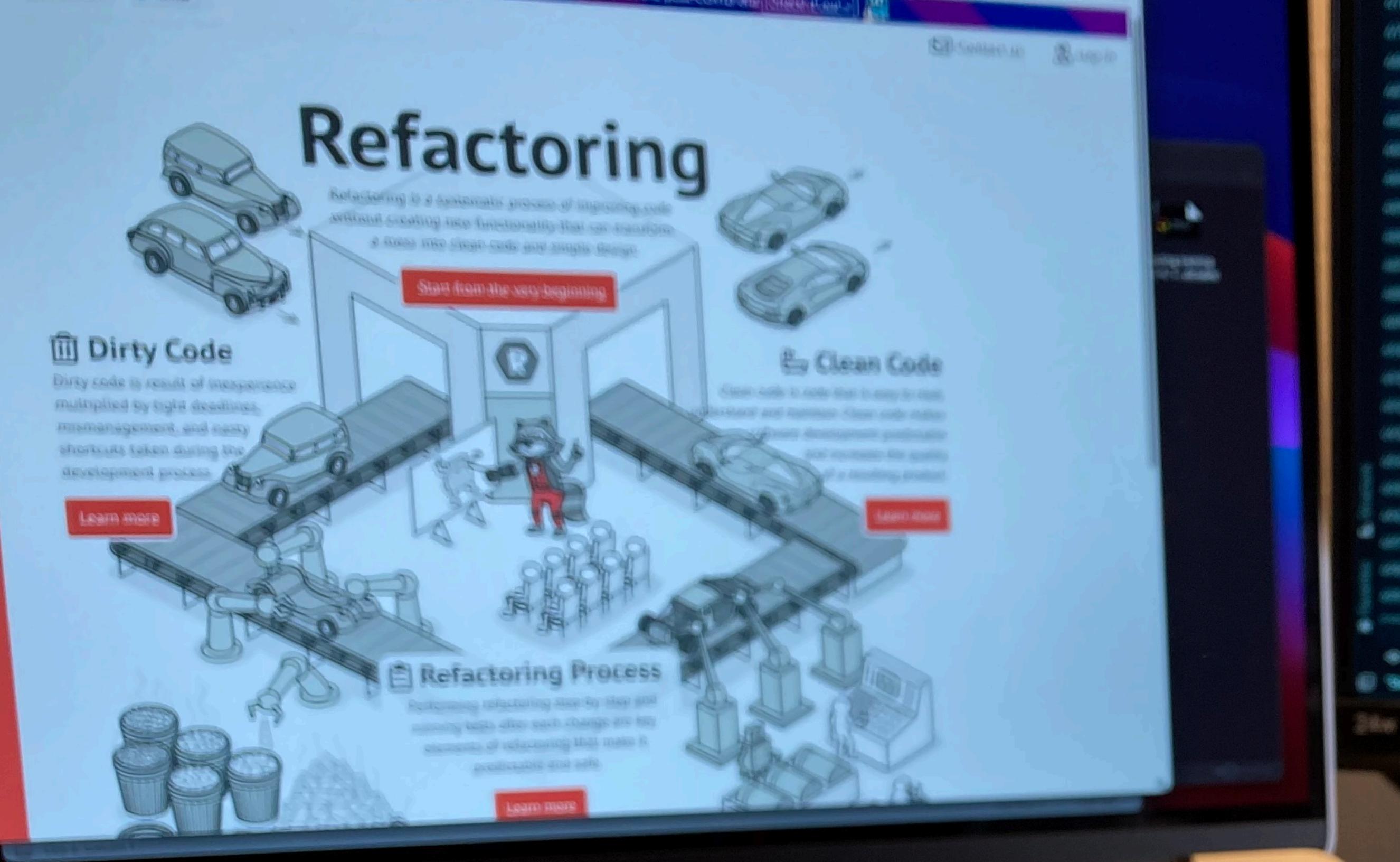


Refactoring Avanzado

por Fran Iglesias
Staff Software Engineer





Bloque 1

Introducción

Ejercicio

¿Qué hace este código?

```
class E {
    private s: number;
    private yE: number;

    constructor(s: number, yE: number) {
        this.s = s;
        this.yE = yE;
    }

    bon(): number {
        if (this.yE > 5) {
            return this.s * 0.2;
        }
        return this.s * 0.1;
    }
}
```

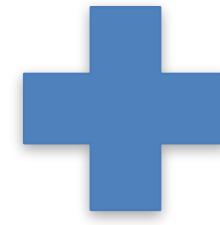
Ejercicio

¿Como introducirías el código para añadir un nuevo tipo de descuento (3 x 2)?

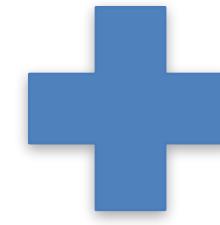
```
export function processOrdersWithDiscounts(
  orders: Array<{ id: string; items: Array<{ price: number }>; customer: { isVip: boolean } }>,
  ): number {
  let total = 0
  for (const order of orders) {
    if (order.items && order.items.length > 0) {
      for (const item of order.items) {
        if (order.customer && order.customer.isVip) {
          if (item.price > 100) {
            total += item.price * 0.8 // gran descuento VIP
          } else {
            total += item.price * 0.9 // pequeño descuento VIP
          }
        } else {
          if (item.price > 100) {
            total += item.price * 0.95 // gran descuento regular
          } else {
            total += item.price // sin descuento
          }
        }
      }
    }
  }
  return total
}
```

EL COSTE DEL CÓDIGO

Complejidad
esencial



Complejidad
accidental



Entropía

**Complejidad
esencial**



**Complejidad
accidental**



Entropía

La propia del
problema que
resolvemos

**Complejidad
esencial**



**Complejidad
accidental**



Entropía

Asociada con
las decisiones
técnicas con
las que se
implementa la
solución



El desorden del
código debido
a la
acumulación
de los cambios
en el tiempo

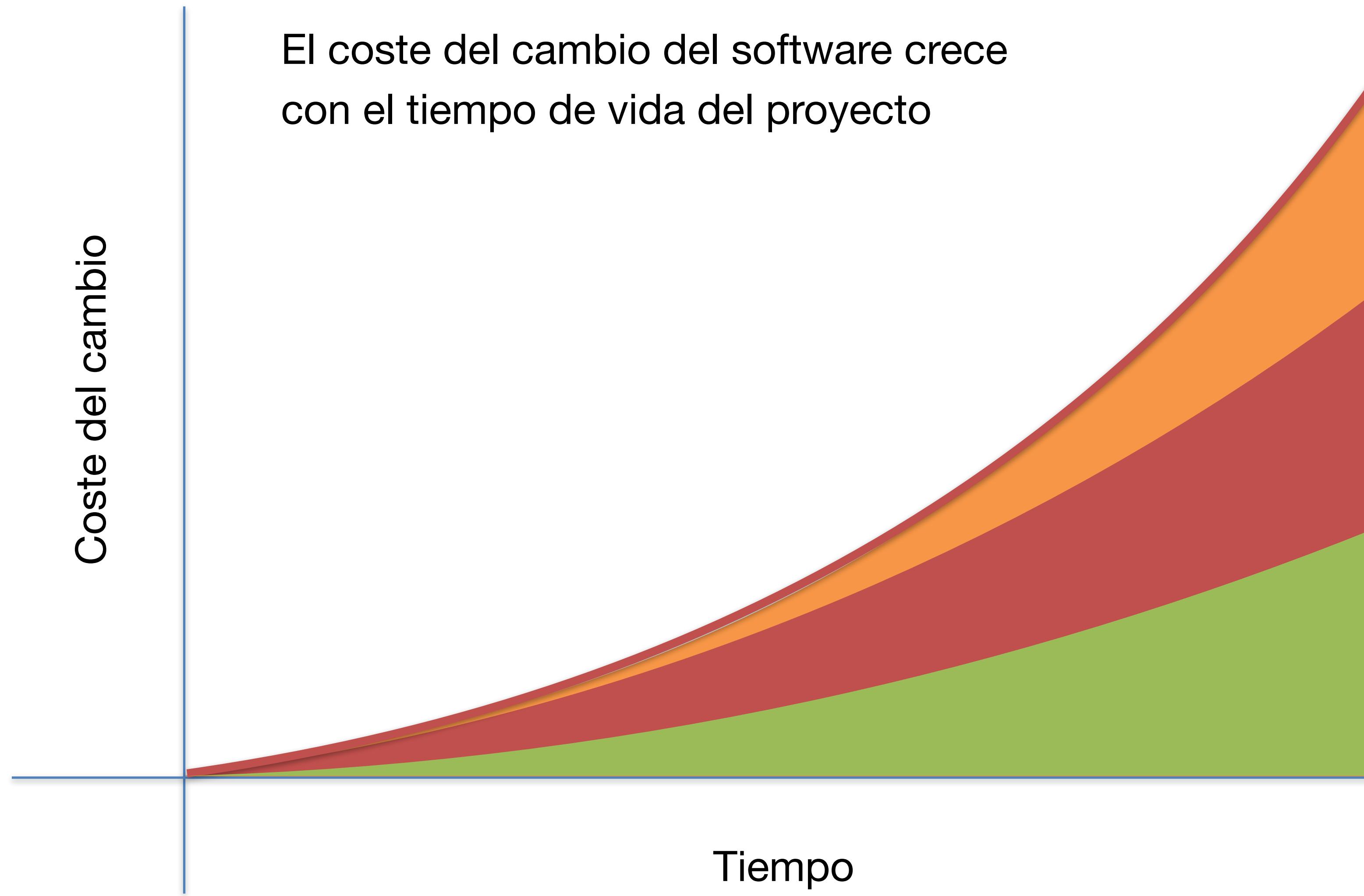


La propia del problema que resolvemos

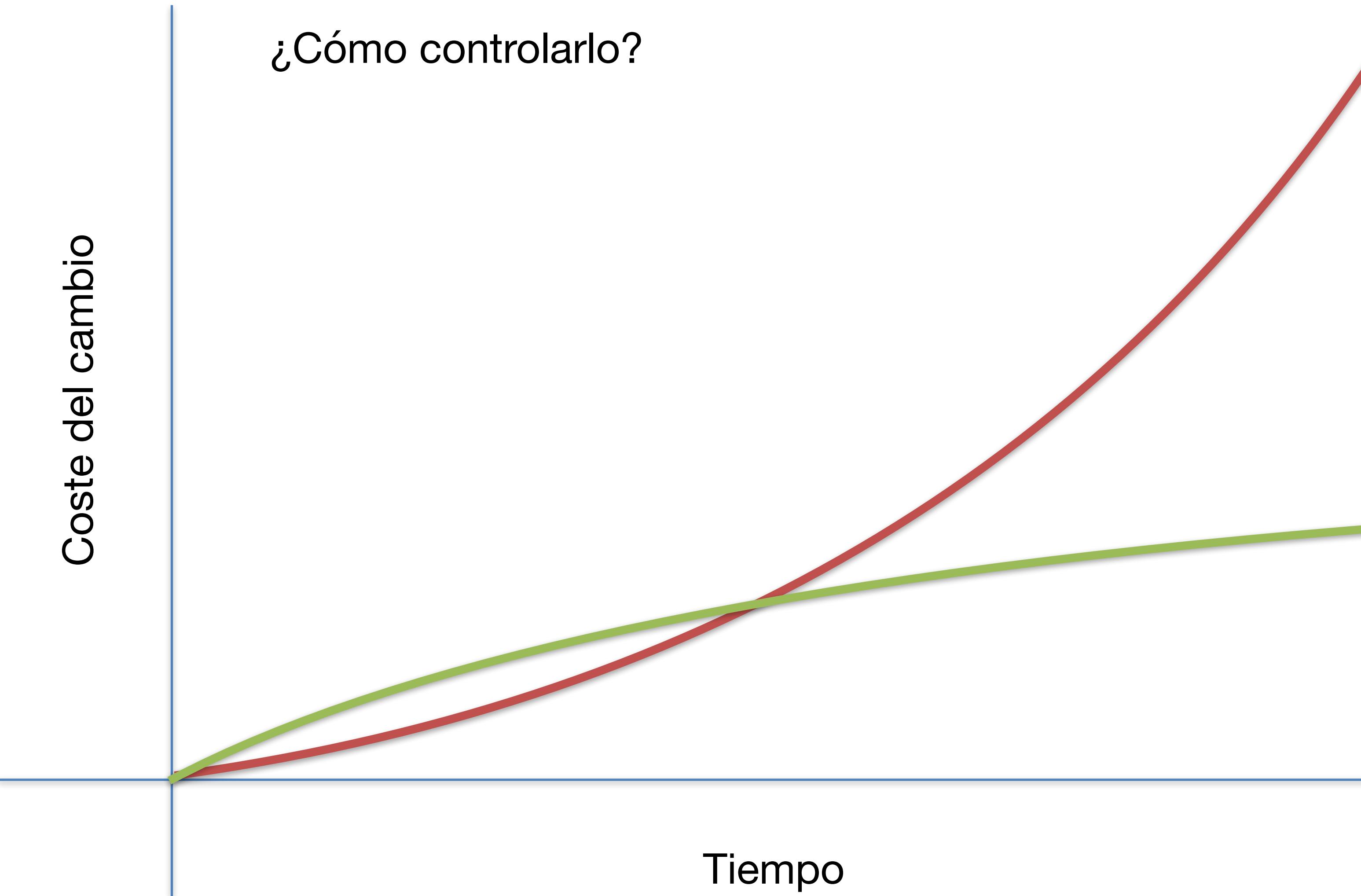
Asociada con las decisiones técnicas con las que se implementa la solución

El desorden del código debido a la acumulación de los cambios en el tiempo

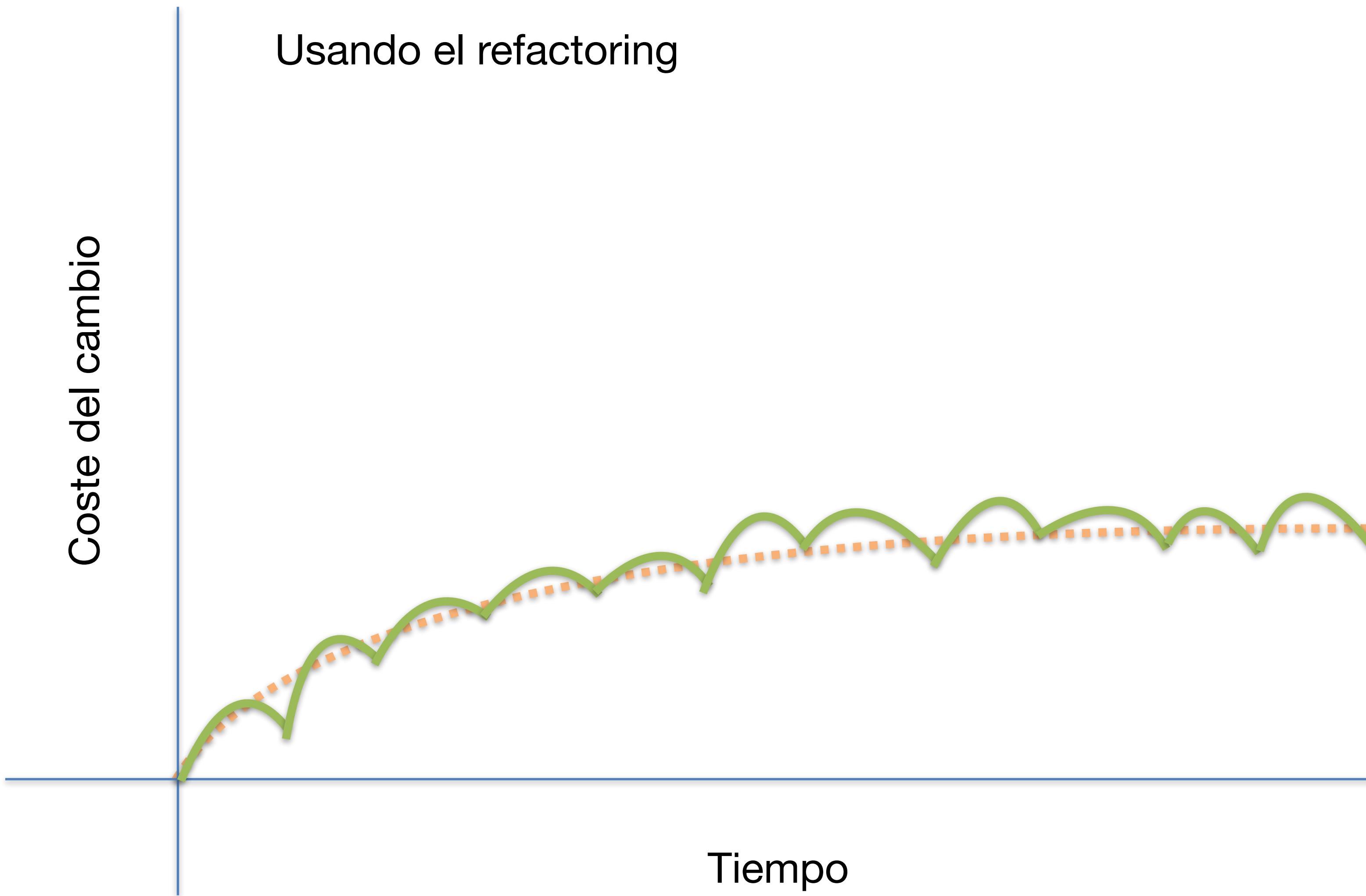
El coste del cambio del software crece
con el tiempo de vida del proyecto



¿Cómo controlarlo?



Usando el refactoring



El *refactoring* actúa sobre

Complejidad
esencial



Complejidad
accidental



Entropía

Mejorando el
modelado de la
solución

Reduciendo el
acoplamiento
con las
tecnologías de
tercera parte

Manteniendo
control sobre
los efectos de
la acumulación
de cambios

Ejercicio

¿A qué tipo de complejidad afectan estos requerimientos?

1. Introducir un nuevo estado en el ciclo de vida de un objeto de negocio.
2. Dar soporte a un nuevo método de pago.
3. Dar soporte a un nuevo requisito legal que obliga a tener una firma digital de una operación.
4. Cambiar el formato en que se generan ciertos informes.
5. Añadir un endpoint a una API para habilitar una nueva funcionalidad
6. Cambiar el ORM usado para hablar con una base de datos

QUÉ ES REFACTORING



Make it work



Make it right



Make it fast

Kent Beck



Make it work



Make it right



Make it fast



Un proceso intencionado y sistemático por el que modificamos la estructura o diseño del código sin alterar su comportamiento a fin de conseguir que su mantenimiento sea sostenible en el largo plazo.

CUÁNDO REFACTORIZAR



Leyendo código

En cualquier momento en el que nos encontramos un código que no entendemos

Refactor preparatorio

Refactor a posteriori

Leyendo código



**Refactor
preparatorio**

Preparando el
código para
facilitar la
introducción de
una nueva
característica

**Refactor a
posteriori**



Leyendo código



**Refactor
preparatorio**



**Refactor a
posteriori**

Tras introducir
un cambio, para
que sea fácil de
entender y
extender en el
futuro



Leyendo código

En cualquier momento en el que nos encontramos un código que no entendemos



Refactor preparatorio

Preparando el código para facilitar la introducción de una nueva característica



Refactor a posteriori

Tras introducir un cambio, para que sea fácil de entender y extender en el futuro

QUÉ NO ES REFACTORING



Make it fast

Optimización

Puede incrementar la complejidad



Reescritura

El código actual, por complejo que sea paga las facturas

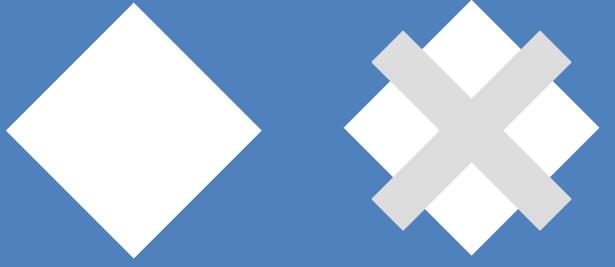
CUATRO REGLAS *DEL DISEÑO SIMPLE*



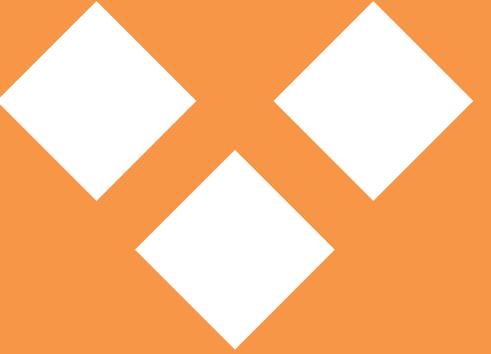
Pasa los tests



Expresa
intención



Reduce
duplicación



Minimiza los
elementos

 <p>Pasa los tests</p>	Hace lo que dice	Previene efectos no deseados	Menos tiempo con Bugs
 <p>Expresa intención</p>	Dice lo que hace	Facilita entender cómo funciona	Menos tiempo lectura
 <p>Reduce duplicación</p>	Hay un sitio para cada unidad de conocimiento	Facilita intervenir	Menos tiempo intervención
 <p>Minimiza elementos</p>	Mantiene controlada complejidad	Facilita saber dónde intervenir	Más fácil intervenir



Pasa los tests

Hace lo que dice

Previene efectos no deseados

Menos tiempo con Bugs



Expresa intención

Dice lo que hace

Facilita entender cómo funciona

Menos tiempo lectura



Reduce duplicación

Hay un sitio para cada unidad de conocimiento

Facilita intervenir

Menos tiempo intervención



Minimiza elementos

Mantiene controlada complejidad

Facilita saber dónde intervenir

Más fácil intervenir



Pasa los tests

Hace lo que dice

Previene efectos no deseados

Menos tiempo con Bugs



Expresa intención

Dice lo que hace

Facilita entender cómo funciona

Menos tiempo lectura



Reduce duplicación

Hay un sitio para cada unidad de conocimiento

Facilita intervenir

Menos tiempo intervención



Minimiza elementos

Mantiene controlada complejidad

Facilita saber dónde intervenir

Más fácil intervenir



Pasa los tests

Hace lo que dice

Previene efectos no deseados

Menos tiempo con Bugs



Expresa intención

Dice lo que hace

Facilita entender cómo funciona

Menos tiempo lectura



Reduce duplicación

Hay un sitio para cada unidad de conocimiento

Facilita intervenir

Menos tiempo intervención



Minimiza elementos

Mantiene controlada complejidad

Facilita saber dónde intervenir

Más fácil intervenir

REFACTOR

Cancel

Refactor

&
6

7

(

8

)

9

0

v

U

I

Y

5

¿Qué es un refactor?

Es un procedimiento concreto para obtener una determinada transformación en el código

```
function tax(p: number): number {  
    return p * 1.21  
}
```

Rename

Rename

```
function applyVAT(price: number): number {  
    return price * 1.21  
}
```

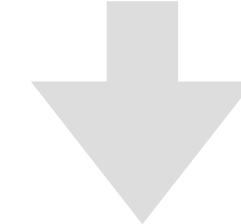
Refactor probado

Es un refactor cuyo efecto es predecible y habitualmente está automatizado.

Para aplicarlo no necesitamos la protección de tests.

```
function applyVAT(price: number): number {  
    return price * 1.21  
}
```

Introduce constant

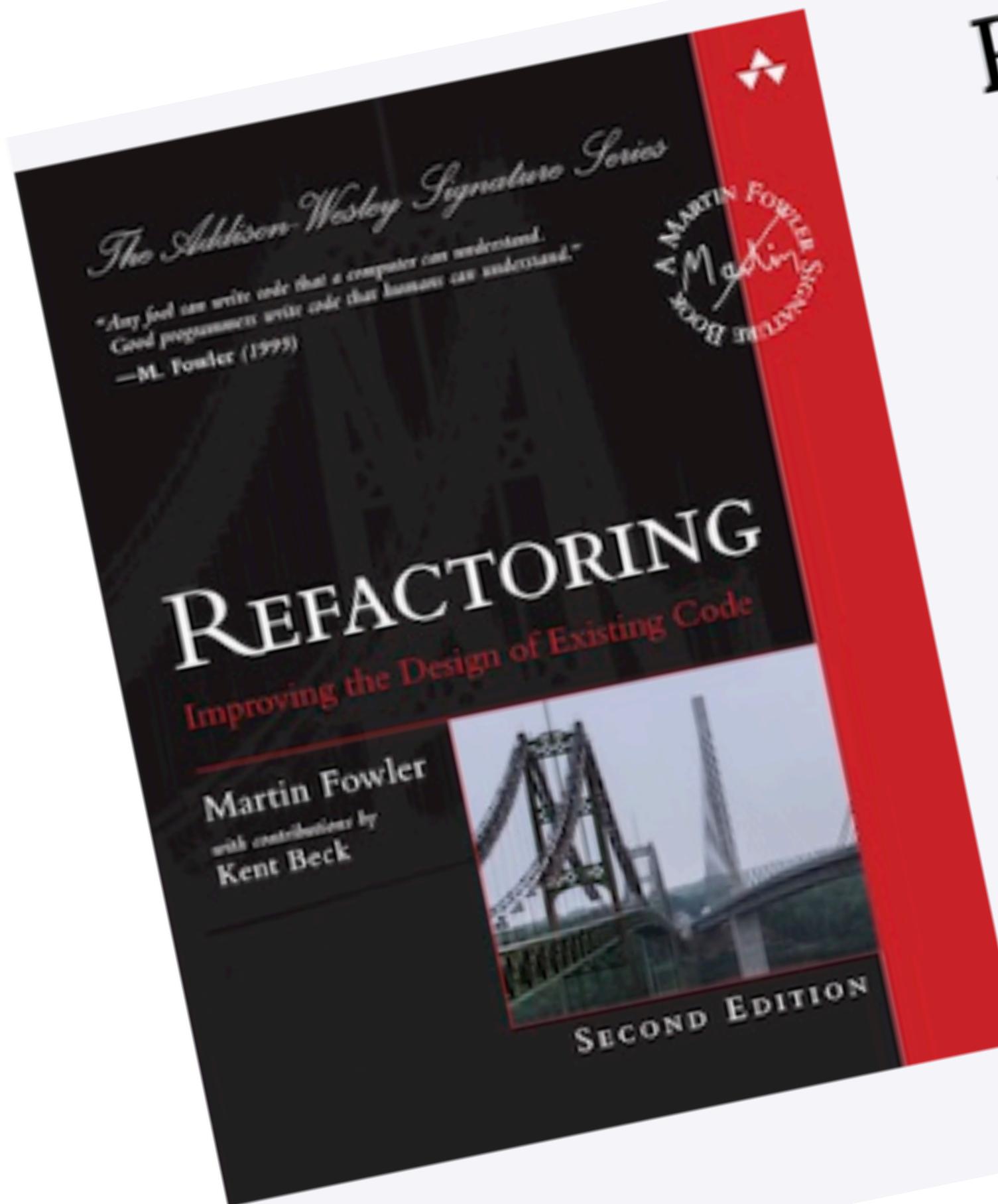


```
const VAT_RATE=1.21  
function applyVAT(price: number): number {  
    return price * VAT_RATE  
}
```

RECURSOS

El manual de referencia

<https://martinfowler.com/books/refactoring.html>



Refactoring
Improving the Design of Existing Code
by Martin Fowler, with Kent Beck
2018

The guide to how to transform code with safe and rapid process, vital to keeping it
cheap and easy to modify for future needs.

amazon informIT

Notes for buying my books

Mi tienda



Refactoring

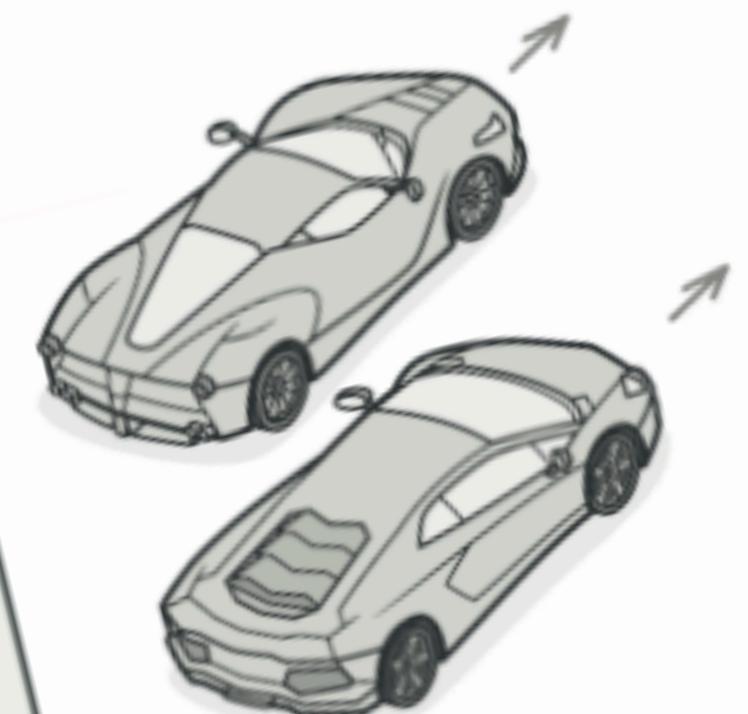
Refactoring is a systematic process of improving code without creating new functionality that can transform a mess into clean code and simple design.

Start from the very beginning

Dirty Code

Dirty code is result of inexperience multiplied by tight deadlines, mismanagement, and nasty shortcuts taken during the development process.

Learn more



Clean Code

Clean code is code that is easy to read, understand and maintain. Clean code makes software development predictable and increases the quality of a resulting product.

Learn more

Con el curso...

La guía del refactor cotidiano

Mantener tu código en forma es un trabajo para todos los días



Este libro está completo al 100%
ÚLTIMA ACTUALIZACIÓN EL 2025-01-11

\$7.99 PRECIO MÍNIMO \$19.99 PRECIO SUGERIDO

USTED PAGA

\$19.99

EL AUTOR GANA

\$15.99

USTED PAGA

\$19.99

Ciudades de la UE: Precio sin IVA.
El IVA se añade durante el pago.

Puede pagar en US \$ o en su moneda local (EUR, GBP, CAD, etc.)

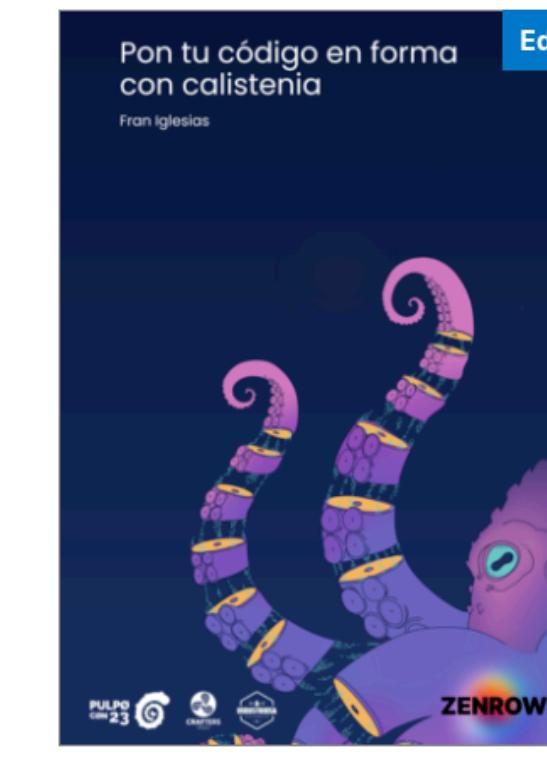
cuando realice el pago con tarjeta de crédito usando Stripe.

Añadir Ebook al Carrito

[Añadir a la Lista de Deseos](#)

Pon tu código en forma con calistenia

Conoce, practica y aplica los consejos de Object Calisthenics, para mejorar la calidad del diseño de tu código



This book is 100% complete
LAST UPDATED ON 2023-11-29

\$19.00 MINIMUM PRICE \$29.00 SUGGESTED PRICE

YOU PAY

\$29.00

AUTHOR EARNES

\$23.20

YOU PAY

\$29.00

EU customers: Price excludes VAT.
VAT is added during checkout.

You can pay in US \$ or in your local currency (EUR, GBP, CAD, etc.)
when you checkout with a credit card using Stripe.

Add Ebook to Cart

[Add to Wish List](#)

Bloque 2

Controlando la entropía: Calistenia

Calistenia

Un conjunto de reglas que, aplicadas,
nos llevan a un mejor diseño del código



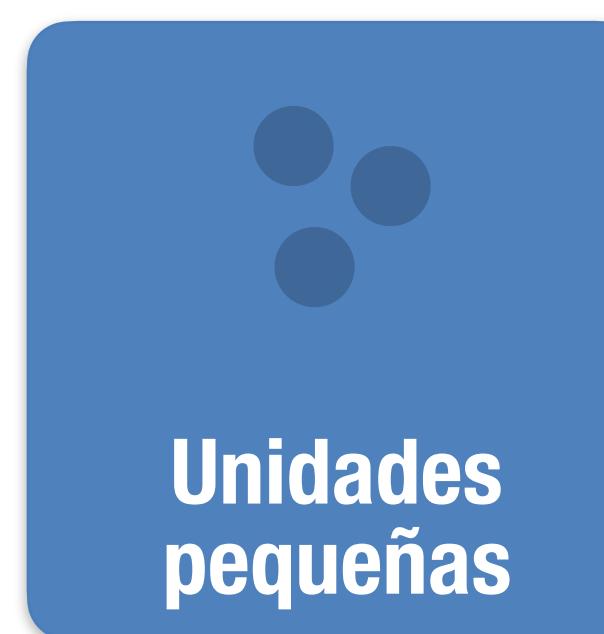
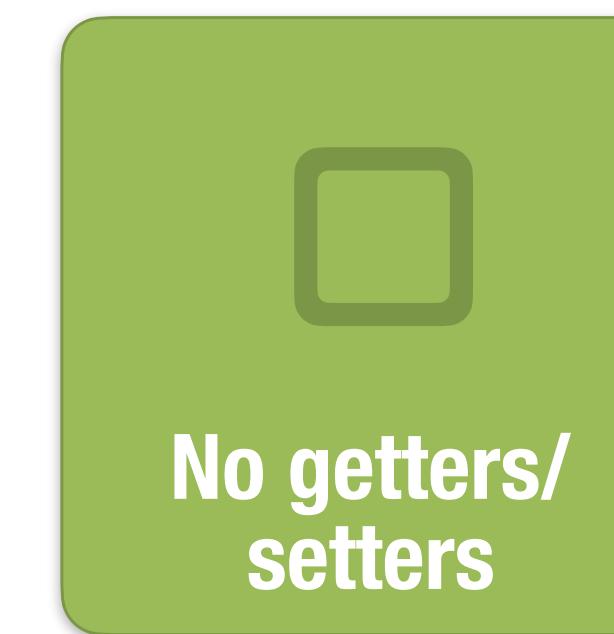
<https://bolcom.github.io/student-dojo/legacy-code/DevelopersAnonymous-ObjectCalisthenics.pdf>

Reglas de Calistenia

Se basan en fijarnos
en ciertas
características
aplicables a
cualquier código.

Si observamos que
el código incumple
alguna de esas
reglas, intentamos
arreglarlo para que
se ajuste a ella.

Reglas de calistenia



No usar abreviaturas

```
class E {
    private s: number;
    private yE: number;

    constructor(s: number, yE: number) {
        this.s = s;
        this.yE = yE;
    }

    bon(): number {
        if (this.yE > 5) {
            return this.s * 0.2;
        }
        return this.s * 0.1;
    }
}
```

No usar abreviaturas

```
class Employee {  
    private salary: number;  
    private yearsOfExperience: number;  
  
    constructor(salary: number, yearsOfExperience: number) {  
        this.salary = salary;  
        this.yearsOfExperience = yearsOfExperience;  
    }  
  
    bonus(): number {  
        if (this.yearsOfExperience > 5) {  
            return this.salary * 0.2;  
        }  
        return this.salary * 0.1;  
    }  
}
```

Rename

Un solo nivel de indentación

```
function processOrder(order: Order): void {
    if (order.isValid()) {
        if (!order.isPaid()) {
            if (order.hasStock()) {
                order.pay();
                console.log("Order processed.");
            } else {
                console.log("Out of stock");
            }
        } else {
            console.log("Order already paid");
        }
    } else {
        console.log("Invalid order");
    }
}
```

Un solo nivel de indentación

```
function processOrder(order: Order): void {
    if (!order.isValid()) {
        console.log("Invalid order");
        return
    }
    if (order.isPaid()) {
        console.log("Order already paid");
        return
    }
    if (!order.hasStock()) {
        console.log("Out of stock");
        return
    }
    order.pay();
    console.log("Order processed");
}
```

Cláusulas de
guarda

Extraer
métodos

Un solo nivel de indentación

```
function processOrder(order: Order): void {
    assertValidOrder(order)
    assertPaidOrder(order)
    assertStock(order)

    order.pay();
    console.log("Order processed");
}

function assertValidOrder(order: Order): void {
    if (!order.isValid()) {
        throw new Error("Invalid order");
    }
}

function assertPaidOrder(order: Order): void {
    if (order.isPaid()) {
        throw new Error("Order already paid");
    }
}

function assertStock(order: Order): void {
    if (!order.hasStock()) {
        throw new Error("Out of stock");
    }
}
```

Cláusulas de
guarda

Extraer
métodos

No usar else

```
function categorizeUser(age: number): string {  
    let category:string  
    if (age < 13) {  
        category = "child"  
    } else if (age < 18) {  
        category = "teenager"  
    } else {  
        category = "adult"  
    }  
  
    return category  
}
```

No usar else

```
function categorizeUser(age: number): string {  
    if (age < 13) {  
        return "child"  
    }  
    if (age < 18) {  
        return "teenager"  
    }  
    return "adult"  
}
```

Cláusulas de
guarda

Switch/Map

Extraer método

Encapsular primitivas

```
class User {  
    private name: string;  
    private email: string;  
  
    constructor(  
        name: string,  
        email: string  
    ) {  
        if (name.length < 3) {  
            throw new Error("Name must be at least 3 characters")  
        }  
        this.name = name;  
        if (!email.includes("@")) {  
            throw new Error("Invalid email");  
        }  
        this.email = email;  
    }  
  
    changeEmail(newEmail: string): void {  
        if (!newEmail.includes("@")) {  
            throw new Error("Invalid email");  
        }  
        this.email = newEmail;  
    }  
}
```

Encapsular primitivas

```
class UserName {  
    private name: string;  
  
    constructor(name: string) {  
        if (name.length < 3) throw new Error("Name must be at least 3  
characters")  
        this.name = name;  
    }  
}  
  
class Email {  
    private email: string;  
    constructor(email: string) {  
        if (!email.includes("@")) throw new Error("Invalid email")  
        this.email = email;  
    }  
}  
  
class User {  
    constructor(  
        private name: UserName,  
        private email: Email  
    ) {}  
    changeEmail(newEmail: string): void {  
        this.email = new Email(newEmail)  
    }  
}
```

Introducir Value Objects

Colecciones de primera clase

```
class Project {  
    private name: string = "";  
    private responsible: string = "";  
    private tasks: string[] = [ ];  
  
    addTask(task: string): void {  
        this.tasks.push(task);  
    }  
  
    getTasks(): string[] {  
        return this.tasks;  
    }  
}  
  
const project = new Project();  
project.addTask("Design UI");  
project.getTasks().push("Hack the system");
```

Colecciones de primera clase

```
class Tasks {
    tasks: string[] = []

    constructor(tasks: string[]) {
        this.tasks = tasks
    }

    addTask(task: string): void {
        this.tasks.push(task)
    }

    getTasks(): string[] {
        return this.tasks
    }
}

class Project {
    private name: string = "";
    private responsible: string = "";
    private tasks: Tasks = new Tasks([]);

    addTask(task: string): void {
        this.tasks.addTask(task);
    }

    getTasks(): string[] {
        return this.tasks.getTasks()
    }
}
```

Envolver
colección en
objeto

No getters, setters o propiedades públicas

```
class BankAccount {
    private balance: number;

    constructor(balance: number) {
        this.balance = balance;
    }

    getBalance(): number {
        return this.balance;
    }

    setBalance(amount: number): void {
        this.balance = amount;
    }
}

const account = new BankAccount(1000);
const current = account.getBalance();
account.setBalance(current - 200);
```

No getters, setters o propiedades públicas

```
class BankAccount {  
    private balance: number;  
  
    constructor(balance: number) {  
        this.balance = balance;  
    }  
  
    getBalance(): number {  
        return this.balance;  
    }  
  
    deposit(amount: number): void {  
        this.balance += amount;  
    }  
  
    withdraw(amount: number): void {  
        if (amount > this.balance) {  
            throw new Error('Insufficient funds');  
        }  
        this.balance -= amount;  
    }  
}
```

Introducir
método

Mantener las unidades pequeñas

```
class Order {
    private items: { name: string; price: number; quantity: number }[] = [];
    private discount: number = 0;

    addItem(name: string, price: number, quantity: number): void {
        this.items.push({ name, price, quantity });
    }

    setDiscount(discount: number): void {
        this.discount = discount;
    }

    calculateTotal(): number {
        let total = 0;
        for (const item of this.items) {
            total += item.price * item.quantity;
        }
        return total - this.discount;
    }

    printInvoice(): void {
        console.log("INVOICE");
        for (const item of this.items) {
            console.log(` ${item.name} x${item.quantity}: ${item.price * item.quantity}`);
        }
        console.log(`Discount: ${this.discount}`);
        console.log(`Total: ${this.calculateTotal()}`);
    }
}
```

Mantener las unidades pequeñas (1)

```
class Order {
    private items: Items = new Items();
    private discount: number = 0;

    addItem(name: string, price: number, quantity: number): void {
        this.items.addItem(name, price, quantity);
    }

    applyDiscount(discount: number): void {
        this.discount = discount;
    }

    calculateTotal(): number {
        return this.items.totalAmount() - this.discount;
    }

    printInvoice(): void {
        const printer = new InvoicePrinter();
        printer.print(this.items.print(), this.calculateTotal(),
this.discount);
    }
}
```

Extraer clases

Introducir Value Objects

Mantener las unidades pequeñas (2)

```
class OrderItem {
  name: string
  price: number
  quantity: number

  constructor(name: string, price: number, quantity: number) {
    this.name = name
    this.price = price
    this.quantity = quantity
  }

  amount(): number {
    return this.price * this.quantity;
  }

  line(): string {
    return `${this.name} x${this.quantity}: ${this.amount()}`;
  }
}

class Items {
  items: OrderItem[] = [];

  addItem(name: string, price: number, quantity: number): void {
    this.items.push(new OrderItem(name, price, quantity));
  }

  totalAmount(): number {
    return this.items.reduce((acc, item) => acc + item.amount(), 0);
  }

  lines(): string {
    return this.items.map(item => item.line()).join('\n');
  }
}
```

Extraer clases

Introducir Value Objects

Mantener las unidades pequeñas (3)

```
class InvoicePrinter {  
    print(lines: string, total: number, discount: number): void  
{  
    console.log("INVOICE");  
    console.log(lines);  
    console.log(`Discount: ${discount}`);  
    console.log(`Total: ${total}`);  
}  
}
```

Extraer clases

Introducir Value
Objects

No más de dos variables de instancia

```
class Report {  
    constructor(  
        private title: string,  
        private author: string,  
        private content: string,  
        private createdAt: Date,  
        private tags: string[]  
    ) {}  
  
    printSummary(): void {  
        console.log(`Report: ${this.title} by ${this.author} (${  
            this.createdAt.toDateString()});  
    }  
}
```

No más de dos variables de instancia (1)

```
class Report {  
    private metadata: Metadata  
    private content: Content  
  
    constructor(  
        metadata: Metadata,  
        content: Content  
    ) {  
        this.metadata = metadata;  
        this.content = content;  
    }  
  
    printSummary(): void {  
        const template = `Report: {{title}}\n{{content}}  
\n{{createdAt}}\nTags: {{tags}}`;  
        const templateWithContent = this.content.print(template);  
        console.log(this.metadata.print(templateWithContent));  
    }  
}
```

Introducir Value Objects

No más de dos variables de instancia (2)

```
class Metadata {
    private author: string
    private createdAt: Date
    private tags: string[]

    constructor(author: string, createdAt: Date, tags: string[]) {
        this.author = author;
        this.createdAt = createdAt;
        this.tags = tags;
    }

    print(template: string): string {
        return template.replace('{{author}}', this.author)
            .replace('{{createdAt}}', this.createdAt.toDateString())
            .replace('{{tags}}', this.tags.join(', '))
    }
}

class Content {
    private title: string
    private content: string

    constructor(title: string, content: string) {
        this.title = title;
        this.content = content;
    }

    print(template: string): string {
        return template.replace('{{title}}', this.title)
            .replace('{{content}}', this.content)
    }
}
```

Introducir Value Objects

Máximo un punto por línea

```
class Address {  
    constructor(private city: string) {}  
    getCity(): string {  
        return this.city;  
    }  
}  
  
class Customer {  
    constructor(private address: Address) {}  
    getAddress(): Address {  
        return this.address;  
    }  
}  
  
class Order {  
    constructor(private customer: Customer) {}  
    getCustomer(): Customer {  
        return this.customer;  
    }  
}  
  
const order = new Order(new Customer(new Address("Madrid")));  
console.log(order.getCustomer().getAddress().getCity());
```

Máximo un punto por línea

```
class Address {  
    constructor(private city: string) {}  
    getCity(): string {  
        return this.city;  
    }  
}  
  
class Customer {  
    constructor(private address: Address) {}  
    getAddress(): Address {  
        return this.address;  
    }  
  
    getCity(): string {  
        return this.getAddress().getCity();  
    }  
}  
  
class Order {  
    constructor(private customer: Customer) {}  
    getCustomer(): Customer {  
        return this.customer;  
    }  
  
    getCity(): string {  
        return this.getCustomer().getCity();  
    }  
}  
  
const order = new Order(new Customer(new Address("Madrid")));  
console.log(order.getCity());
```

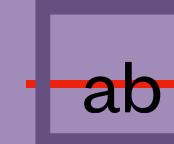
Ocultar
delegado

Introducir
método

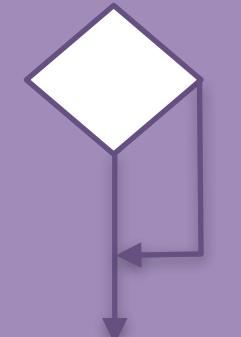
Estas reglas se
pueden aplicar a
cualquier código, en
cualquier orden.

Si las tenemos
presentes al escribir
código o al
refactorizarlo, el
código irá
evolucionando
hacia un mejor
diseño.

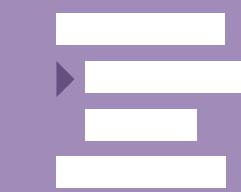
Reglas de calistenia



No abreviaturas



No usar ELSE



Solo 1 nivel
indentación



Encapsular
primitivas



Colecciones de
Primera Clase



No getters/
setters



Unidades
pequeñas

2

Máx. 2
variables
instancia



Un punto por
línea

Tennis refactoring kata

<https://github.com/emilybache/Tennis-Refactoring-Kata>

1. Escoge una de las variantes de TennisGame
2. Identifica violaciones de las reglas de calistenia en esa variante
3. Modifica el código para reducir o eliminar esas violaciones

FizzBuzz Refactoring

<https://github.com/emilybache/FizzBuzz-Refactoring-Kata>

Queremos añadir las siguientes reglas:

7 o múltiplo de 7 -> Whizz

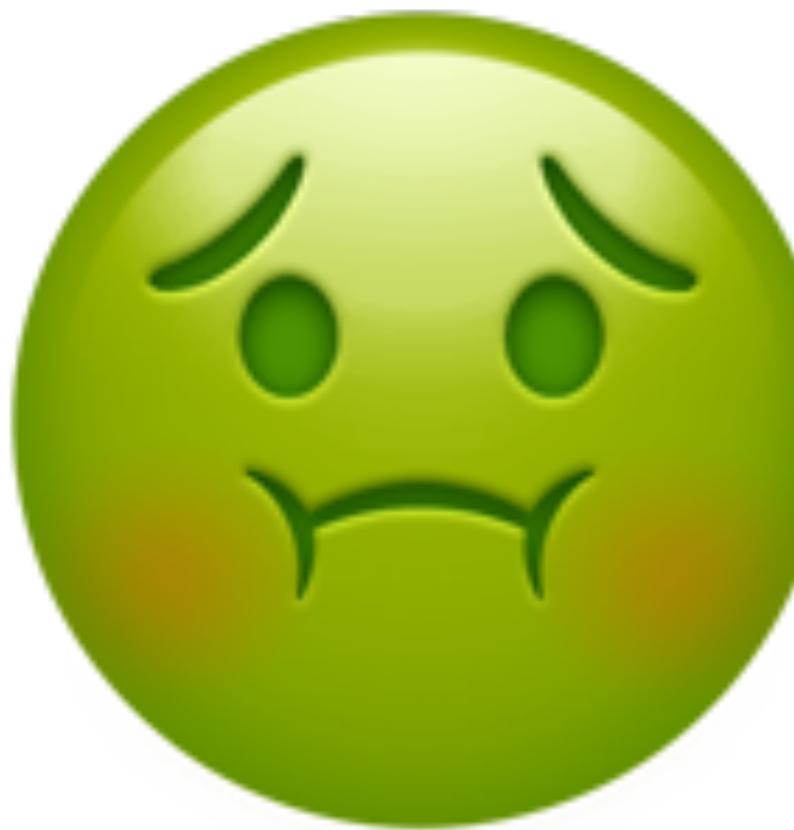
11 o múltiplo de 11 -> Bang

Refactorizar este código para que añadir la nueva función sea muy fácil. "Haz el cambio fácil, luego haz el cambio fácil".

Bloque 3

Code Smells

Patrones en el código que
revelan problemas en el
diseño



¿Qué es un code smell?



Síntoma



Identificación



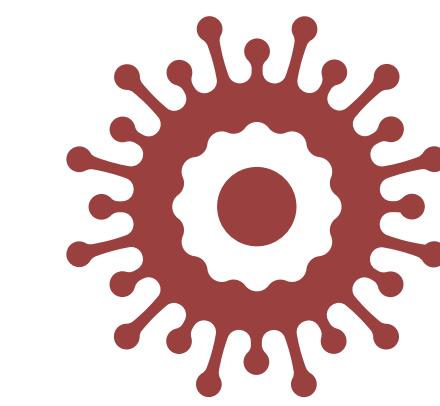
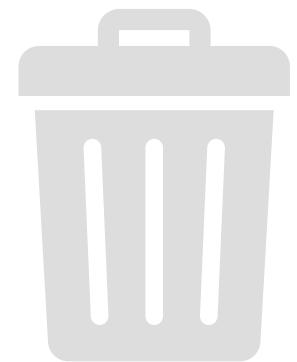
Refactor

Un Code Smell es un patrón en el código que es el síntoma de un problema de diseño.

El código funciona correctamente, pero resulta difícil de entender o de cambiar.

Identificar code smells nos permite entender si un código necesita intervención y cuál.

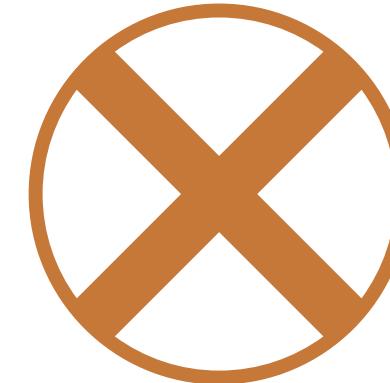
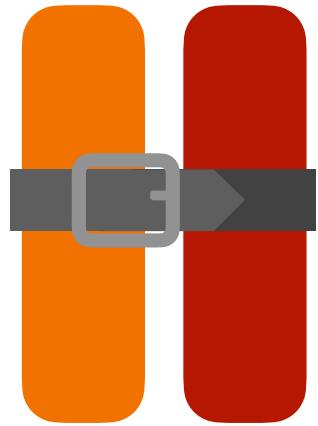
Categorías



Rebosantes

Desechables

Abusadores



Acopladores

Anti-cambios



Rebosantes

Unidades de código han crecido tanto que se hace difícil trabajar con ellas

Método Largo

Método con más de un cierto número de líneas no cohesivas (10)

Clase Gigante

Clase con muchos métodos, que pueden ser largos, o muchas propiedades

Agrupación de Datos

Datos que siempre van juntos y se utilizan juntos

Obsesión Primitiva

Representar conceptos con tipos primitivos en lugar de introducir propios

Lista Larga de Parámetros

Pasar más de 3 ó 4 parámetros a un método

Primitive obsession

```
const VAT_RATE=1.21
function applyVAT(price: number): number {
  return price * VAT_RATE
}

const email:string = 'fran@example.com'
const ADMIN = 1
const CUSTOMER = 2
```

Uso de tipos
primitivos para
codificar información,
simular tipos...

Nos obliga a validar
constantemente lo
que contienen

Primitive obsession

```
class Price {  
    private amount: number  
  
    constructor(amount: number) {  
        this.amount = amount;  
    }  
  
    applyVAT(rate: number = VAT_RATE): Price {  
        return new Price(this.amount * rate)  
    }  
}
```

Introducir Value Objects

Cada tipo es responsable de mantener sus propias invariantes

Proporcionan comportamientos propios que podemos usar donde sea necesario

Método largo

```
start() {  
    const win = False  
    const player_moves = []  
    const computer_moves = []  
    const attempt = []  
    const turns = 0  
    const tic_tac_screen = [[[], [], []],  
                           [[], [], []],  
                           [[], [], []]]  
    const list_1 = [5]  
    const list_2 = [1, 3]...  
    // 300 more lines...  
    list_4 = [1, 7]  
    list_5 = [1, 3, 7, 9]  
    list_6 = [3, 9]  
    list_7 = [5]  
    list_8 = [7, 9]  
    list_9 = [5]  
    winning_combinations = {(1, 2, 3), (4, 5, 6), (7, 8, 9), (1, 4, 7), (2, 5, 8), (3, 6, 9), (1, 5, 9),  
                           (3, 5, 7)}  
    def comp():  
        computer_check = str(computer_moves[-1])  
        if "1" in computer_check:  
            tic_tac_screen[0][0] = "o"  
        elif "2" in computer_check:  
            tic_tac_screen[0][1] = "o"  
        elif "3" in computer_check:  
            tic_tac_screen[0][2] = "o"  
        elif "4" in computer_check:  
            tic_tac_screen[1][0] = "o"  
        elif "5" in computer_check:  
            tic_tac_screen[1][1] = "o"  
        elif "6" in computer_check:  
            tic_tac_screen[1][2] = "o"  
        elif "7" in computer_check:  
            tic_}  
}
```

Una función o método que tiene muchas líneas de código. Por ejemplo, más de 10.

Posiblemente el método tiene más de una responsabilidad.

Extraer métodos

Extraer clase

Long parameter list

```
function processExamScore(  
    courseId: string,  
    examScore: number,  
    isHomeworkComplete: boolean,  
    attendanceScore: number,  
    bonusActivities: BonusActivity[],  
    isRetake: boolean,  
    scoreWeights: ScoreWeights,  
    coursePolicy: CoursePolicy,  
    // ... some more parameters  
): ExamResult {  
// ... code
```

Una función o método recibe más de 3 ó 4 parámetros

Posibilidad de introducir errores por posición o tipos

Introducir Objeto Parámetro

Introducir Value Objects

Clase grande

Una clase tiene
muchos métodos,
propiedades o líneas
de código

Muchas
responsabilidades y
resulta complejo
entender como
funciona

Extraer clases

Data clump

```
const name: string = 'Fran';
const surname: string = 'Iglesias';

const user = new User(name, surname);

const longitude: number = 45;
const latitude: number = -15;

const location = getLocation(
  longitude,
  latitude
);
```

Uno o varios datos que van siempre juntos y se encuentran en distintos sitios del código

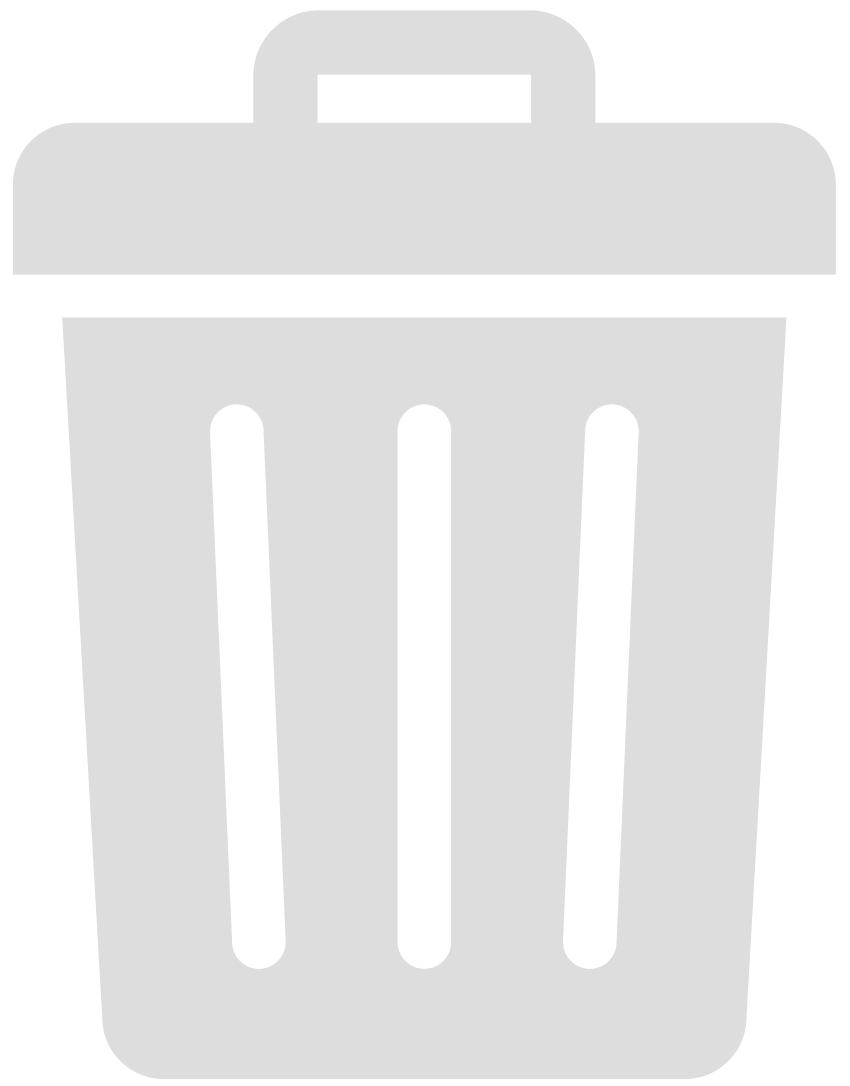
Nos está diciendo que hay un concepto pidiendo aparecer

Data clump

```
class PersonName {  
  constructor(public name: string, public surname: string) {}  
  
  fullName(): string {  
    return `${this.name} ${this.surname}`;  
  }  
}  
  
const user = new User(new PersonName('Fran',  
  'Iglesias'));  
  
class Coordinates {  
  constructor(public longitude: number, public  
  latitude: number) {}  
}  
  
const longitude: number = 45;  
const latitude: number = -15;  
  
const location = getLocation(  
  new Coordinates(longitude, latitude)  
);
```

Introducir Value Objects

Los Value Objects representan mejor los conceptos, protegen sus propias invariantes y atraen comportamiento



Desechables

Unidades de código que han perdido su razón de ser y sería mejor quitarlas.

Comentarios

Muchos comentarios que no aportan valor, incluyendo código comentado

Código duplicado

Fragments de código muy similares haciendo lo mismo en distintos lugares

Código muerto

Código que nunca se ejecuta

Clase perezosa

Clase que solo delega en otras, sin ninguna razón aparente

Clase de datos

Clase que no tiene comportamientos y solo aporta datos a otras

Comentarios

```
start() {  
    // Signals if the game has a winner  
    let win = False  
    // Stores the player_moves  
    let player_moves = []  
    // Stores the computer_moves  
    let computer_moves = []  
    // Stores the attempts  
    let attempt = []  
    // Manages the turns  
    let turns = 0  
    // Represents the game board in...screen  
    const tic_tac_screen = [[ "", "", "" ],  
                           [ "", "", "" ],  
                           [ "", "", "" ]]  
    //...  
}
```

El código está lleno de comentarios explicando cosas

Riesgo de que los comentarios queden desactualizados o que el código no se expresa bien.

Comentarios

```
start() {  
  let gameHasWinner = False  
  let player_moves = []  
  let computer_moves = []  
  let attempts = []  
  let turns = 0  
  const tic_tac_board = [[ "", "", "" ],  
    [ "", "", "" ],  
    [ "", "", "" ]]  
  //...  
}
```

El código está lleno de comentarios explicando cosas

Riesgo de que los comentarios queden desactualizados o que el código no se expresa bien.

Quitar comentarios

Renombrar

Extraer métodos

Data Class

```
class Discount {  
    private rate = 0.1;  
  
    constructor(rate: number) {  
        this.rate = rate;  
    }  
  
    getRate() {  
        return this.rate;  
    }  
}
```

Una clase solo tiene propiedades y accesores, pero no tiene comportamiento propio.

O bien está pidiendo comportamiento o bien es innecesaria.

Data Class

```
class Discount {  
    private rate = 0.1;  
  
    constructor(rate: number) {  
        this.rate = rate;  
    }  
  
    apply(base: number): number {  
        return base * (1 - this.rate);  
    }  
}
```

Introducir métodos

Proporciona un comportamiento que le es propio y que es útil a otros

Quitar la clase

Si no tiene sentido que tenga su propio comportamiento, posiblemente sobra

Código duplicado

```
export type Product = { price: number; qty: number }

// First duplicate
export function discountedTotalA(items: Product[], discountRate = 0.1): number {
  let subtotal = 0
  for (const it of items) {
    subtotal += it.price * it.qty
  }
  const discount = subtotal * discountRate
  return subtotal - discount
}

// Second duplicate (same logic, slightly different variable names)
export function discountedTotalB(products: Product[], discountRate = 0.1): number {
  let sum = 0
  for (const p of products) {
    sum += p.price * p.qty
  }
  const discount = sum * discountRate
  return sum - discount
}
```

Dos ó más fragmentos de código son idénticos o casi idénticos

Posibilidad de olvidar alguna copia y obtener comportamientos incoherentes

Código duplicado

```
export type Product = { price: number; qty: number }

export function discountedTotalA(items: Product[], discountRate = 0.1): number {
  return calculateDiscountedTotal(items, discountRate)
}

export function discountedTotalB(products: Product[], discountRate = 0.1): number {
  return calculateDiscountedTotal(products,
discountRate);
}

function calculateDiscountedTotal(products: Product[], discountRate: number) {
  let sum = 0
  for (const p of products) {
    sum += p.price * p.qty
  }
  const discount = sum * discountRate
  return sum - discount
}
```

Extraer método

Extraemos la lógica común y la usamos en los viejos métodos hasta que podemos dejar de usarlos

Lazy class

```
export type Address = {
  name: string;
  line1: string;
  city?: string
}

export class ShippingLabelBuilder {
  build(a: Address): string {
    return `${a.name} - ${a.line1}${a.city ? `,
" + a.city : ""}`
  }
}
```

Una clase que básicamente delega en otras clases todo lo que tenga que hacer.

Lazy class

```
export class Address {  
    name: string  
    line1: string  
    city?: string  
  
    constructor(name: string, line1: string,  
    city?: string) {  
        this.name = name  
        this.line1 = line1  
        this.city = city  
    }  
  
    shippingLabel(): string {  
        return `${this.name} - ${this.line1}$  
{this.city ? ", " + this.city : ""}`  
    }  
}
```

Quitar la clase

Es una clase menos que mantener y además unifica comportamientos en torno a ella

Dead code

```
findTheWinner(): winner {  
    if (this.player1.win()) {  
        return this.player1  
    }  
    if (this.player2.win()) {  
        return this.player2  
    }  
  
    const score1 = this.player1.score()  
    const score2 = this.player2.score()  
    if (score1 > score2) {  
        return this.player1  
    }  
    return this.player2  
}
```

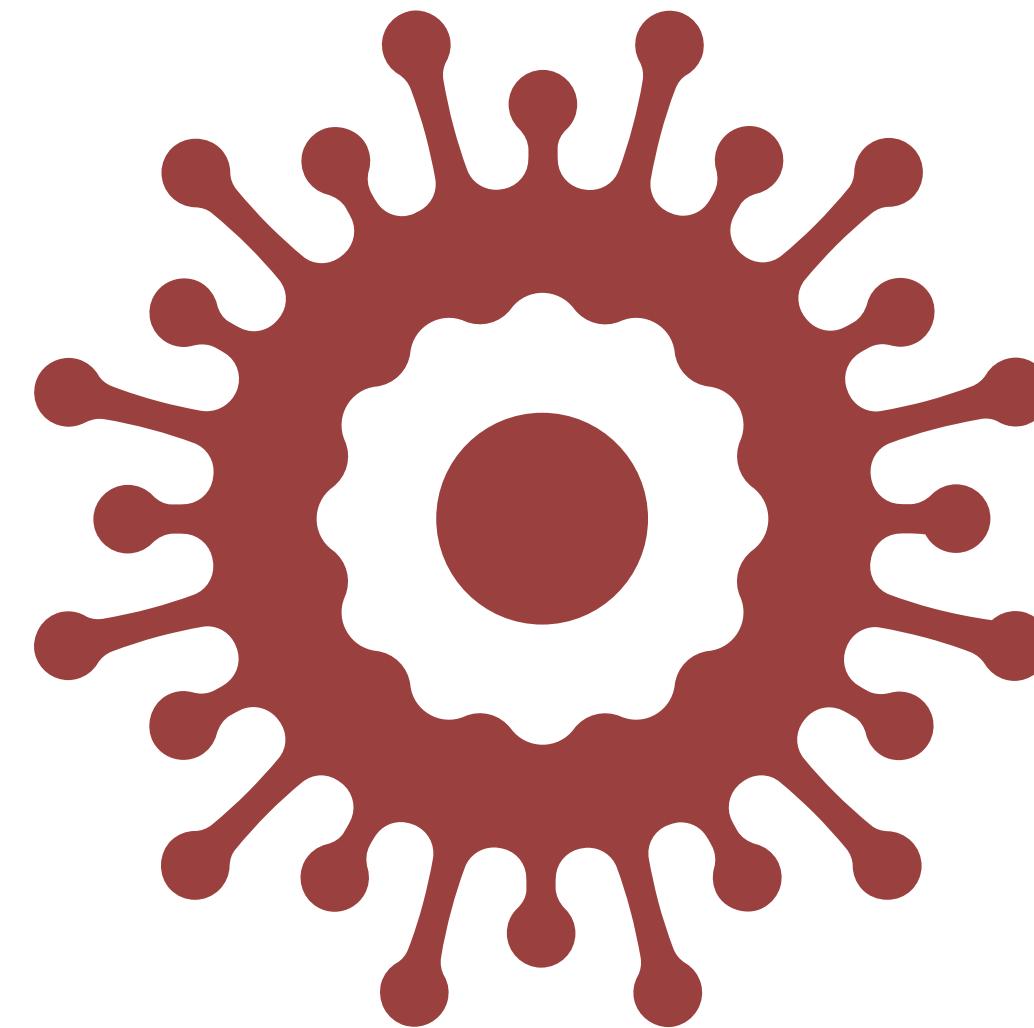
Código que nunca se ejecuta

Dead code

```
findTheWinner(): winner {  
    if (this.player1.win()) {  
        return this.player1  
    }  
    return this.player2  
}
```

Quitar el código que no se usa

El código muerto solo nos introduce ruido y desorden



Abusadores de la OOP

Consecuencia de aplicar de forma incorrecta los principios de la orientación a objetos

Clases alternativas dif. interfaz

Dos clases que tienen un rol similar pero cuya interfaz es diferente

Legado rechazado

Clase hija hereda métodos que no usa

Switch

Usualmente un método que cambia su comportamiento en función de una propiedad

Campo temporal

Solo se usa para llevar un valor a de un método a otro

Clases alternativas con diferente interfaz

```
export class TextLogger {
  log(message: string): void {
    console.log(`[text] ${message}`)
  }
}

export class MessageWriter {
  write(entry: string): void {
    console.log(`[text] ${entry}`)
  }
}

export function useAltClasses(choice: 'logger' | 'writer', msg: string): void {
  if (choice === 'logger') {
    new TextLogger().log(msg)
  } else {
    new MessageWriter().write(msg)
  }
}
```

Dos clases que tienen un rol similar pero cuya interfaz es diferente

Implica una duplicación de comportamiento que dificulta los cambios.

Clases alternativas con diferente interfaz

```
export class TextLogger {
  log(message: string): void {
    console.log(`[text] ${message}`)
  }
}

export class MessageWriter {
  write(entry: string): void {
    new TextLogger().log(entry)
  }
}
```

Dos clases que tienen un rol similar pero cuya interfaz es diferente

Introducir Delegación

Refused bequest

```
class BaseController {
  start(): void {
    console.log('starting')
  }

  stop(): void {
    console.log('stopping')
  }

  reset(): void {
    console.log('resetting')
  }
}

export class ReadOnlyController extends BaseController {
  start(): void {
  }

  stop(): void {
  }
}

export function demoRefusedBequest(readonly: boolean): void {
  const controller: BaseController = readonly ? new
  ReadOnlyController() : new BaseController()
  controller.start()
  controller.stop()
}
```

Clases que descienden de otras, pero que no usan los métodos heredados.

Nos obliga a saber qué soporta cada clase que podemos necesitar usar.

Refused bequest

```
interface Resettable {
  reset(): void
}

interface Controller {
  start(): void

  stop(): void
}

class BaseController implements Controller, Resettable {
  start(): void {
    console.log('starting')
  }

  stop(): void {
    console.log('stopping')
  }

  reset(): void {
    console.log('resetting')
  }
}

export class ReadOnlyController implements Controller {
  start(): void {
  }

  stop(): void {
  }
}

export function demoRefusedBequest(readonly: boolean): void {
  const controller: Controller = readonly ? new ReadOnlyController() : new BaseController()
  controller.start()
  controller.stop()
}
```

Clases que descienden de otras, pero que no usan los métodos heredados.

Nos obliga a saber qué soporta cada clase que podemos necesitar usar.

Introducir Interfaces más estrechas

Switch statements

```
export type EmployeeKind = 'engineer' | 'manager' | 'sales'

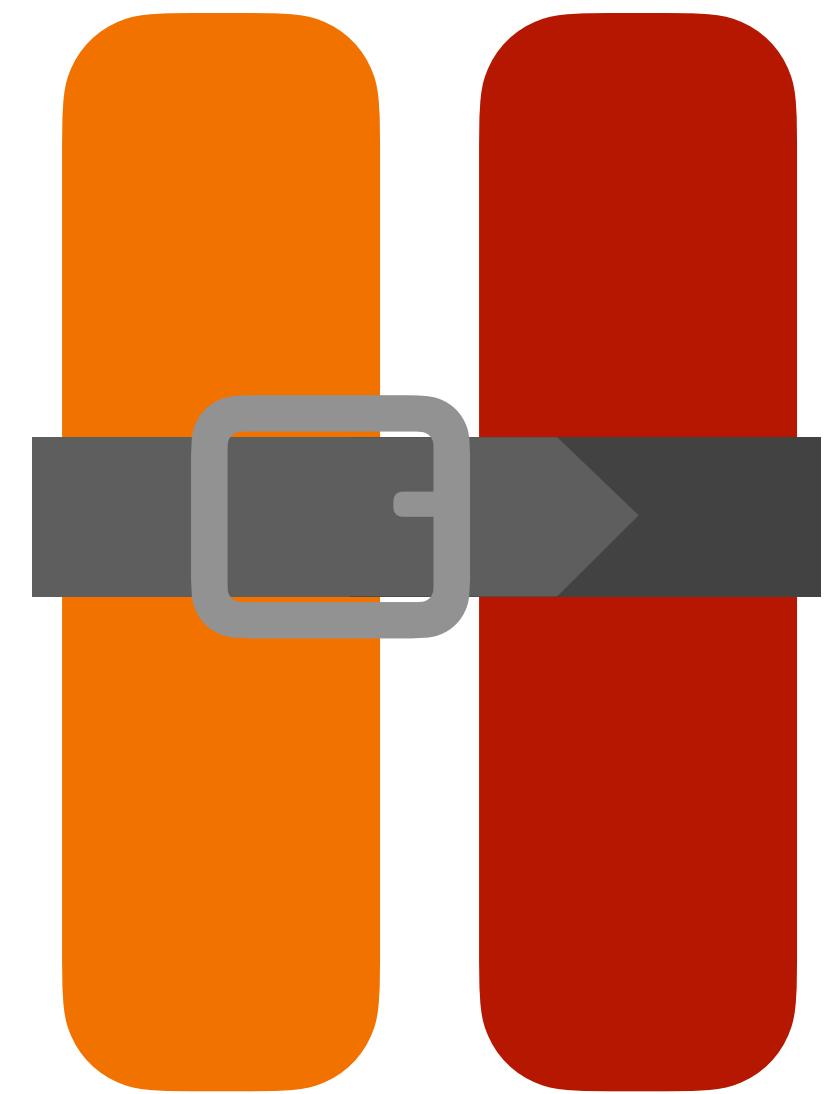
export interface EmployeeRecord {
  kind: EmployeeKind
  base: number
  bonus?: number
  commission?: number
}

export function calculatePay(rec: EmployeeRecord): number {
  switch (rec.kind) {
    case 'engineer':
      return rec.base
    case 'manager':
      return rec.base + (rec.bonus ?? 0)
    case 'sales':
      return rec.base + (rec.commission ?? 0)
    default:
      const _exhaustive: never = rec.kind
      return _exhaustive
  }
}

export function demoSwitchStatements(): number[] {
  return [
    calculatePay({kind: 'engineer', base: 1000}),
    calculatePay({kind: 'manager', base: 1000, bonus: 200}),
    calculatePay({kind: 'sales', base: 800, commission: 500}),
  ]
}
```

Cuando el comportamiento del objeto depende de una propiedad

Por lo general, nos está diciendo que podemos aplicar polimorfismo.



Acopladores

Cuando los cambios de un objeto afectan a otros

Envidia de características

Un objeto usa más cosas de otro que de las suyas propias

Intimidad inapropiada

Un objeto accede directamente a propiedades de otro

Cadenas de mensajes

Pedir algo a un objeto que está dentro de otro, que está dentro de otro, que está dentro de otro

Hombre interpuesto

Hace una cosa, delegando en otra clase

Feature Envy

```
class Discount {  
    private rate = 0.1;  
  
    constructor(rate: number) {  
        this.rate = rate;  
    }  
  
    getRate() {  
        return this.rate;  
    }  
}  
  
class Price {  
    base!: number;  
    final!: number;  
  
    applyDiscount(discount: Discount) {  
        const rate = discount.getRate();  
        const finalAmount = this.base * (1 - rate);  
        this.final = finalAmount;  
    }  
}
```

Un método accede más a la información de otro objeto que a la del suyo

Indica encapsulación o atribución de responsabilidades incorrecta

Feature Envy

```
class Discount {  
    private rate = 0.1;  
  
    constructor(rate: number) {  
        this.rate = rate;  
    }  
  
    apply(base: number): number {  
        return base * (1 - this.rate);  
    }  
}  
  
class Price {  
    base!: number;  
  
    constructor(base: number) {  
        this.base = base;  
    }  
  
    applyDiscount(discount: Discount): Price {  
        return new Price(discount.apply(this.base));  
    }  
}
```

Introducir métodos y mover el comportamiento a la otra clase

En lugar de pedir información, le pedimos un comportamiento al otro objeto

Message Chain

```
class Product {
    private price: number = 10
    private name: string = 'A product'

    getPrice(): number {
        return this.price
    }
}

class Row {
    private product: Product = new Product()

    getProduct(): Product {
        return this.product
    }
}

class Order {
    private rows: Row[] = []

    getRow(rowNumber: number): Row | undefined {
        return this.rows[rowNumber]
    }
}

// The message chain happens here
const order = new Order()
const price = order
    .getRow(1)!
    .getProduct()
    .getPrice()
```

Se hace una llamada a un objeto, que hemos obtenido de hacer una llamada o otro objeto, el cual hemos obtenido de...

Indica encapsulación incorrecta.

Message Chain

```
class Product {
    private price: number = 10
    private name: string = 'A product'

    getPrice(): number {
        return this.price
    }
}

class Row {
    private product: Product = new Product()

    getProduct(): Product {
        return this.product
    }
}

class Order {
    private rows: Row[] = []

    private getRow(rowNumber: number): Row | undefined {
        return this.rows[rowNumber]
    }

    getPrice(rowNumber: number) {
        return this.getRow(rowNumber)?.getProduct().getPrice()
    }
}

const order = new Order()
const price = order.getPrice(1)
```

Ocultar delegado

En algunos casos nos basta con exponer un método en el objeto con el que interactuamos

Introducir métodos

Muchas veces, se trata de un problema de diseño más profundo



Anti cambios

Hacen difícil cambiar el código porque ha de hacerse en muchos lugares

Cambio divergente

Una clase tiene muchas razones para cambiar

Jerarquía paralela

Si hago un cambio en una jerarquía también tengo que hacerlo en la otra

Cirugía a tiros

Para aplicar un cambio tengo que hacerlo en muchos sitios del código

Shotgun surgery

```
export type Item = { price: number; qty: number }

export function cartTotal(items: Item[]): number {
  const subtotal = items.reduce((s, i) => s + i.price
* i.qty, 0)
  const vat = subtotal * 0.21 // duplicated logic
  return subtotal + vat
}

export function printReceipt(items: Item[]): string {
  const sum = items.reduce((s, i) => s + i.price *
i.qty, 0)
  const total = sum + sum * 0.21 // duplicated logic
  return `TOTAL: ${total.toFixed(2)}`
}

export function loyaltyPoints(items: Item[]): number {
  const base = items.reduce((s, i) => s + i.price *
i.qty, 0)
  const withTax = base + base * 0.21 // duplicated
logic
  return Math.floor(withTax / 10)
}
```

Para hacer una modificación tienes que aplicar cambios en muchos lugares del código

Posibilidad de olvidar alguna parte y obtener comportamientos incoherentes

Shotgun surgery

```
export type Item = { price: number; qty: number }

export type CartTotals = { subtotal: number; vat: number; total: number }

export function CalculateCartTotals(items: Item[]) {
  let subtotal = items.reduce((s, i) => s + i.price * i.qty, 0);
  let vat = subtotal * 0.21;
  return {
    subtotal,
    vat,
    total: subtotal + vat
  }
}

export function cartTotal(items: Item[]): number {
  const {total} = CalculateCartTotals(items)
  return total
}

export function printReceipt(items: Item[]): string {
  const {total} = CalculateCartTotals(items)
  return `TOTAL: ${total.toFixed(2)}`
}

export function loyaltyPoints(items: Item[]): number {
  const {total} = CalculateCartTotals(items)
  return Math.floor(total / 10)
}
```

Extraer clases

Separamos la lógica común de la particular, dejando una fuente de verdad para cada concepto

Cada usuario de esa lógica hace lo que necesite con el resultado

Cambio divergente

```
export type Product = { sku: string; name: string; quantity: number }

export class InventoryCoordinator {
  private store = new Map<string, Product>()

  add(p: Product): void {
    if (!/^SKU-[A-Z0-9]{4,}$/.test(p.sku)) throw new Error('invalid sku')
    if (!p.name.trim()) throw new Error('invalid name')
    if (!Number.isInteger(p.quantity) || p.quantity < 0) throw new
Error('invalid quantity')
    this.store.set(p.sku, p)
  }

  exportCsv(): string {
    const rows =
['sku,name,quantity', ...Array.from(this.store.values()).map((p) => `
${p.sku},${p.name},${p.quantity}`)]
    return rows.join('\n')
  }

  lowStock(threshold = 5): string[] {
    const msgs: string[] = []
    for (const p of this.store.values()) {
      if (p.quantity <= threshold) {
        msgs.push(`LOW: ${p.name} (${p.sku}) = ${p.quantity}`)
      }
    }
    return msgs
  }
}
```

Cuando una clase tiene muchas razones para cambiar

Posibilidad de obtener comportamientos incoherentes

Cambio divergente

```
export class InventoryCoordinator {
  private store = new Map<string, Product>()

  add(p: Product): void {
    this.store.set(p.skuValue, p)
  }

  exportCsv(): string {
    const rows =
      ['sku, name, quantity', ...Array.from(this.store.values())].
      map((p) => `${p.csvReport()}`)
    return rows.join('\n')
  }

  lowStock(threshold = 5): string[] {
    return Array.from(this.store.values()).reduce((msgs: string[], p) => {
      if (p.lowStock(threshold)) {
        msgs.push(`LOW: ${p.stockReport()}`)
      }
      return msgs
    }, [])
  }
}
```

Extraer clases

Cada motivo de cambio nos indica que debería haber un lugar distinto para cada cosa (normalmente una clase)

Este ejemplo no está completo...

Cambio divergente

```
class Sku {
  value: string;

  constructor(value: string) {
    this.value = value;
  }

  static createValid(value: string): Sku {
    if (!/^SKU-[A-Z0-9]{4,}$/.test(value)) throw new Error('invalid sku')

    return new Sku(value)
  }
}

class ProductName {
  value: string;

  constructor(value: string) {
    this.value = value;
  }

  static createValid(value: string): ProductName {
    if (!value.trim()) throw new Error('invalid name')
    return new ProductName(value)
  }
}

class Stock {
  value: number;

  constructor(value: number) {
    this.value = value;
  }

  static initiateWith(value: number): Stock {
    if (!Number.isInteger(value) || value < 0) throw new Error('invalid quantity')
    return new Stock(value)
  }

  low(threshold = 5): boolean {
    return this.value <= threshold
  }
}
```

Extraer clases

Hemos extraído Value Objects que se han llevado su propia lógica

Smelly Tic-Tac-Toe

<https://github.com/exeal-es/smelly-tic-tac-toe-kata>

1. Identifica los code smells
2. Aplica refactors para resolverlos

- Primitive obsession
- Feature envy
- Data class
- Message chain
- Long method
- Comments
- Long parameter list
- Shotgun surgery
- Duplicated code
- Large class
- Divergent change
- Data clump
- Lazy class
- Dead code

Code smell catalog

<https://luzkan.github.io/smells/>

Filters

Expanses

Between

Within

Obstruction

Bloaters

Change Preventers

Couplers

Data Dealers

Dispensables

@

☰

Bloater - Responsibility

Combinatorial Explosion

The Combinatorial Explosion occurs when a lot of code does almost the same thing - here, the word "almost" is crucial. The number of cases...

“ ” ↗ ⓘ

Bloater - Measured Smells

Long Method

One of the most apparent complications developers can encounter in the code is the length of a method. The more lines of code a function has, the more...

“ ” ↗ ⓘ

Bloater - Data

Data Clump

Data Clumps refer to a situation in which a few variables are passed around many times in the codebase instead of being packed into a separate object...

“ ” ↗ ⓘ

Bloater - Measured Smells

Large Class

When one combines the smell of Long Method and Long Parameter List, but on a higher abstraction level, then he would get the Large Class code smell. Many...

“ ” ↗ ⓘ

Bloater - Conditional Logic

Null Check

Null check is widespread everywhere because the programming languages allow it. It causes a multitude of or checks everywhere: in guard checks, in...

“ ” ↗ ⓘ

Bloater - Responsibility

Required Setup or Clean-up Code

Bloque 4

Técnicas avanzadas

CAMBIO PARALELO

Expandir, migrar, contraer

Expandir

Escribe código nuevo en lugar de cambiar el que existe.
Añade tests o aplica TDD

Migrar

Depreca el código a reemplazar.
Empieza a usar el código nuevo siempre que tengas oportunidad.

Contraer

Una vez que todo el código ha sido migrado, elimina el código deprecado, sus tests y otros restos

Demo

<https://github.com/unclejamal/parallel-change>

Usando Parallel Change, modifica la clase ShoppingCart para poder manejar varios elementos en lugar de uno solo.

Los tests existentes tienen que pasar siempre.

Escribe nuevos tests si lo consideras necesario.



Ejercicio

```
export class User {
  constructor(
    public readonly id: string,
    public readonly name: string,
    public readonly email: string,
  ) {
  }
}

// --- Consumers of User.name ---

export function formatGreeting(user: User): string {
  return `Hello, ${user.name}!`
}

export function formatEmailHeader(user: User): string {
  return `From: ${user.name} <${user.email}>`
}

export function formatDisplayName(user: User): string {
  return `${user.name} (${user.id})`
}

export function buildUserSummary(users: User[]): string {
  return users.map((u) => `- ${u.name}`).join('\n')
}
```

TESTS DE CARACTERIZACIÓN

Tests de Caracterización

Son los tests que escribimos para describir el comportamiento de código que ya existe, pero que no entendemos.

Estos tests tienen el objetivo de describir o caracterizar el comportamiento actual del código. De este modo, podemos refactorizarlo con seguridad.

Golden Master

Es una técnica en la que creamos un gran número de tests que prueban combinaciones de valores de parámetros que el código acepte.

Capturamos el resultado y lo guardamos para usarlo como criterio durante el proceso de refactor.

Ejercicio

```
export class ReceiptPrinter {
    // Do not change this function at the beginning of the exercise; first create the Golden Master.
    print(order: Order): string {
        const now = new Date(Date.now())

        const header = `Recibo ${order.id} - ${now.toLocaleDateString()} ${now.toLocaleTimeString()}`

        let subtotal = 0
        let lines = order.items.map((it, idx) => {
            const lineTotal = round(it.unitPrice * it.quantity)
            subtotal = round(subtotal + lineTotal)
            return `${idx + 1}. ${it.description} (${it.sku}) x${it.quantity} = ${lineTotal.toFixed(2)}`
        })

        let luckyDiscountPct = 0
        if (Math.random() < 0.1) {
            luckyDiscountPct = Math.random() * 0.05
        }
        const luckyDiscount = round(subtotal * luckyDiscountPct)

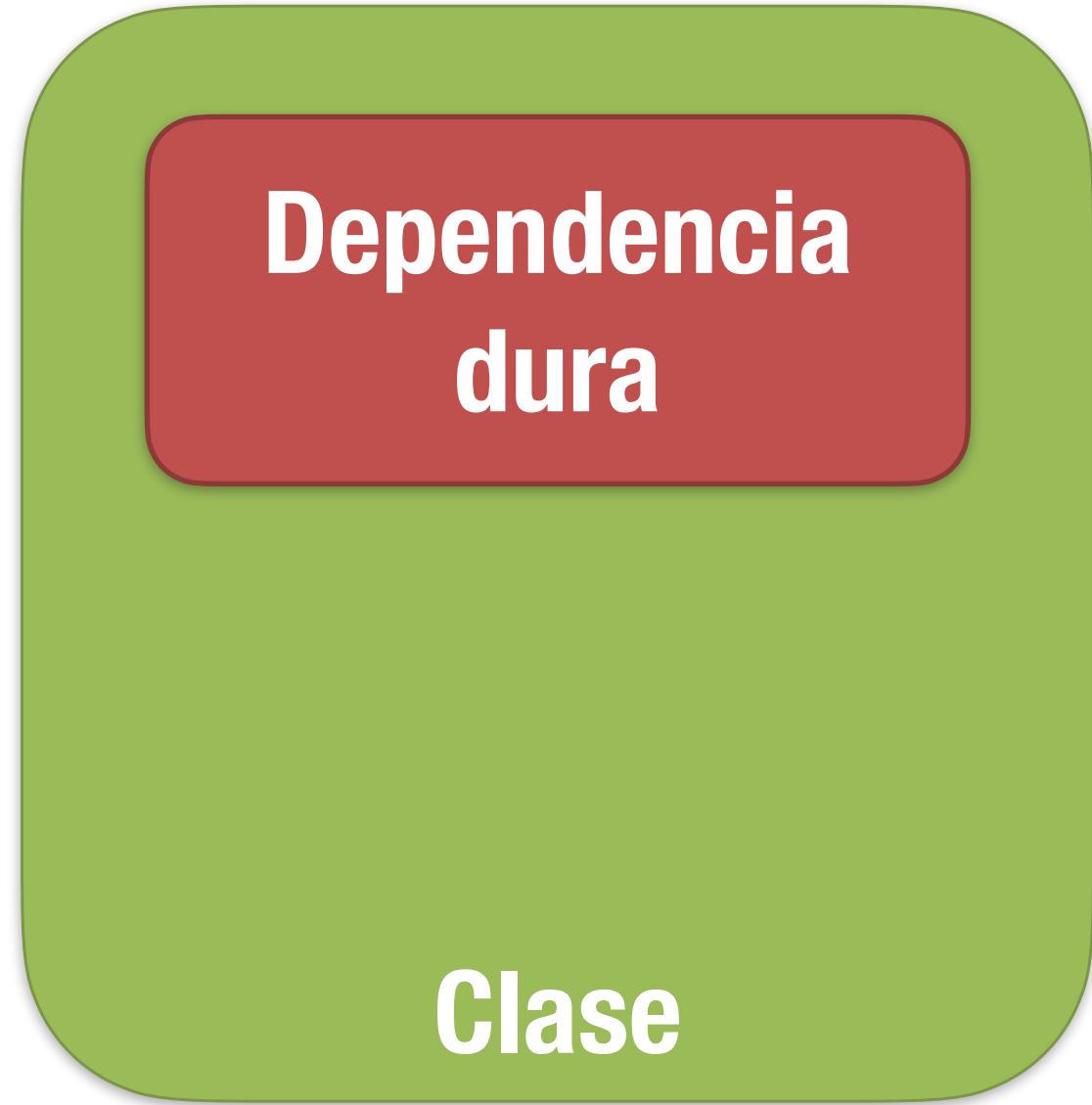
        const taxableGeneral = order.items
            .filter((i) => i.category !== 'books')
            .reduce((s, i) => s + (i.category === 'food' ? 0 : i.unitPrice * i.quantity), 0)
        const foodTax = order.items
            .filter((i) => i.category === 'food')
            .reduce((s, i) => s + i.unitPrice * i.quantity * 0.03, 0)
        const generalTax = taxableGeneral * 0.07
        const taxes = round(generalTax + foodTax)

        const total = round(subtotal - luckyDiscount + taxes)

        const summary = [
            `Subtotal: ${subtotal.toFixed(2)}`,
            luckyDiscount > 0
                ? `Descuento de la suerte: -${luckyDiscount.toFixed(2)} (${(luckyDiscountPct * 100).toFixed(2)}%)`
                : `Descuento de la suerte: $0.00 (0.00%)`,
            `Impuestos: ${taxes.toFixed(2)}`,
            `TOTAL: ${total.toFixed(2)}`,
        ]
        return [header, ...lines, '---', ...summary].join('\n')
    }
}
```

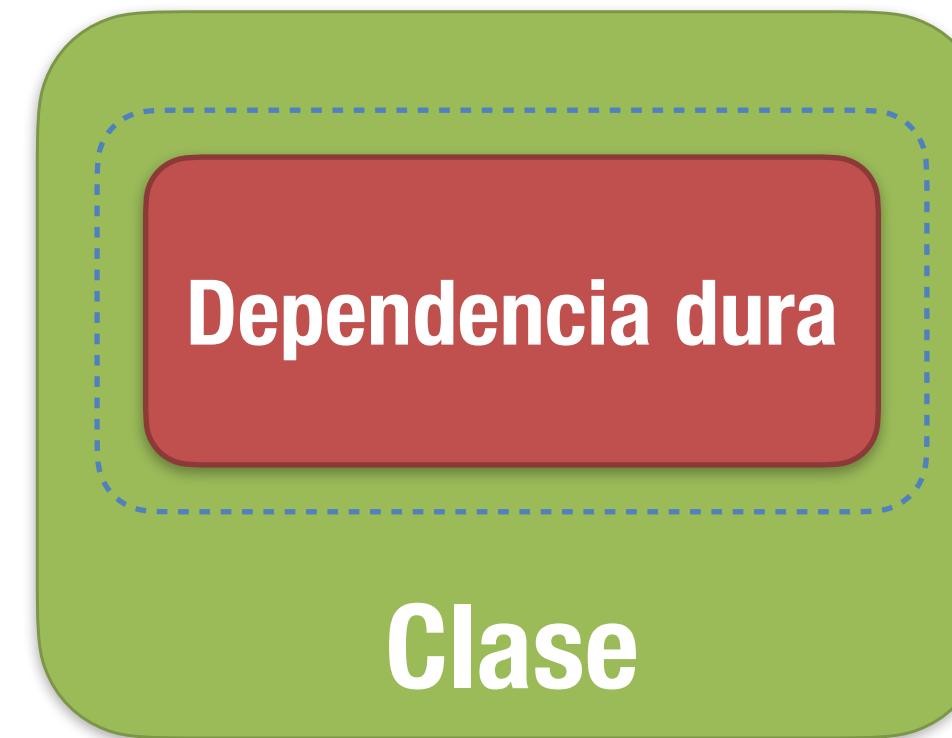
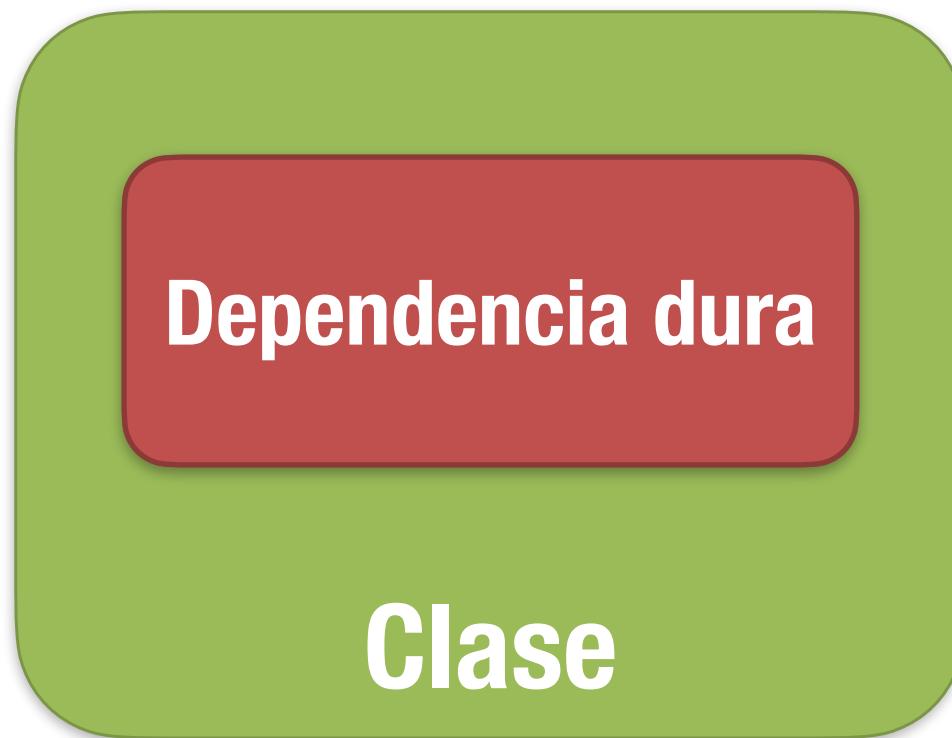
*ROMPER DEPENDENCIAS
DURAS*

Seams o Costuras



Una dependencia dura es un fragmento de código que impide que una clase se pueda poner bajo test.

Un *seam* o *costura* es un lugar en el código que nos permite cambiar ese código de una manera controlada para poder introducir tests.



Identificar las dependencias que hacen la clase no testable

Seam: Aislar ese fragmento de código en un método `protected`

Extender la clase sobre escribiendo el `seam`
Usar esta Subclase en los tests
Invertir la dependencia

Ejercicio

```
export class ReceiptPrinter {
    // Do not change this function at the beginning of the exercise; first create the Golden Master.
    print(order: Order): string {
        const now = new Date(Date.now())

        const header = `Recibo ${order.id} - ${now.toLocaleDateString()} ${now.toLocaleTimeString()}`

        let subtotal = 0
        let lines = order.items.map((it, idx) => {
            const lineTotal = round(it.unitPrice * it.quantity)
            subtotal = round(subtotal + lineTotal)
            return `${idx + 1}. ${it.description} (${it.sku}) x${it.quantity} = ${lineTotal.toFixed(2)}`
        })

        let luckyDiscountPct = 0
        if (Math.random() < 0.1) {
            luckyDiscountPct = Math.random() * 0.05
        }
        const luckyDiscount = round(subtotal * luckyDiscountPct)

        const taxableGeneral = order.items
            .filter((i) => i.category !== 'books')
            .reduce((s, i) => s + (i.category === 'food' ? 0 : i.unitPrice * i.quantity), 0)
        const foodTax = order.items
            .filter((i) => i.category === 'food')
            .reduce((s, i) => s + i.unitPrice * i.quantity * 0.03, 0)
        const generalTax = taxableGeneral * 0.07
        const taxes = round(generalTax + foodTax)

        const total = round(subtotal - luckyDiscount + taxes)

        const summary = [
            `Subtotal: ${subtotal.toFixed(2)}`,
            luckyDiscount > 0
                ? `Descuento de la suerte: -${luckyDiscount.toFixed(2)} (${(luckyDiscountPct * 100).toFixed(2)}%)`
                : `Descuento de la suerte: $0.00 (0.00%)`,
            `Impuestos: ${taxes.toFixed(2)}`,
            `TOTAL: ${total.toFixed(2)}`,
        ]
        return [header, ...lines, '---', ...summary].join('\n')
    }
}
```

SPROUT

Sprout o Injerto

Es una técnica para introducir código nuevo (bien construido y bien testado) en una pieza de software legacy, reemplazando la existente, pero manteniendo su uso hasta tener la seguridad de que podemos hacer el cambio.

Aplica a situaciones en las que queremos reemplazar un algoritmo o una dependencia completamente.

Sprout / Injerto

Escribe código nuevo que haga lo que necesitas.
Ponlo bajo test o desarrollalo con TDD.

Identifica desde donde deberías llamarlo en el código existente

Introduce las llamadas al nuevo código para reemplazar los usos del viejo

Código nuevo

Código existente

Código reemplazable

Código existente

Código nuevo

Ejercicio

Queremos introducir soporte de impuestos para otras regiones.

```
export function calculateTotal(cart: CartItem[], region: Region): number
{
  const subtotal = cart.reduce((s, it) => s + it.price * it.qty, 0)

  let tax = 0
  if (region === 'US') {
    tax = subtotal * 0.07 // 7% plano
  } else if (region === 'EU') {
    // exenciones ingenuas en línea
    const taxable = cart
      .filter((it) => it.category !== 'books' && it.category !== 'food')
      .reduce((s, it) => s + it.price * it.qty, 0)
    tax = taxable * 0.2 // 20% plano solo sobre los ítems gravables
  }

  return roundCurrency(subtotal + tax)
}
```

WRAP

Wrap o Envoltorio

Esta técnica consiste en aislar un código problemático, de forma que podamos mejorar o incluso reemplazar su uso, pero sin perder la interfaz.

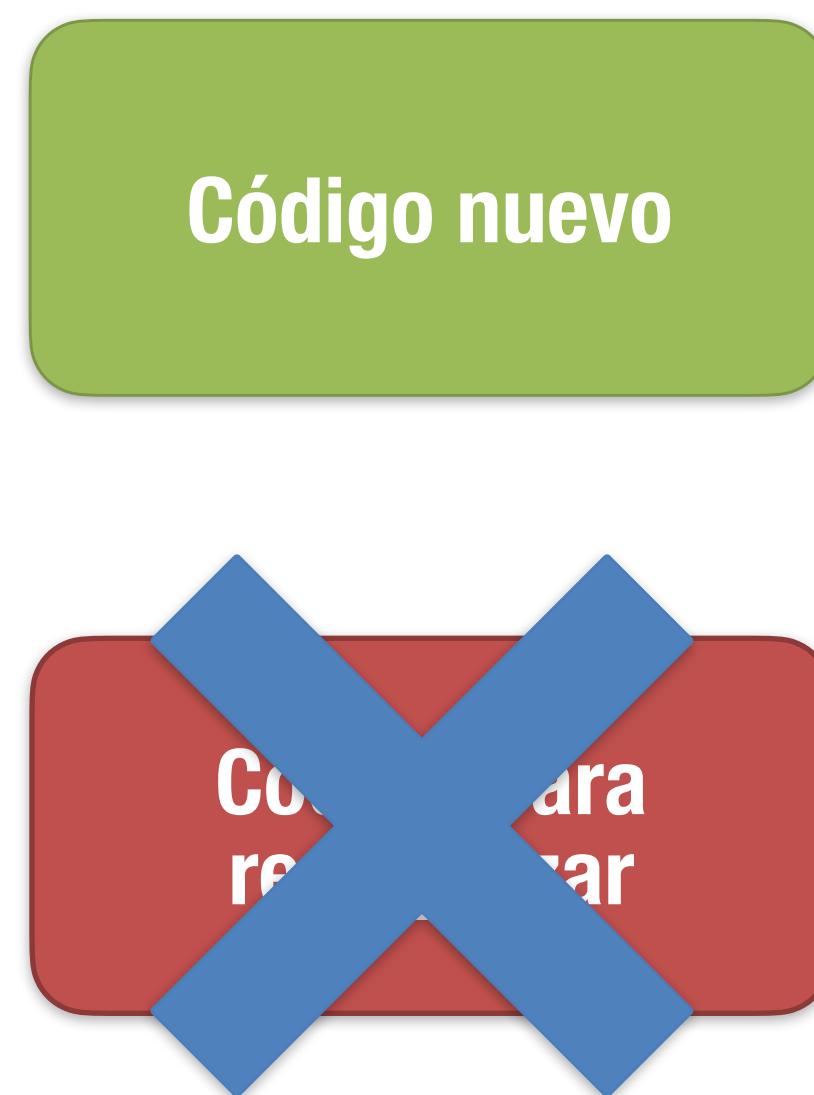
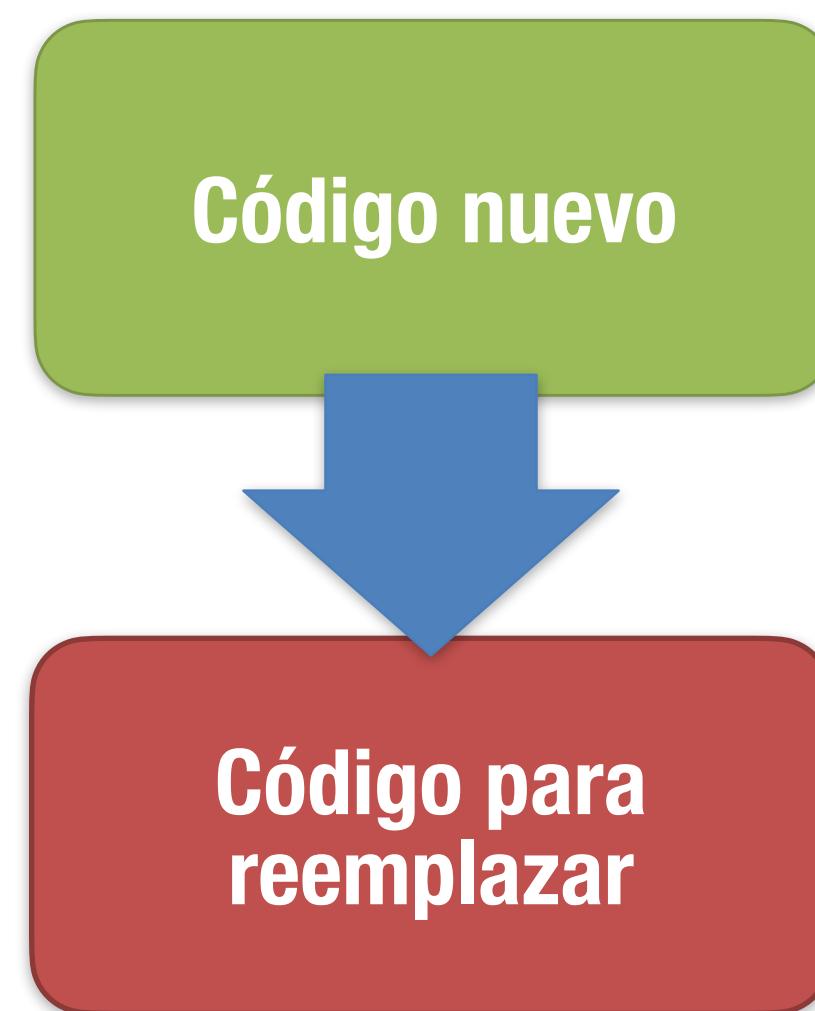
Aplica cuando queremos mejorar la funcionalidad de un código, pero no perder la compatibilidad con el uso que ya tenemos.

Wrap / Envoltorio

Aísla el código que quieras reemplazar en un método y/o cámbiale el nombre

Crea un nuevo método con el **mismo nombre y firma**. Llama al viejo desde el nuevo

Añade lógica antes o después de la llamada hasta dejar de necesitar el viejo (p.ej. *feature flag*)



Ejercicio

Queremos añadir soporte de log, validación, sanitización, plantillas, etc.

```
export class LegacyEmailService {
  sendEmail(to: string, subject: string, body: string): string {
    return `Email sent to ${to}, subject: ${subject}, body: ${body}`
  }
}

// Código de nuestra aplicación que usa directamente el servicio legacy
const legacyService = new LegacyEmailService()

export function notifyWelcome(userEmail: string): string {
  return legacyService.sendEmail(userEmail, 'Welcome!', 'Thanks for joining our app.')
}

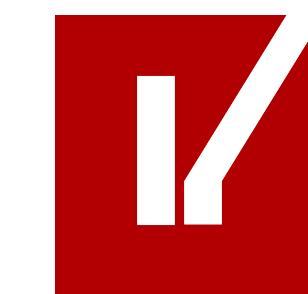
export function notifyPasswordReset(userEmail: string): string {
  return legacyService.sendEmail(userEmail, 'Reset your password', 'Click the link to
reset...')
}

export function notifyOrderConfirmation(userEmail: string, orderId: string): string {
  return legacyService.sendEmail(
    userEmail,
    'Order Confirmation',
    `Your order ${orderId} has been confirmed.
  )
}
```

Refactoring Avanzado

por Fran Iglesias

Staff Software Engineer



Vitae