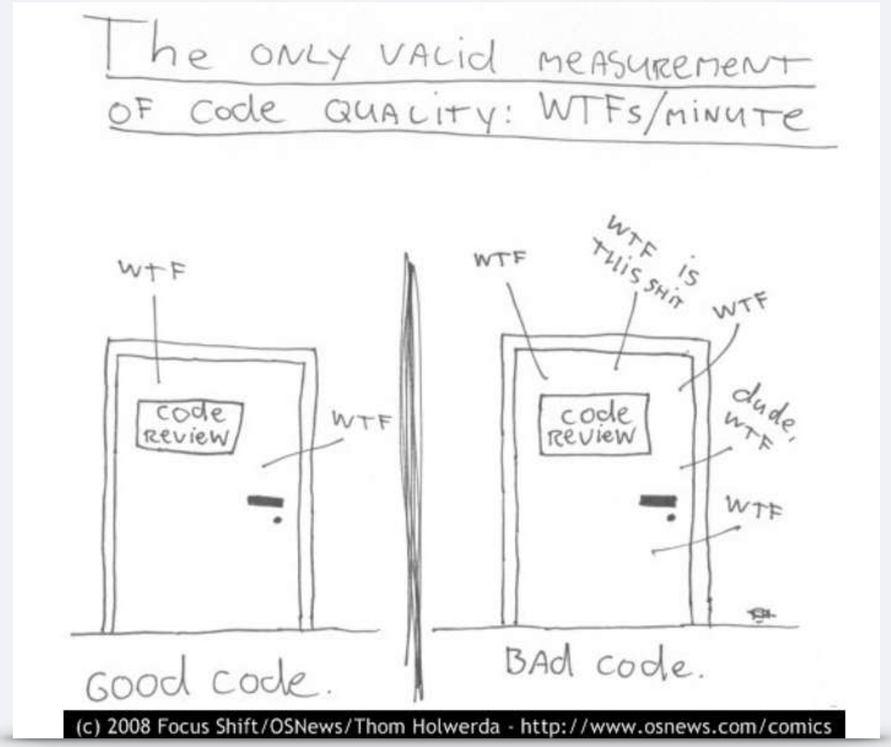
Cuando implementamos una solución compleja a la mínima indispensable.

Complejidad accidental

Nombres poco descriptivos

{dev/talles}

- Nombres de variables mal nombradas.
- Nombres de clases genéricas.
- Nombres de funciones mal nombradas.
- Ser muy especifico o demasiado genérico.



La única medida de calidad de código es el “WTF” por minuto

Duplicidad de Código

No aplicar el principio DRY

Accidental

- Código luce similar pero cumple funciones distintas.
- Cuando hay un cambio, sólo hay que modificar un sólo lugar.
- Este tipo de duplicidad se puede trabajar con parámetros u optimizaciones.

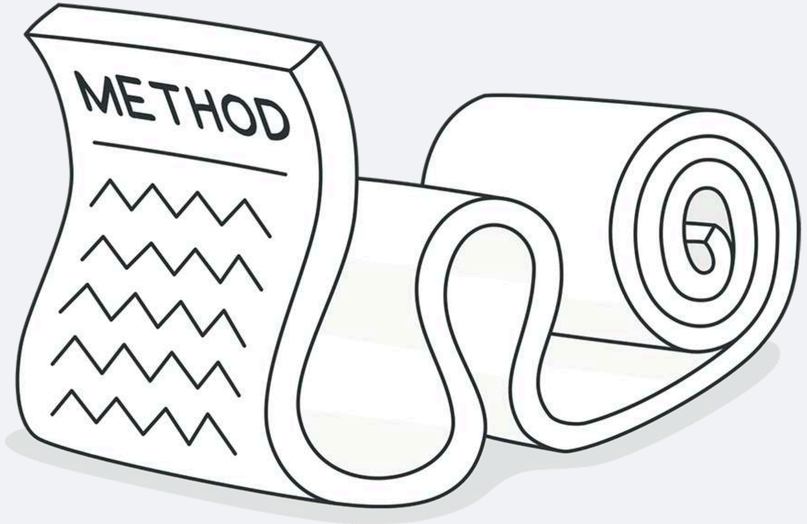
Real

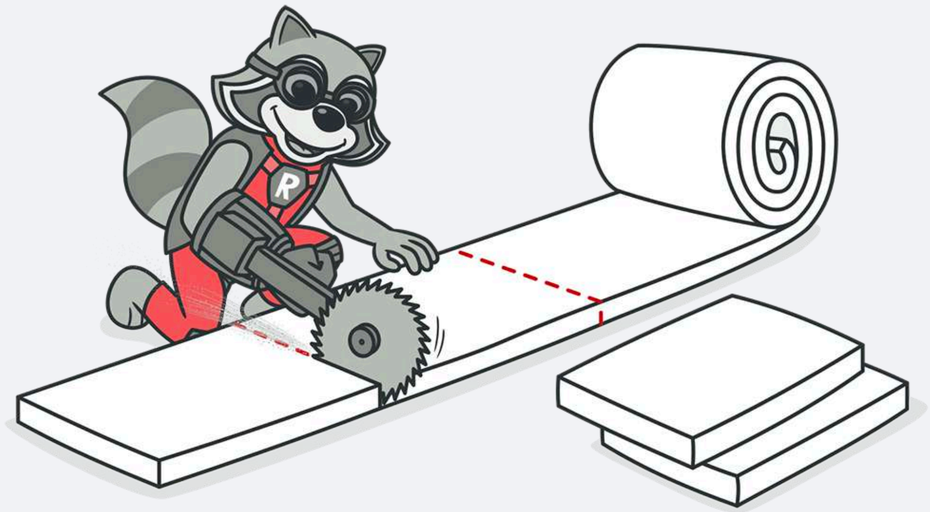
- Código es idéntico y cumple la misma función.
- Un cambio implicaría actualizar todo el código idéntico en varios lugares.
- Incrementa posibilidades de error humano al olvidar una parte para actualizar.
- Mayor cantidad de pruebas innecesarias.

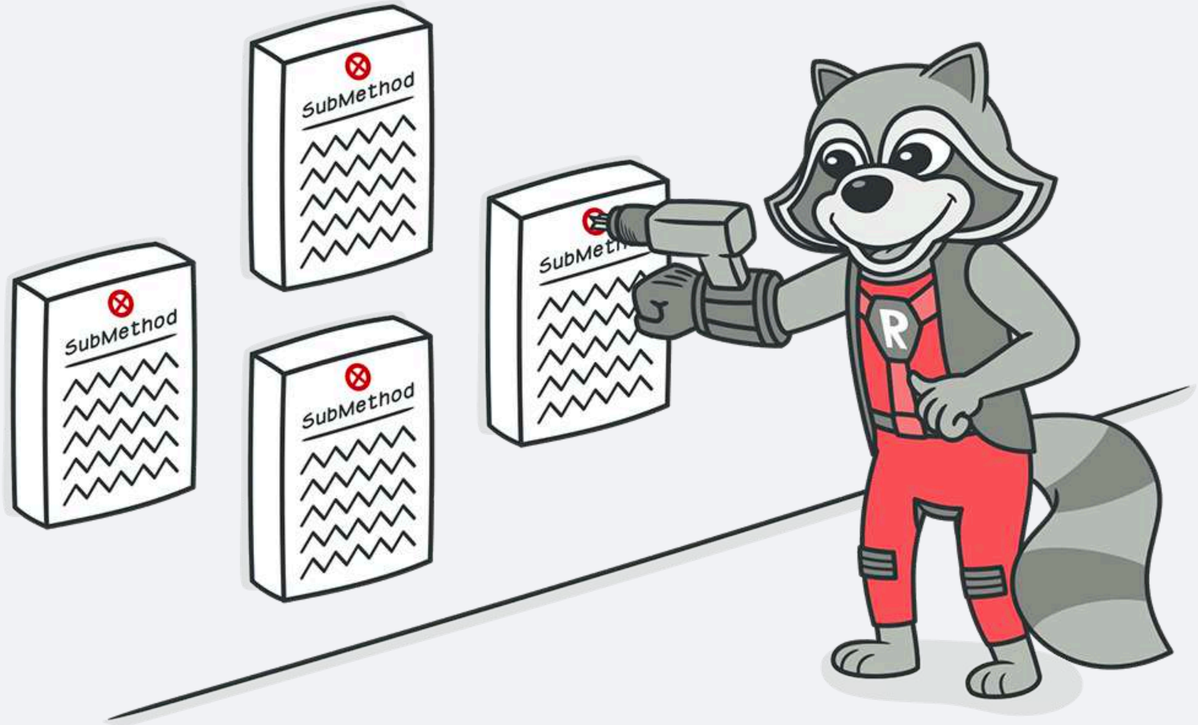
Otros “code smells”

Que merecen tener su mención especial

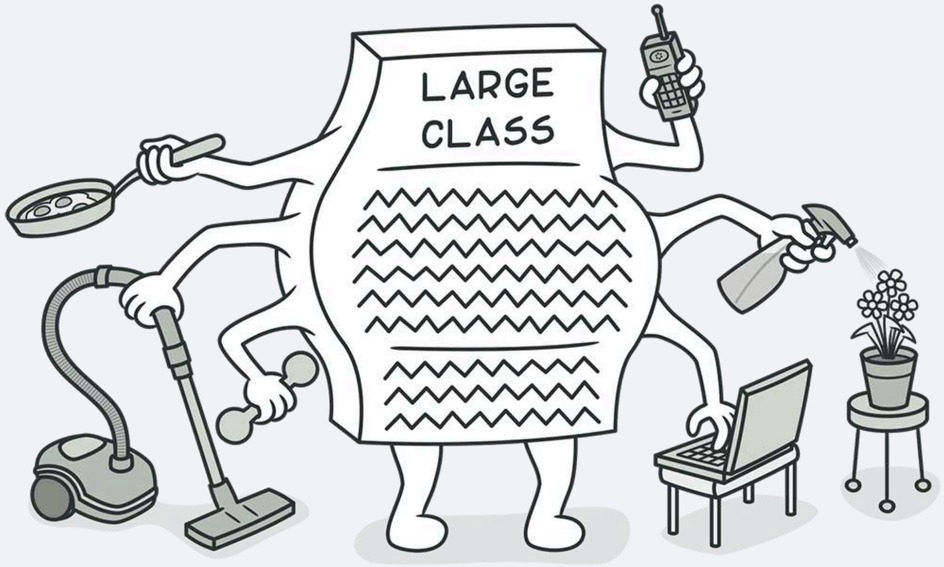
Refactoring.Guru

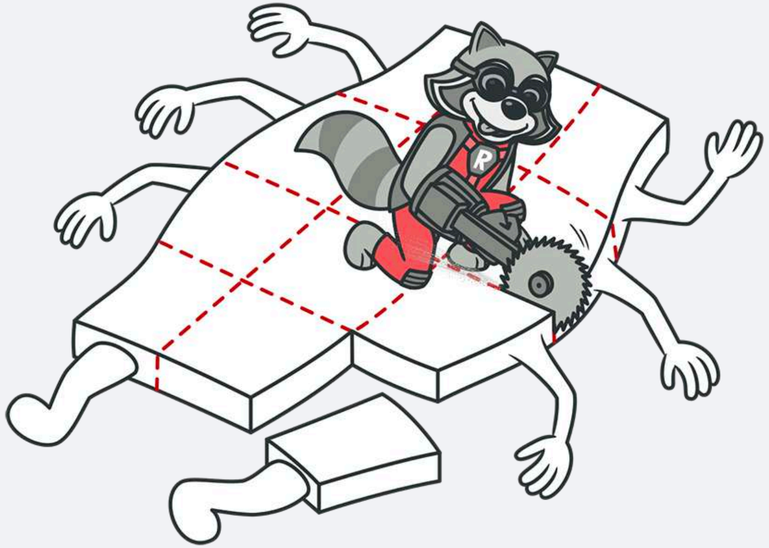


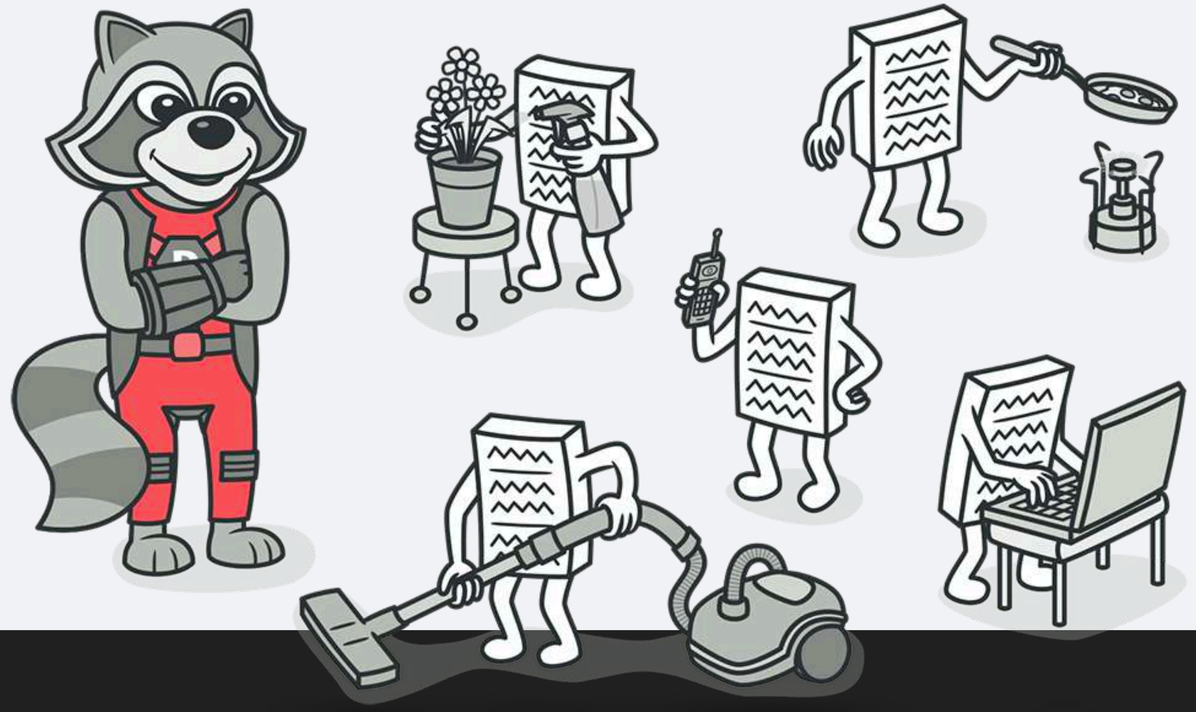




Inflación

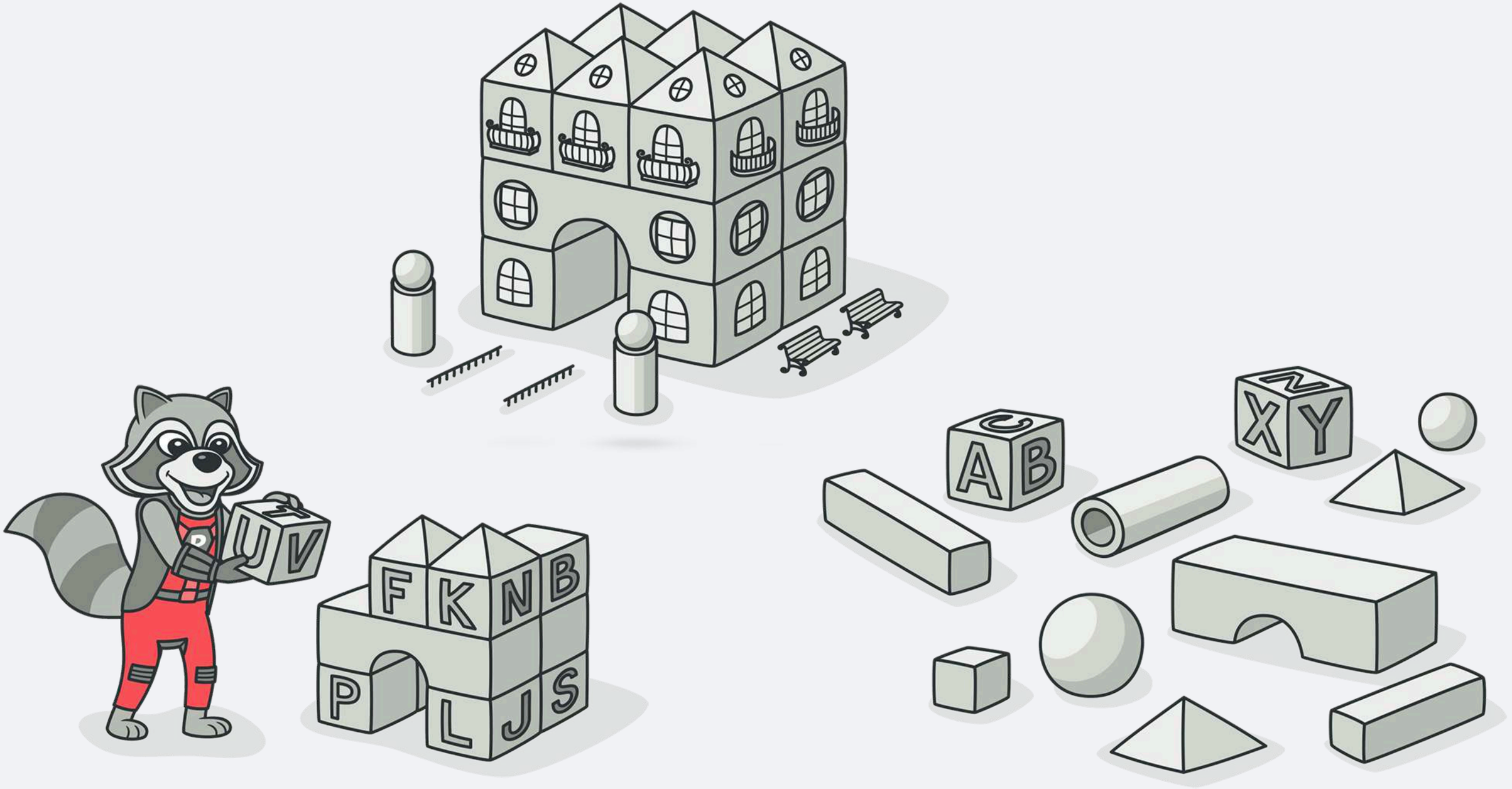






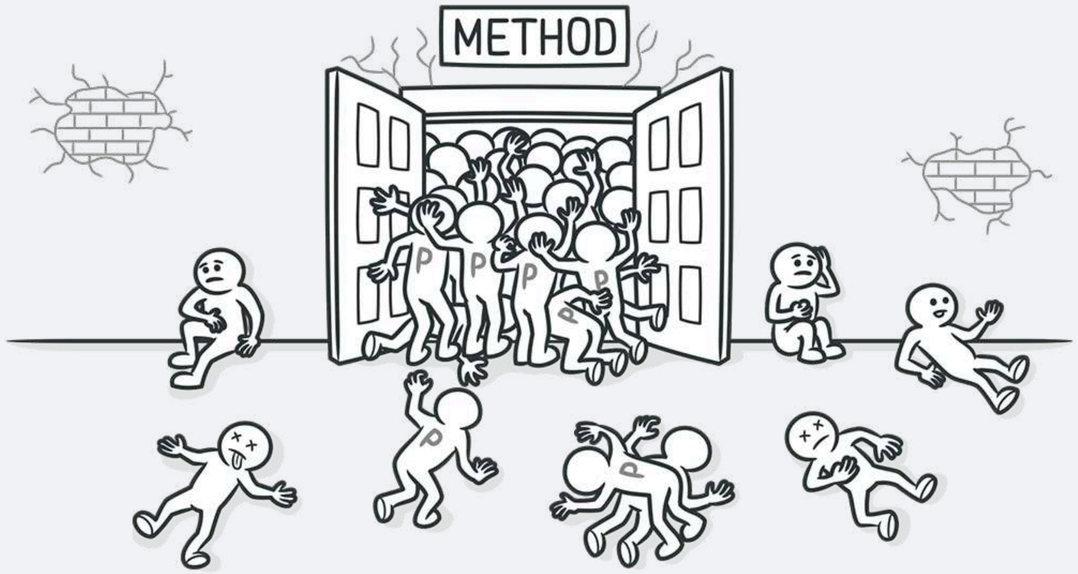
{dev/talles}

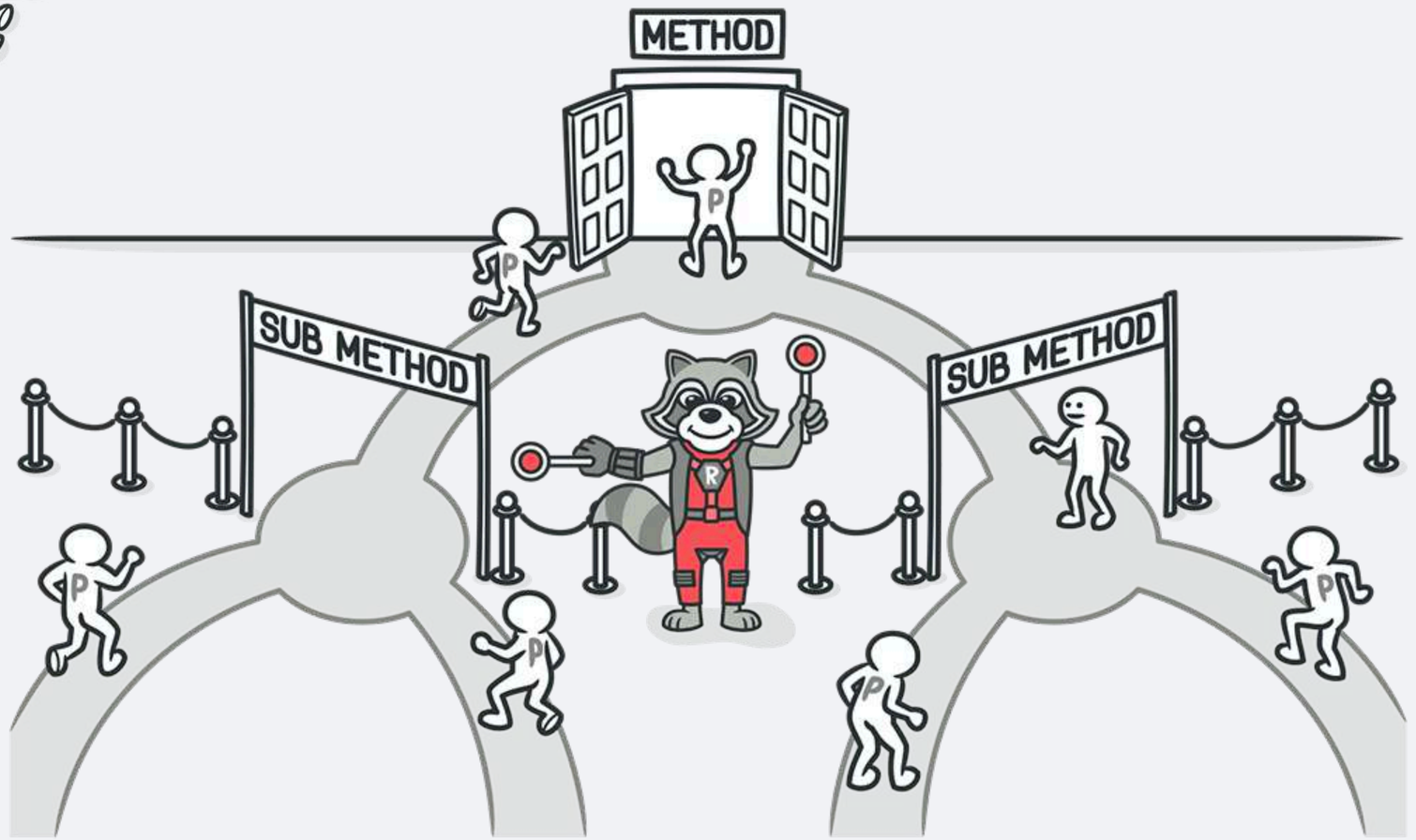
Inflación



Obsesión primitiva

Inflación

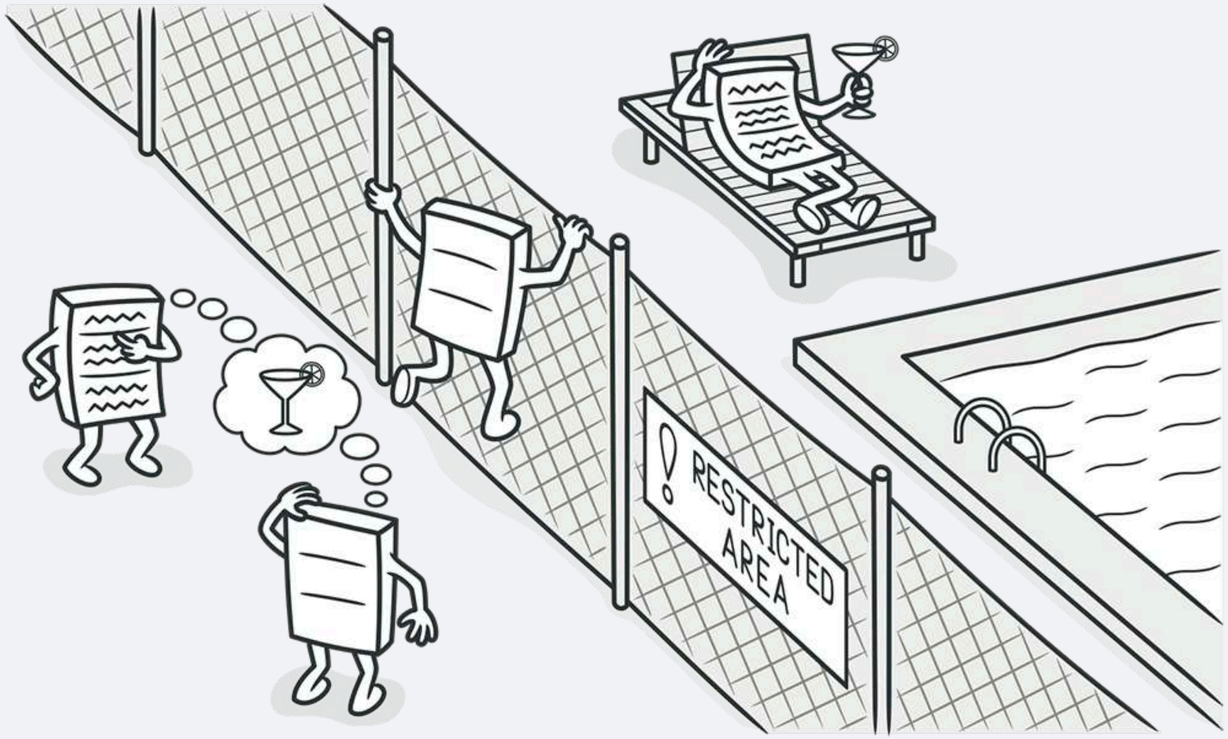




Lista larga de parámetros

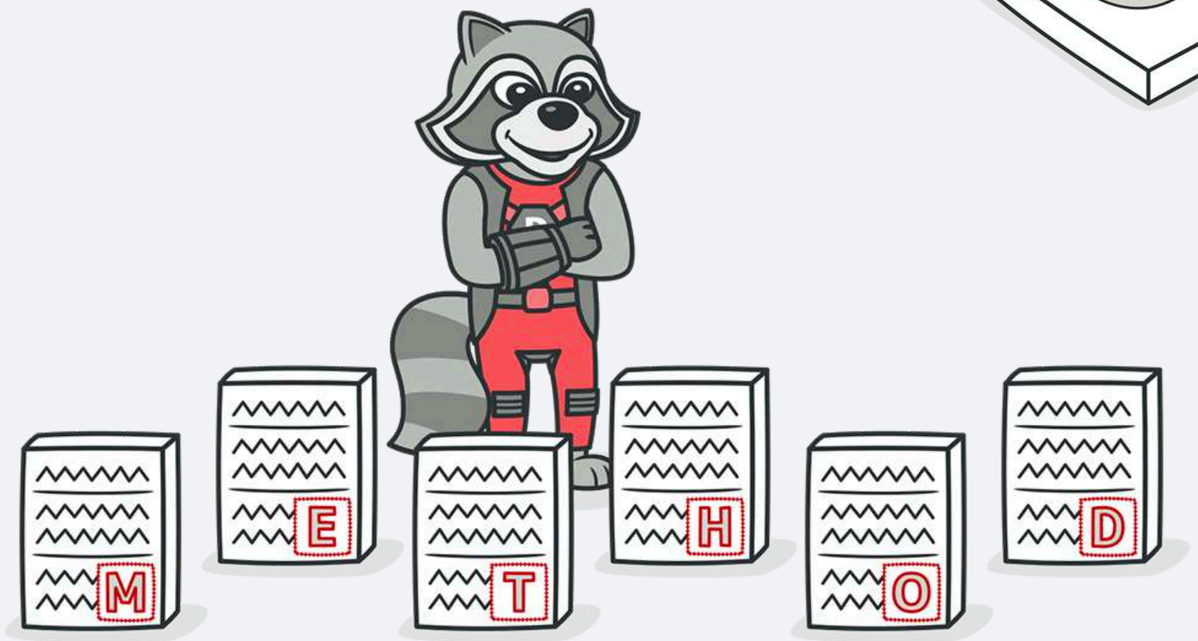
Inflación

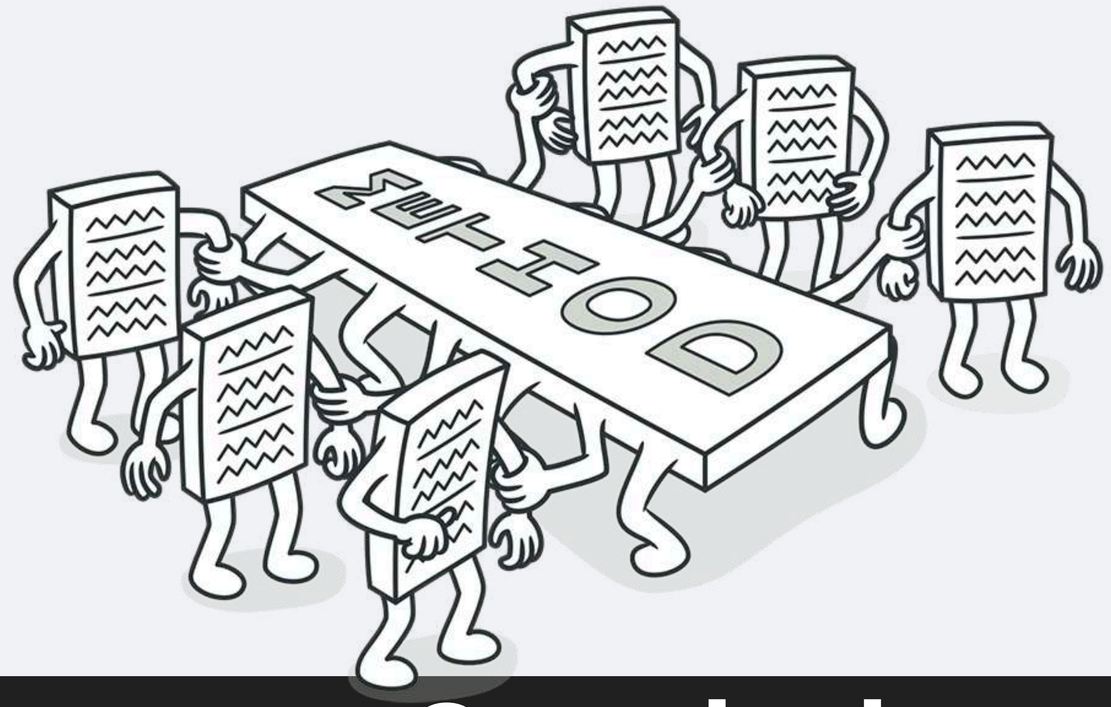
Todos los olores de este grupo contribuyen al acoplamiento excesivo entre clases o muestran lo que sucede si el acoplamiento se reemplaza por una delegación excesiva.



Feature Envy

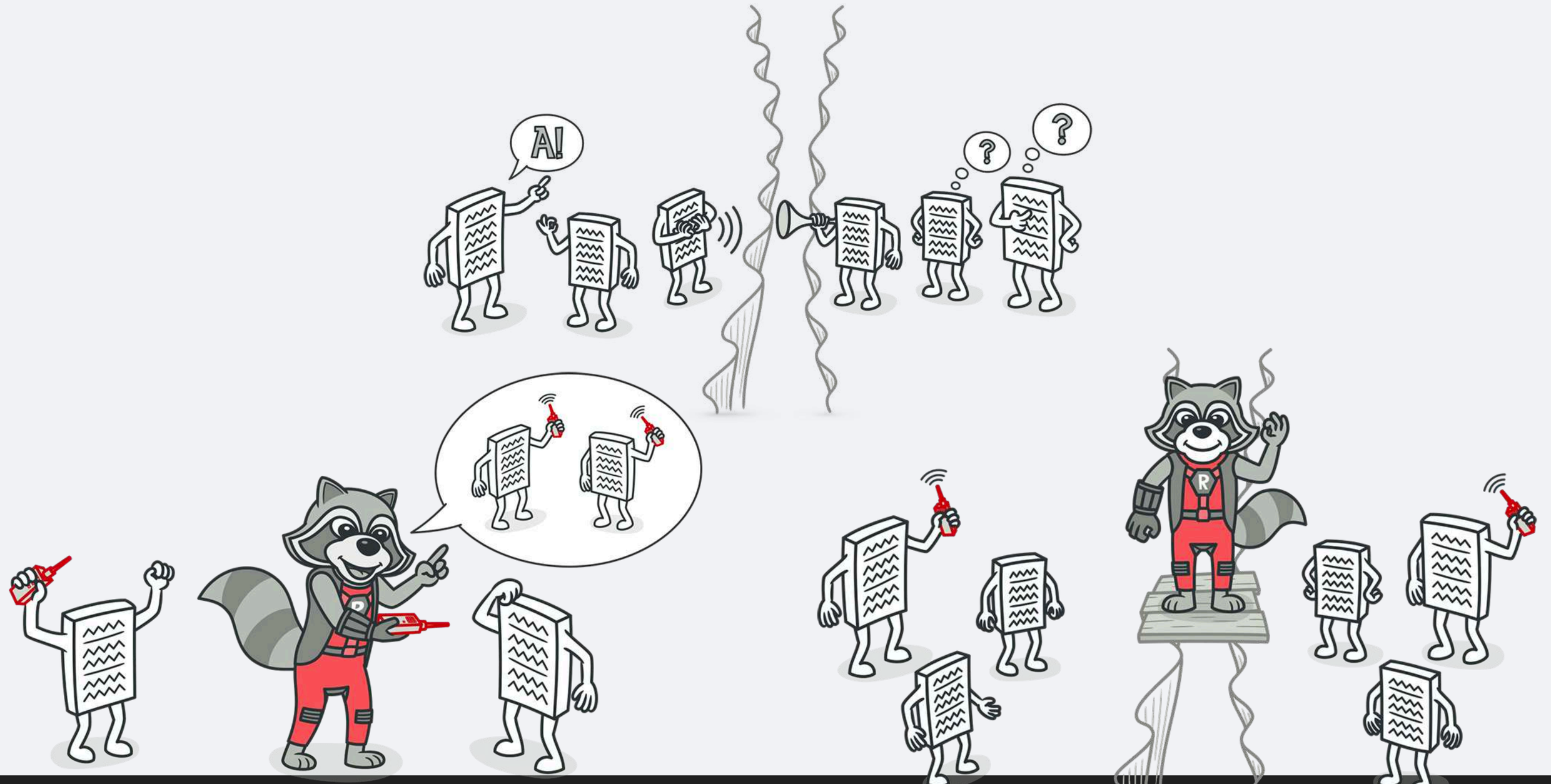
Acopladores





Intimidad inapropiada

Acopladores



Cadenas de mensajes

Acopladores

The Middle Man

Acopladores

Principios SOLID

Los principios de SOLID nos indican cómo organizar nuestras funciones y estructuras de datos en componentes y cómo dichos componentes deben estar interconectados.

Principios SOLID

- **S**ingle Responsibility: Responsabilidad única.
- **O**pen and Close: Abierto y cerrado.
- **L**iskov Substitution: Sustitución de Liskov.
- **I**nterface segregation: Segregación de interfaz.
- **D**ependency inversion: Inversión de dependencias.

SRP - Principio de responsabilidad única

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

– Robert C. Martin

SRP - Principio de responsabilidad única

“tener una única responsabilidad” !== “hacer una única cosa”

Ejemplo

SRP - Detectar violaciones

- Nombres de clases y módulos demasiado genéricos.
- Cambios en el código suelen afectar la clase o módulo.
- La clase involucra múltiples capas.
- Número elevado de importaciones.
- Cantidad elevada de métodos públicos.
- Excesivo número de líneas de código.

Principios SOLID

- **S**ingle Responsibility: Responsabilidad única.
- **O**pen and Close: Abierto y cerrado.
- **L**iskov Substitution: Sustitución de Liskov.
- **I**nterface segregation: Segregación de interfaz.
- **D**ependency inversion: Inversión de dependencias.

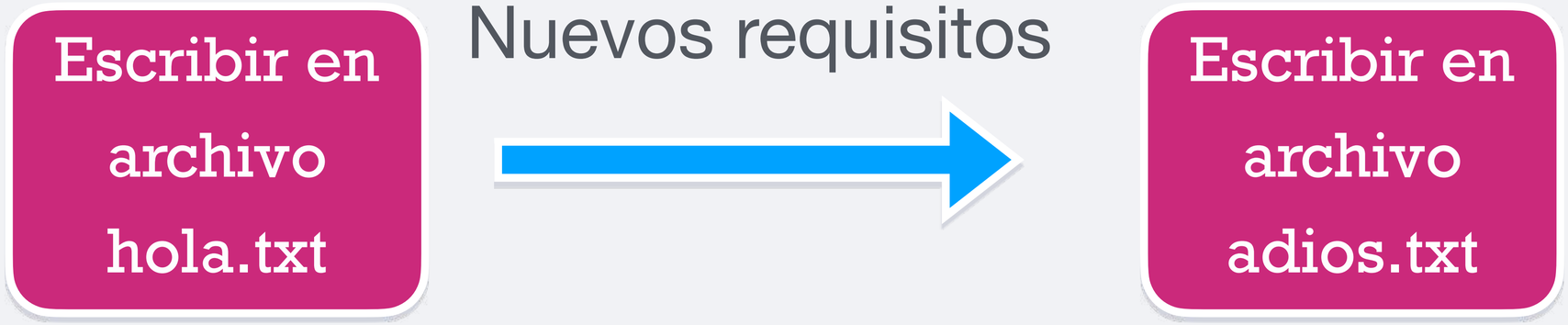
Principio de Abierto y Cerrado

Es un principio que depende mucho del contexto.

Establece que las entidades de software (clases, módulos, métodos, etc.) deben estar abiertas para la extensión, pero cerradas para la modificación.

Principio de Abierto y Cerrado

La forma más sencilla de demostrar este principio es considerar un método que hace una cosa.



Principio de Abierto y Cerrado

La forma más sencilla de demostrar este principio es considerar un método que hace una cosa.

```
writeFile( fileName: string)
```

```
writeFile( 'hola.txt' ) ;
```

```
writeFile( 'adios.txt' ) ;
```

Principio de Abierto y Cerrado

El principio abierto-cerrado también se puede lograr de muchas otras maneras, incluso mediante el uso de la herencia o mediante patrones de diseño de composición como el patrón de estrategia.

Principio de Abierto y Cerrado

Ejemplo

Detectar violaciones de OPC

- Cambios normalmente afectan nuestra clase o módulo.
- Cuando una clase o módulo afecta muchas capas. (Presentación, almacenamiento, etc.)

Principios SOLID

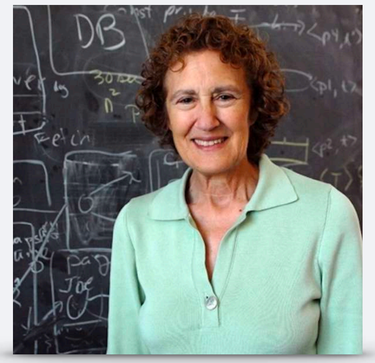
- **S**ingle Responsibility: Responsabilidad única.
- **O**pen and Close: Abierto y cerrado.
- **L**iskov Substitution: Sustitución de Liskov.
- **I**nterface segregation: Segregación de interfaz.
- **D**ependency inversion: Inversión de dependencias.

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

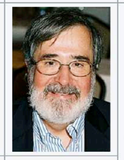
– Robert C. Martin

Principio de Substitución de Liskov

Doctora Barbara Jane Huberman, mas conocida como Barbara Liskov.



Liskov.- 2010

	Bob Kahn		
2005	Peter Naur		For fundamental contributions to programming language design and the definition of ALGOL 60, to compiler design, and to the art and practice of computer programming.
2006	Frances E. Allen		For pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.
2007	Edmund M. Clarke		For their roles in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries.[37]
	E. Allen Emerson		
	Joseph Sifakis		
2008	Barbara Liskov		For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.
2009	Charles P. Thacker		For his pioneering design and realization of the Xerox Alto, the first modern personal computer, and in addition for his contributions to the Ethernet and the Tablet PC.
2010	Leslie Valiant		For transformative contributions to the theory of computation, including the theory of probably approximately correct (PAC) learning, the complexity of enumeration and of algebraic computation, and the theory of parallel and distributed computing.
2011	Judea Pearl[38]		For fundamental contributions to artificial intelligence through the development of a calculus for probabilistic and causal reasoning.[39]
2012	Silvio Micali		For transformative work that laid the complexity-theoretic foundations for the science of cryptography and in the process pioneered new methods for efficient verification of mathematical proofs in complexity theory.[40]
	Shafi Goldwasser		

Turing Award

Por contribuciones a los fundamentos prácticos y teóricos del lenguaje de programación y el diseño de sistemas, especialmente relacionados con la abstracción de datos, la tolerancia a fallas y la computación distribuida.

{dev/talles}

Substitución de Liskov

“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”

Ejemplo

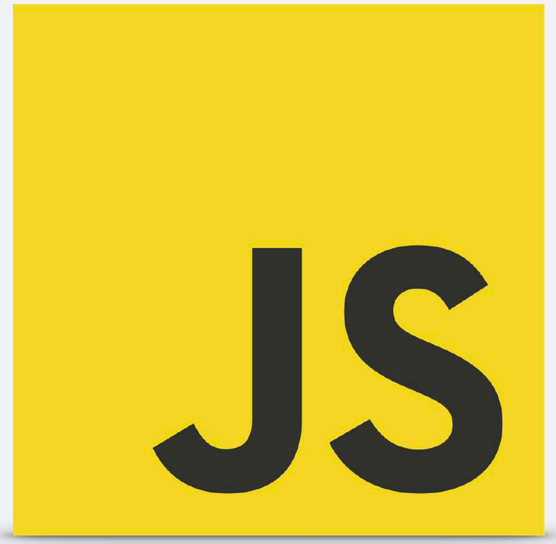
Principios SOLID

- **S**ingle Responsibility: Responsabilidad única.
- **O**pen and Close: Abierto y cerrado.
- **L**iskov Substitution: Sustitución de Liskov.
- **I**nterface segregation: Segregación de interfaz.
- **D**ependency inversion: Inversión de dependencias.

“Los clientes no deberían estar obligados a depender de interfaces que no utilicen”.

– Robert C. Martin

Principio de segregación de la interfaz



Principio de segregación de la interfaz

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

Ejemplo

Detectar violaciones ISP

- Si las interfaces que diseñamos nos obligan a violar los principios de responsabilidad única y substitución de Liskov.

Principios SOLID

- **S**ingle Responsibility: Responsabilidad única.
- **O**pen and Close: Abierto y cerrado.
- **L**iskov Substitution: Sustitución de Liskov.
- **I**nterface segregation: Segregación de interfaz.
- **D**ependency inversion: Inversión de dependencias.

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

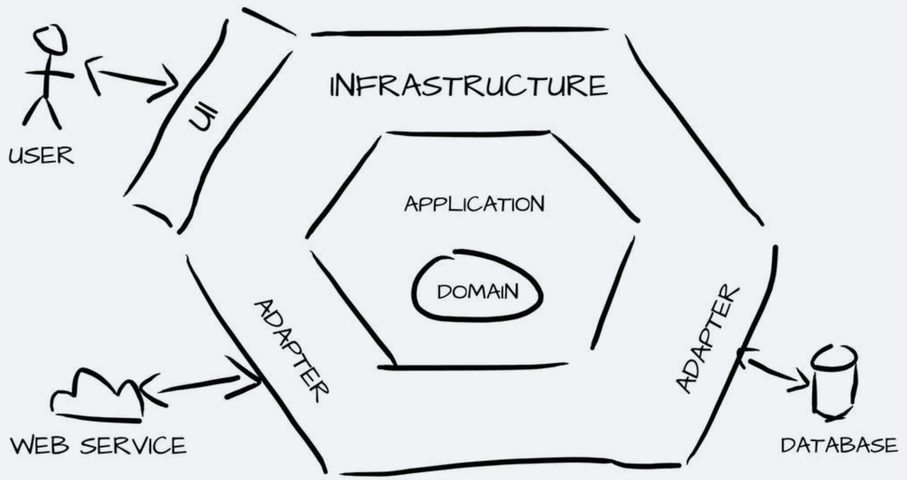
Principio de inversión de dependencias

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel.
- Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de detalles.
- Los detalles deberían depender de abstracciones.

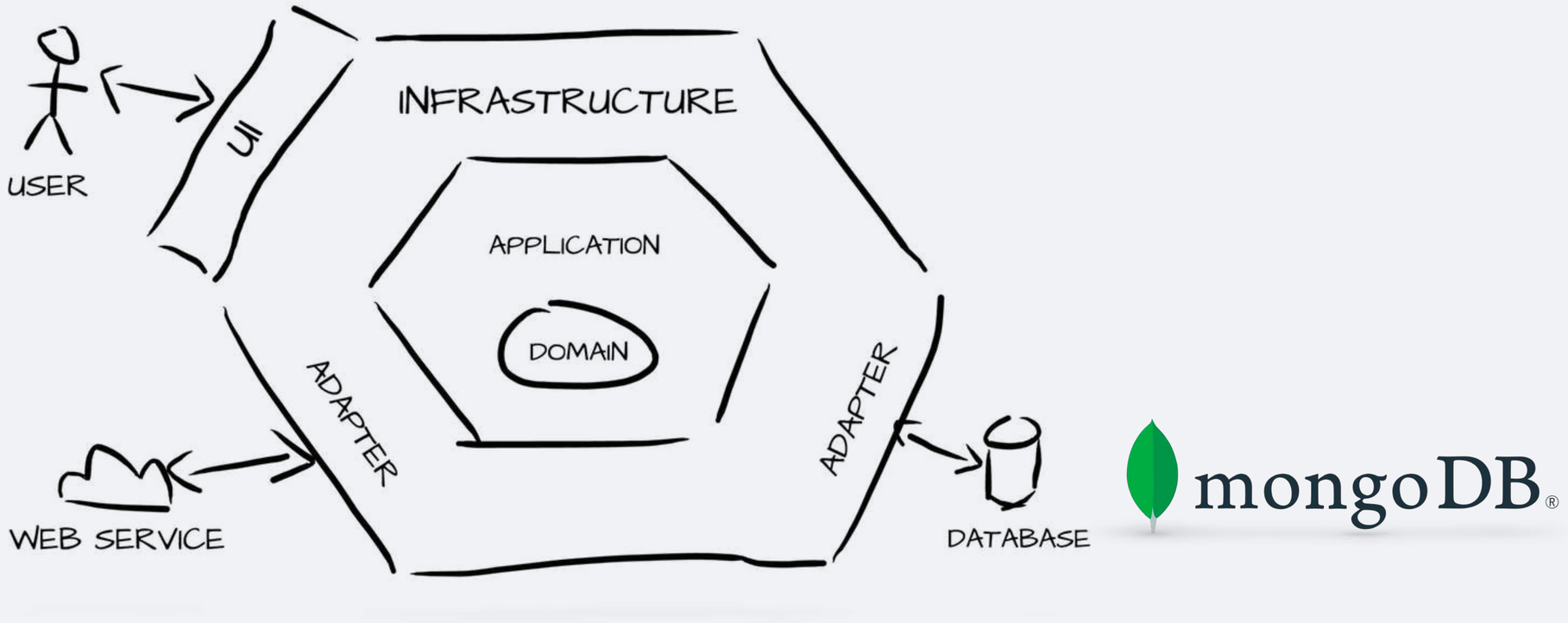
Principio de inversión de dependencias

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.



Principio de inversión de dependencias



Depender 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.

¿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.

Inyección de dependencias

Dependencia en programación, significa que un módulo o componente requiere de otro para poder realizar su trabajo.

En algún momento nuestro programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase, es cuando debemos usar inyección de dependencias.

Ejemplo

```
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 }
```

Ejemplo

Diagrama UML de dependencias visible

```
class UseCase{
  constructor(){
    this.externalService = new ExternalService();
  }

  doSomething(){
    this.externalService.doExternalTask();
  }
}

class ExternalService{
  doExternalTask(){
    console.log("Doing task...")
  }
}
```

```
class UseCase{
  constructor(externalService: ExternalService){
    this.externalService = externalService;
  }

  doSomething(){
    this.externalService.doExternalTask();
  }
}

class ExternalService{
  doExternalTask(){
    console.log("Doing task...")
  }
}
```