

Realizado por:
Andrés Felipe
Meneses Mafla



→

Presentación

REFACTORIZAR

Índice de CONTENIDOS



00. Introducción

01. Refactorizar, ¿Qué es?

02. Usos y ventajas de la
refactorizar

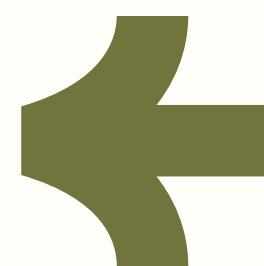
03. ¿Cuándo se debe
refactorizar?

04. ¿Cómo refactorizar?

05. Conclusión

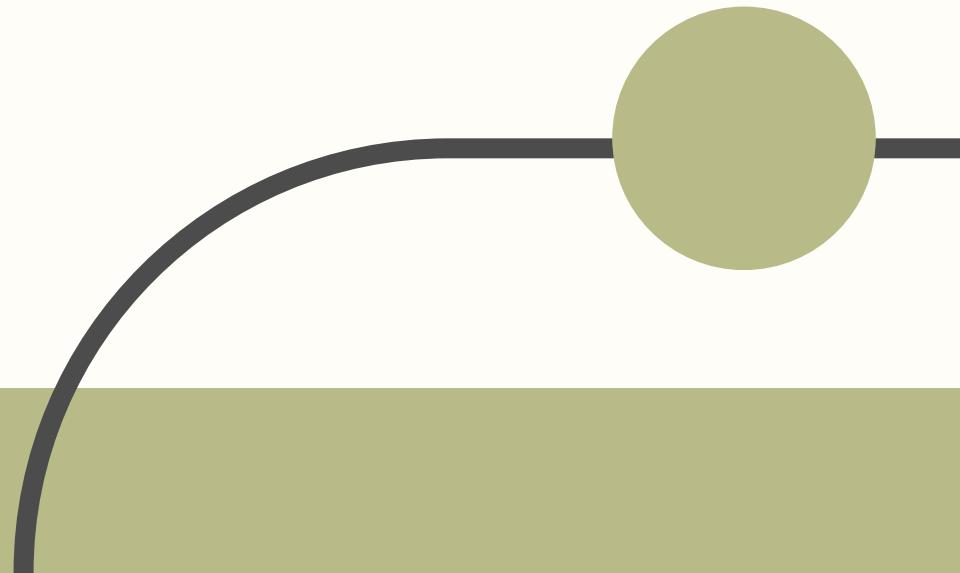


Introducción



En el desarrollo de software, la **refactorización** es un proceso clave que **mejora el rendimiento** del código **sin cambiar su funcionalidad**. A medida que los proyectos crecen, el código se vuelve más complejo y difícil de mantener, por lo que la **refactorización se vuelve esencial**.

Aquí exploraremos los principios, beneficios y usos, así como las mejores prácticas para lograr un código **limpio y eficiente**.





01

Refactorización:
Qué es?

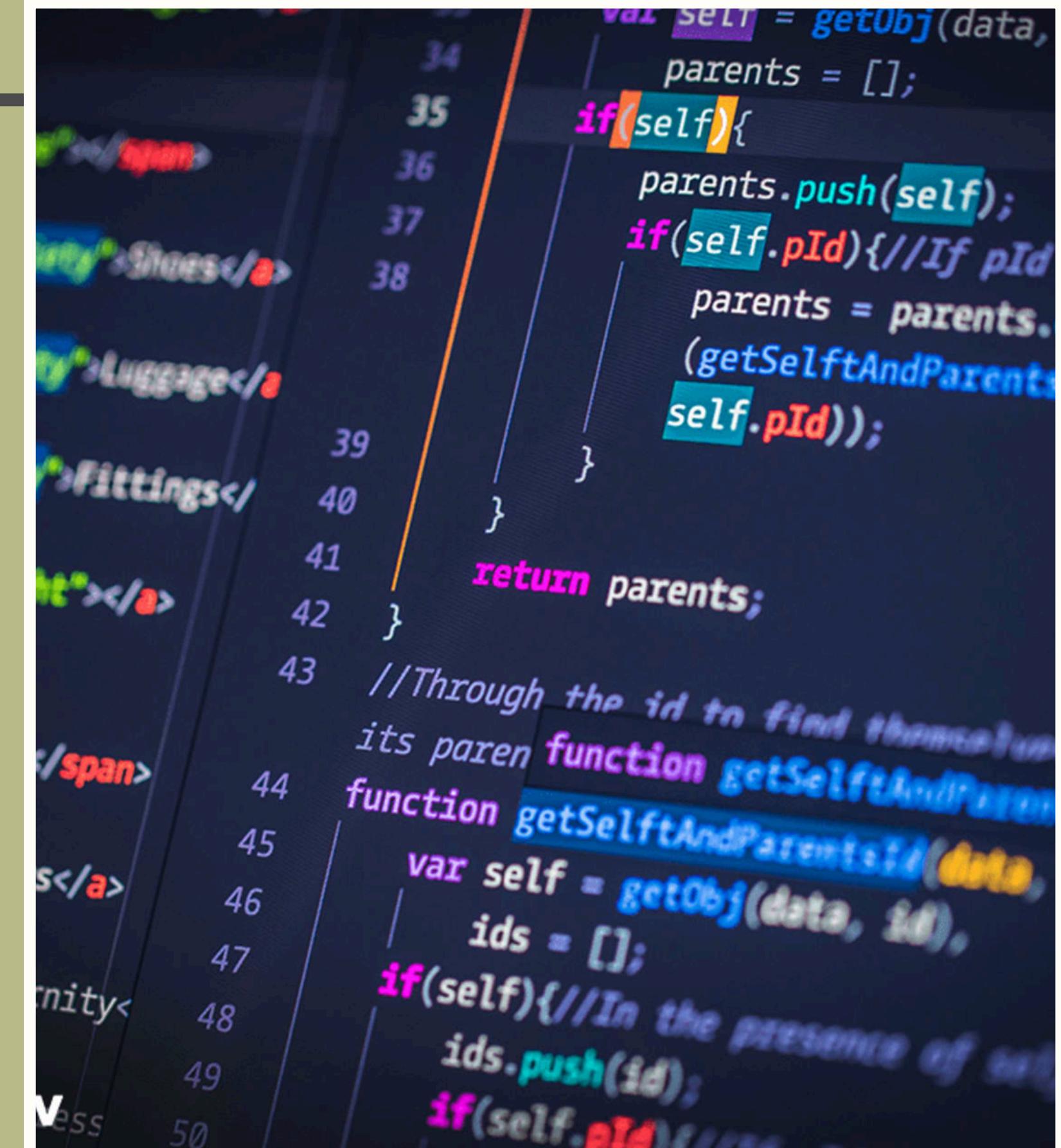
REFACTORIZAR

→ Proceso clave

La **refactorización** es el proceso mediante el cual se **modifica** el **código** de un programa para mejorar su estructura interna **sin cambiar su funcionalidad**, es decir, no debe cambiar lo que hace sino como lo hace.

Esto involucra **mejorar** el diseño, evitar repeticiones, optimizar rendimiento entre otras cosas de **nuestro código**.

El **objetivo principal** es hacer del código, **mas limpio, legible, eficiente y mantenible**.



The screenshot shows a code editor with a dark theme. On the left, there is some XML-like markup. On the right, there is a block of JavaScript code. The code defines a function that takes an object `data` and an optional parameter `id`. It uses a helper function `getObj` to get an object by its ID. If the object is found, it adds it to an array `parents` and then calls itself with the object's parent ID. This recursive call continues until no more parents are found. Finally, it returns the `parents` array. There are several annotations in the code: an orange line highlights the recursive call to the function; a blue line highlights the return statement; and a red line highlights the `self` variable in the recursive call. The code is numbered from 34 to 50.

```
var self = getObj(data, id);
parents = [];
if(self){
    parents.push(self);
    if(self.pId){ //If pId
        parents = parents.(getSelfAndParents(
            self.pId));
    }
}
return parents;
}

//Through the id to find the object
//its parent function getSelfAndParents
function getSelfAndParents(id,data) {
    var self = getObj(data, id);
    ids = [];
    if(self){ //In the presence of an object
        ids.push(id);
        if(self.pId)
            ids.push(self.pId);
        getSelfAndParents(self.pId,data);
    }
}
```



02

Refactorización:
Usos
Ventajas



VENTAJAS

01 Código mas Limpio

Esto facilita que tu código se mas **fácil de trabajar** cn el

02 Mejor Rendimiento

Al tener una **buenas estructura**, indirectamente resulta en una **mayor eficiencia** a la hora de usar los recursos

03 Facilitar testing y debugging

Es mas **fácil realizar test** y depurar el código pues, esta mejor organizado y modulado.

04 Reducir la deuda técnica

Al reducir atajos en el desarrollo, reduce esta deuda y es mas **factible** I realizar **expansiones** y **cambios** en un futuro

Usos

→ MEJORAR LEGIBILIDAD

El código legible es más **comprendible** y **facil de modificar**, un código limpio **reduce** la complejidad

→ OPTIMIZAR RENDIMIENTO

Al refactorizar puede **mejorar el rendimiento** al eliminar ciclos innecesarios o **reducir lo complejo** que sea nuestro algoritmo

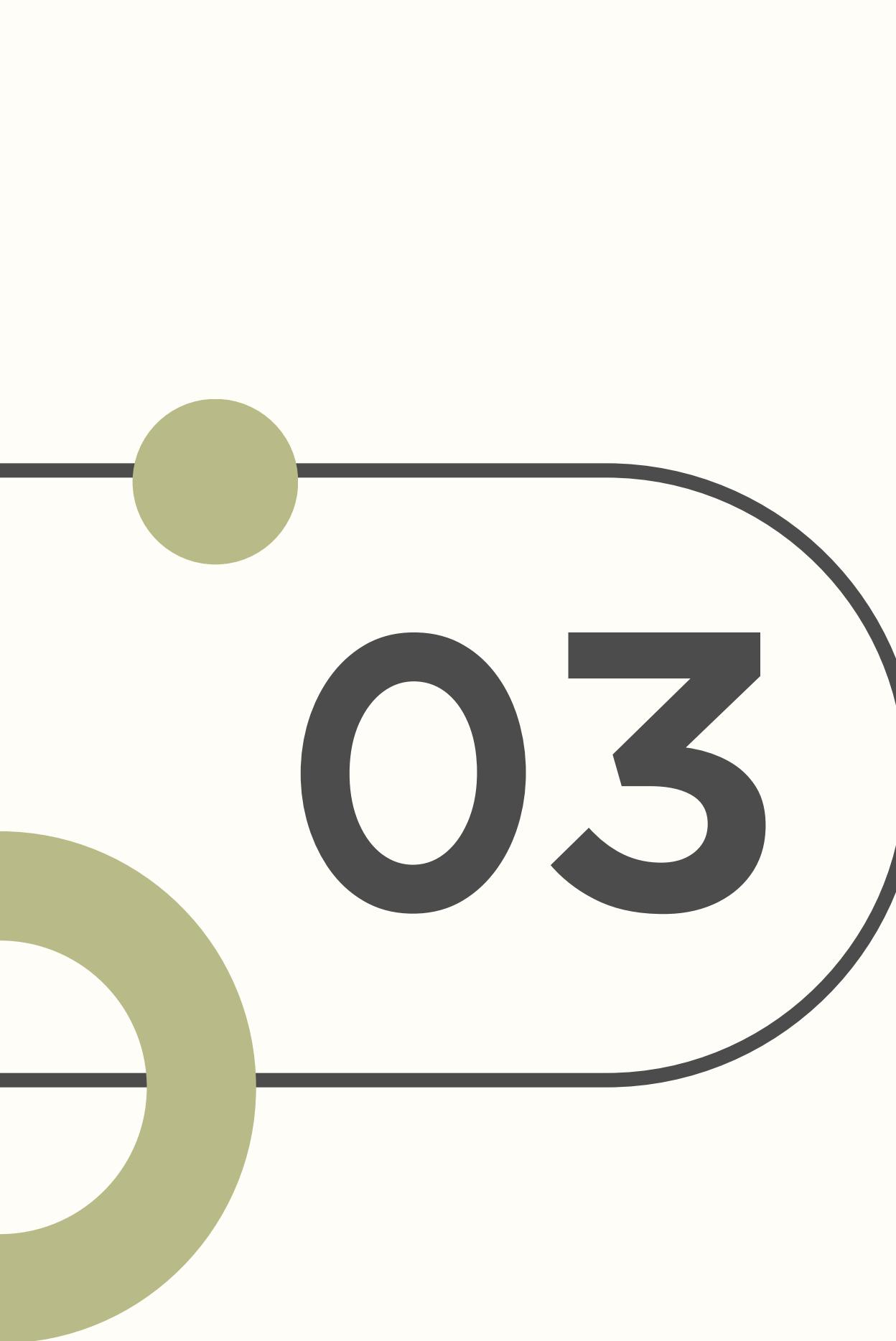
MEJORAR CALIDAD

→

Ayuda a **detectar fallos** lógicos en nuestro código así como mejorar el diseño , **evitando problemas** futuros

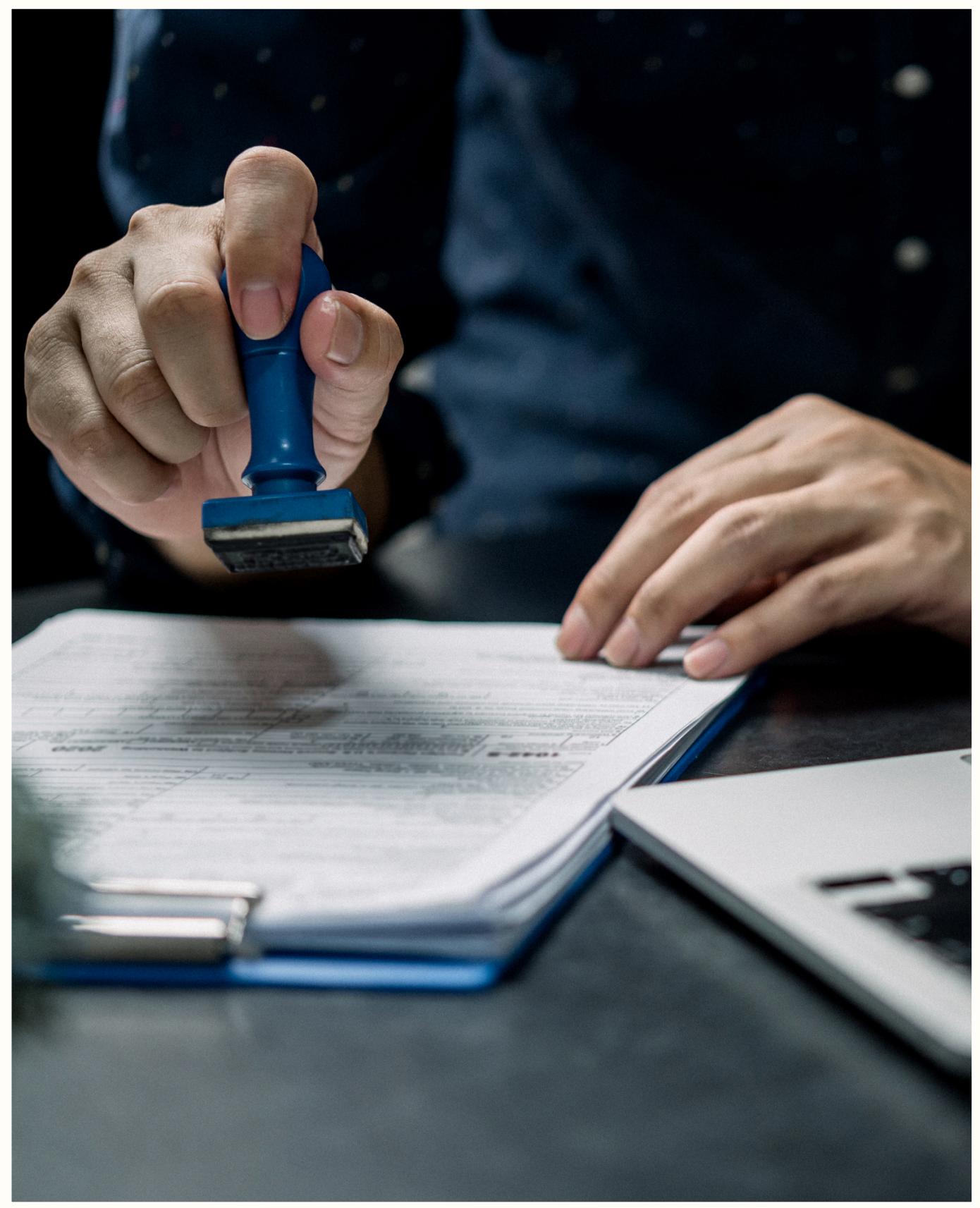
→ FACILITAR MANTENIMIENTO

El código que este bien refactorizado, es más **fácil de actualizar y probar**, además de poder **añadir funcionalidades sin perjudicar el comportamiento** que tiene nuestro programa



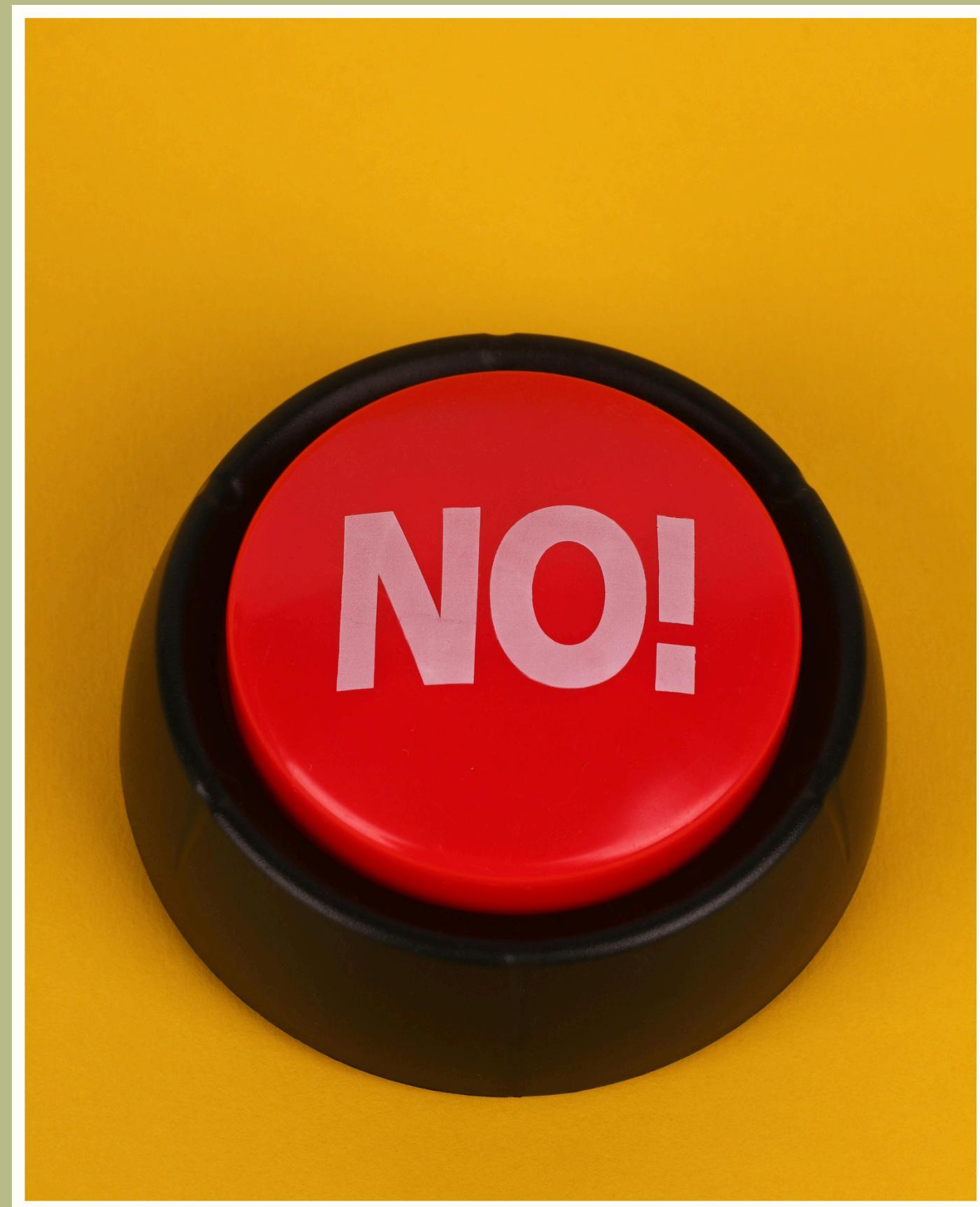
03

Refactorización:
Cuando
realizarla?



REALIZARLA CUANDO EL CODIGO...

- 01 Es difícil de entender o mantener
- 02 Contiene duplicación de código
- 03 Tiene problemas de calidad o malas prácticas
- 04 necesita cambios o nuevas funcionalidades que agregar



NO REALIZARLA CUANDO EL CODIGO...

- 01 No tiene pruebas automatizadas
- 02 Funciona bien y no está causando problemas
- 03 Tiene plazos estrictos o poco tiempo



04

Refactorización: Cómo refactorizar?

Pasos a seguir



PREPARACION

Es fundamental tener una **buenas cobertura de pruebas** antes de comenzar, debemos asegurarnos de ello para garantizar el funcionamiento posterior del código

ANALISIS

Vemos el código y anotamos donde podremos realizar la refactorización, **siempre en bloques pequeños** para evitar riesgos

IDEAS

Pensaremos maneras de **mejorar el código** de baja calidad usando las **técnicas de refactorización**, también podemos pedir opinión a otros sobre que se podría realizar, **diferentes perspectivas dan mejores resultados.**

REFACTORIZAR

Una vez ya **tenemos claro** que vamos a cambiar y como, procedemos a realizar la mejora de código, **siempre en bloques pequeños**

Técnicas

→ RENOMBRAR

Si nuestras **variables**, **clases** o **métodos** tienen nombres no descriptivos, se pueden **renombrar** para que sean **mas claros o estén acordes** al tema del proyecto

→ DIVISION

Si en el código vemos clases o métodos que **violan el principio de responsabilidad única**, se pueden **dividir** en clases o métodos mas pequeños

REEMPLAZAR BUCLES

Al trabajar con colecciones, es normal trabajar con bucles for, en vez de eso, podremos **usar Streams** para hacer el código **mas conciso**.

→ EXTRACCION DE METODOS

Si nuestro código tiene fragmentos **repetidos** o **muy largos** es aconsejable **extraer** ese código en un **método independiente**, esto mejora la claridad y la reutilización.

RENOMBRAR VARIABLES

→ Motivo de los cambios

Como vemos arriba, las variables “l” y “w” no son lo suficientemente descriptivas por si solas.

Realizamos el cambio de nombres a “length” y “width” para ofrecer una mayor claridad.

```
public class Rectangle {  
    private double l;  
    private double w;  
  
    public double calculateArea() {  
        return l * w;  
    }  
}
```

💡

```
public class Rectangle {  
    private double length;  
    private double width;
```

```
public double calculateArea() {  
    return length * width;  
}
```

RENOMBRAR METODOS

→ Motivo de los cambios

Nuestro método “`saveData()`” tiene un nombre genérico o simple que no refleja bien la acción a realizar por este. Al cambiarlo por “`saveCustomerInfo()`”, ya tenemos una idea clara de lo que hace nuestro método y está más acorde al tema.



The screenshot shows a code editor window with a dark theme. At the top, there is a search bar with the placeholder "Search". Below the search bar, the file path "Customer.java 1 X" is displayed, followed by the file content:

```
Customer.java 1 X
ers > a23andresfmm > Desktop > Customer.java > ...
public class Customer {
    public void saveData() {
        // Logic to save customer data
    }
}

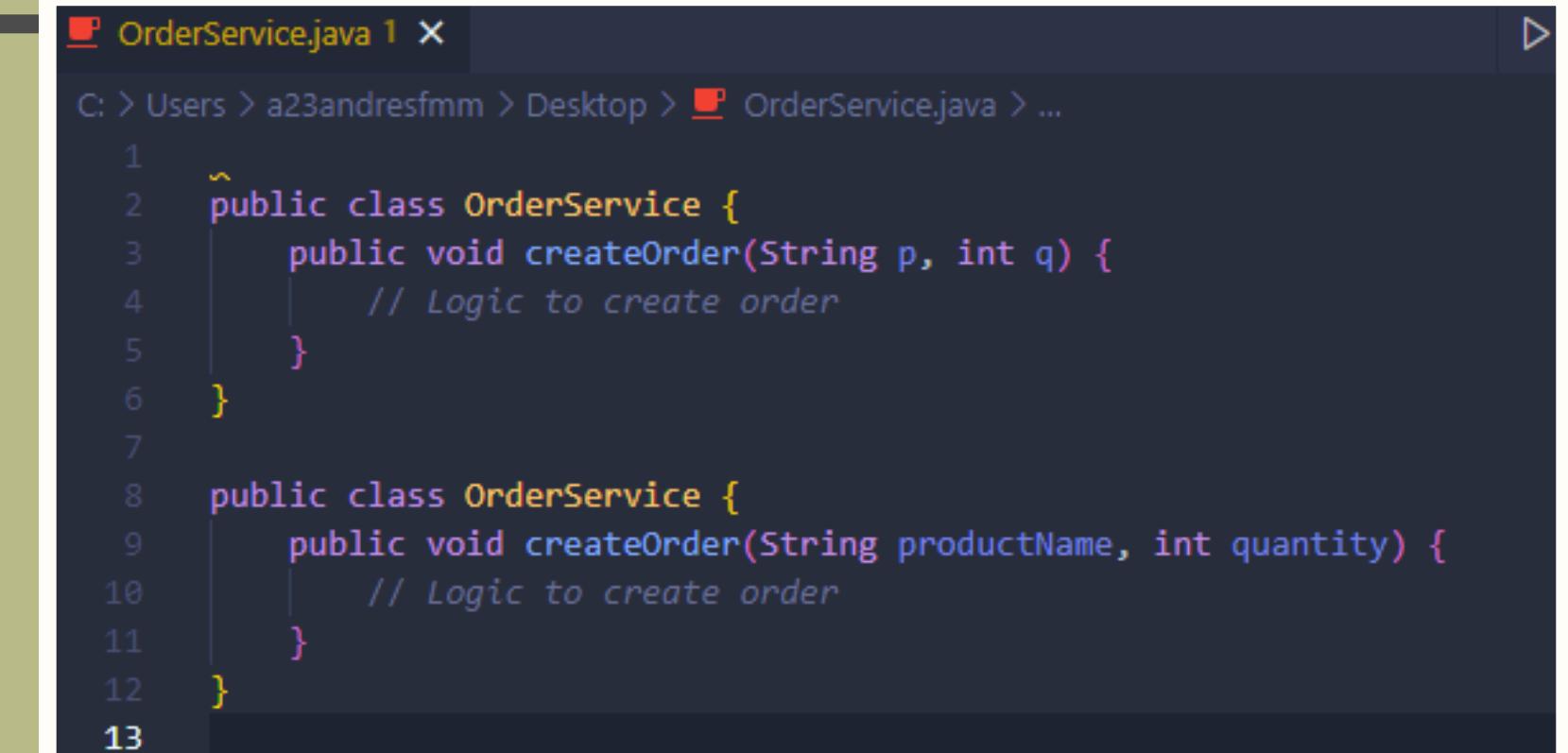
public class Customer {
    public void saveCustomerInfo() {
        // Logic to save customer data
    }
}
```

RENOMBRAR PARAMETROS

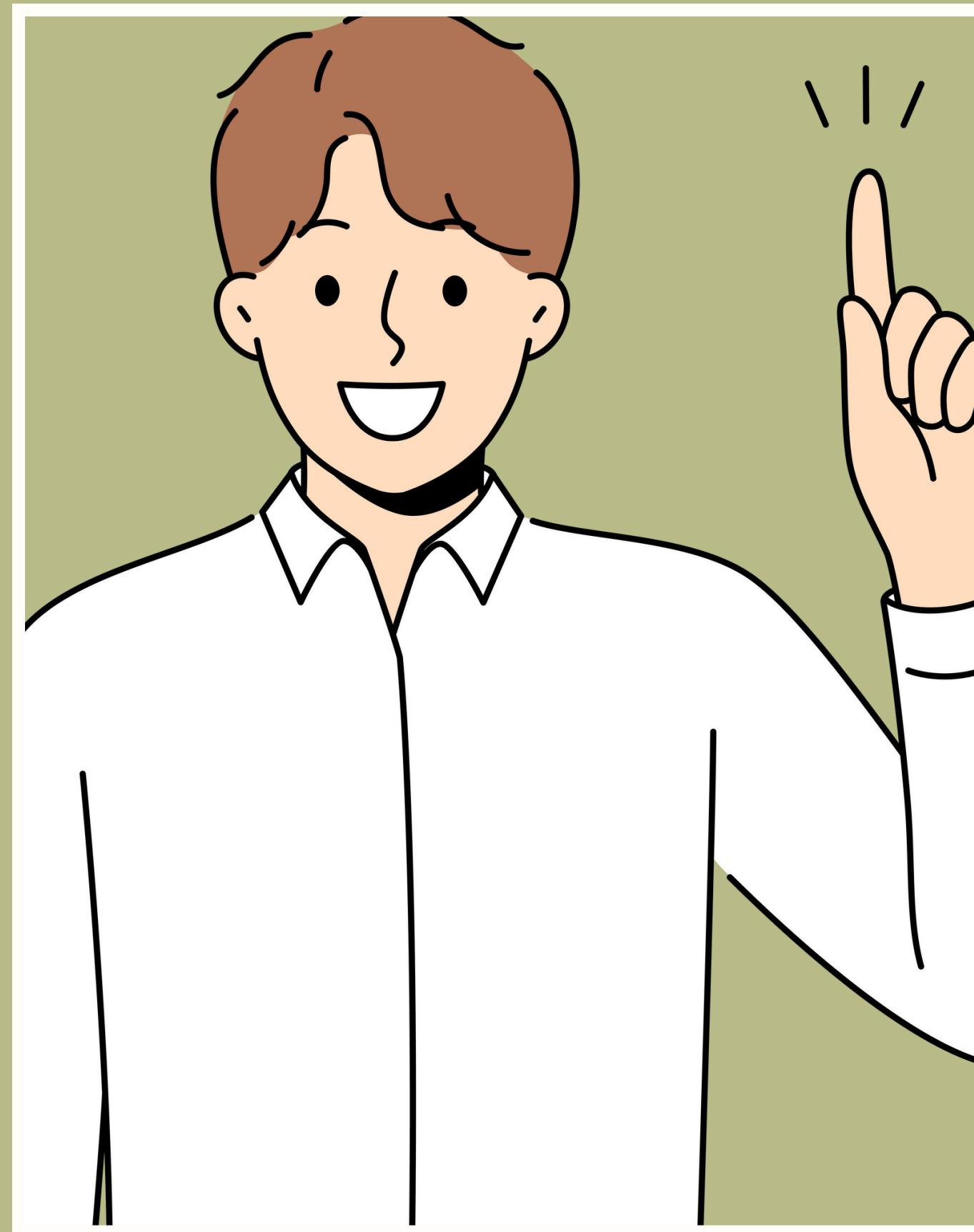
→ Motivo de los cambios

Aquí vemos que nuestros parámetros de entrada “p” y “q”, no nos indican correctamente que valores necesita el método para su correcto funcionamiento.

Al cambiarlos por “productName” y “quantity” indica claramente su propósito, y es intuitivo usar el metodo.



```
OrderService.java 1 X
C: > Users > a23andresfmm > Desktop > OrderService.java > ...
1
2 public class OrderService {
3     public void createOrder(String p, int q) {
4         // Logic to create order
5     }
6
7
8 public class OrderService {
9     public void createOrder(String productName, int quantity) {
10        // Logic to create order
11    }
12
13 }
```



CONSEJOS AL RENOMBRAR

01 Usar Herramientas del IDE

Las **herramientas** que proporciona un **IDE** como el de “**refactor**” o “**rename**” nos ayudan mucho en esta tarea, asegurando que todos las **coindicencias se actualicen** correctamente.

02 Mantener la consistencia

Es recomendable seguir las **convenciones de nomenclatura**, camelCase para métodos y variables, PascalCase para las clases y nombres **descriptivos**.

DIVISION METODOS

→ Motivo de los cambios

Nuestro método de generación de reportes, no es intuitivo. Al descomponerlo cada responsabilidad en métodos mas pequeños, tenemos un código mas comprensible y facilita la reutilización.

```
1 public class ReportGenerator {  
2     public void generateReport(String data) {  
3         System.out.println("Fetching data...");  
4         // Simulate fetching data  
5         String fetchedData = "Fetched: " + data;  
6  
7         System.out.println("Processing data...");  
8         // Simulate data processing  
9         String processedData = fetchedData.toUpperCase();  
10  
11        System.out.println("Saving report...");  
12        // Simulate saving report  
13        System.out.println("Report: " + processedData);  
14    }  
15 }  
16  
17 public class ReportGenerator {  
18     public void generateReport(String data) {  
19         String fetchedData = fetchData(data);  
20         String processedData = processData(fetchedData);  
21         saveReport(processedData);  
22     }  
23  
24     private String fetchData(String data) {  
25         System.out.println("Fetching data...");  
26         return "Fetched: " + data;  
27     }  
28  
29     private String processData(String data) {  
30         System.out.println("Processing data...");  
31         return data.toUpperCase();  
32     }  
33  
34     private void saveReport(String data) {  
35         System.out.println("Saving report...");  
36         System.out.println("Report: " + data);  
37     }  
38 }
```

DIVISION CLASES

→ Motivo de los cambios

Esta clase contiene demasiadas responsabilidades, violando así el principio de responsabilidad única. Al separarlas en diferentes clases, tenemos un código más accesible y coherente, facilitando la reutilización.

```
Users > a23andresfmm > Desktop > UserService.java > ...
public class UserManager {
    public void addUser(String name) {
        System.out.println("Adding user: " + name);
    }

    public void sendEmail(String email, String message) {
        System.out.println("Sending email to " + email + ": " + message);
    }

    public void logActivity(String activity) {
        System.out.println("Logging activity: " + activity);
    }
}

public class UserService {
    public void addUser(String name) {
        System.out.println("Adding user: " + name);
    }
}

public class EmailService {
    public void sendEmail(String email, String message) {
        System.out.println("Sending email to " + email + ": " + message);
    }
}

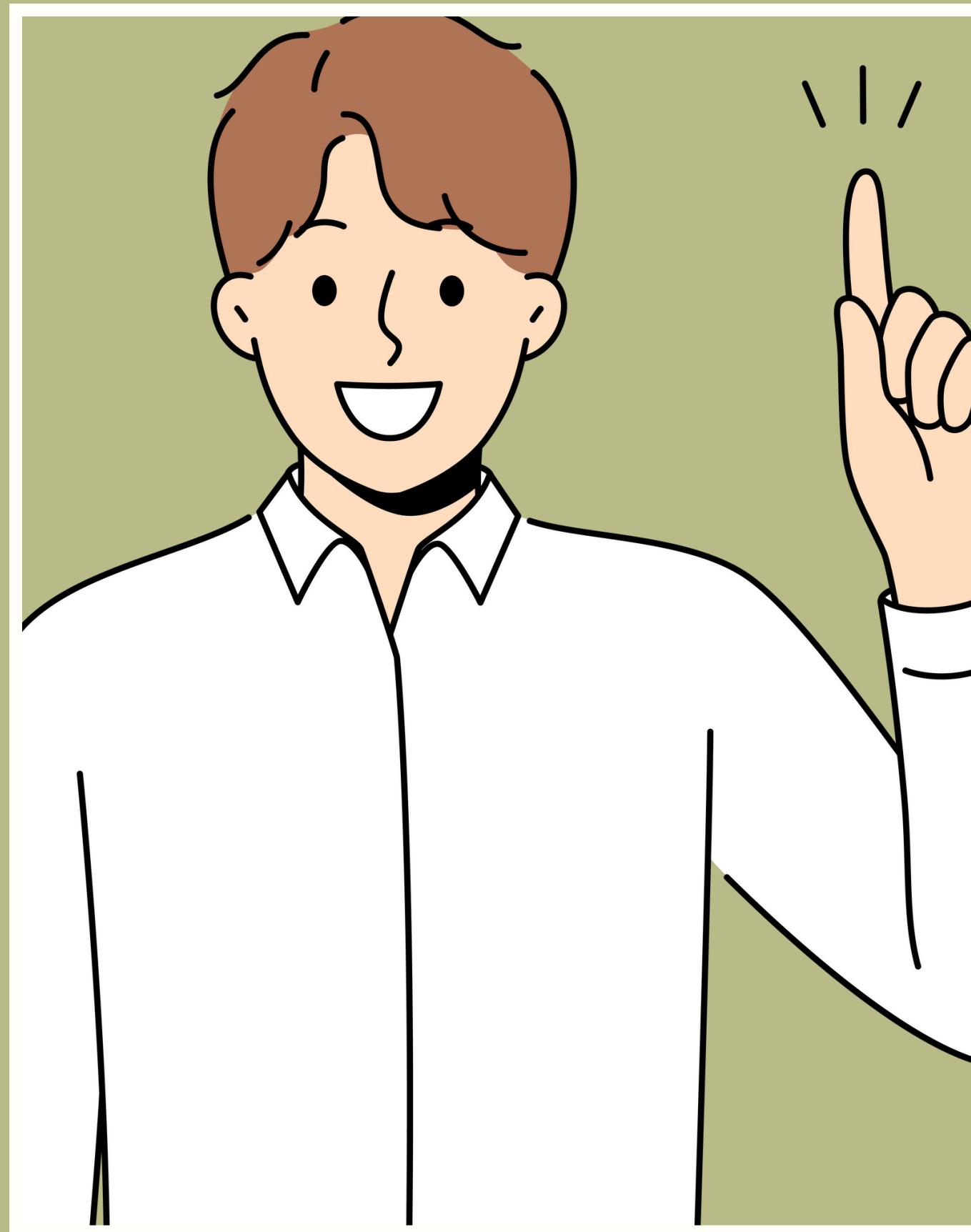
public class ActivityLogger {
    public void logActivity(String activity) {
        System.out.println("Logging activity: " + activity);
    }
}
```

DIVISION INTERFACES

→ *Motivo de los cambios*

La interfaz “Printer”, contiene muchos métodos que pueden no ser necesarios. Al separarlos en diferentes interfaces, podremos implementarlas sin la necesidad de depender de otros métodos que pueden no ser necesarios o útiles para una cierta tarea

```
PRINTER.java 1 ~> Users > a23andresfmm > Desktop > PRINTER.java > ...  
1 public interface Printer {  
2     void print(String content);  
3     void scan(String content);  
4     void fax(String content);  
5 }  
6  
7 public interface Printable {  
8     void print(String content);  
9 }  
10  
11 public interface Scannable {  
12     void scan(String content);  
13 }  
14  
15 public interface Faxable {  
16     void fax(String content);  
17 }  
18
```



CONSEJOS AL SEGREGAR

01 Comprender el código

Antes de realizar la **division**, debemos **comprender** que hace el método, así podemos **identificar que partes se pueden separar**.

02 Principio de Responsabilidad Única

Cada clase, método, interfaz **debe ser específico**. Un método para una sola cosa, una clase para una parte de la lógica y una interfaz específica con los métodos importantes.

REEMPLAZAR BUCLES

→ Motivo de los cambios

Podemos remplazar los bucles for tradicionales por streams, pudiendo así poner en una sola linea de código y evitaremos gestionar índices.

```
users > a23andresfmm > Desktop > Example.java > ...
public class Example {
    public void printNames(List<String> names) {
        for (int i = 0; i < names.size(); i++) {
            System.out.println(names.get(i));
        }
    }
}

public class Example {
    public void printNames(List<String> names) {
        names.forEach(System.out::println);
    }
}
```

REEMPLAZAR BUCKLES

→ Motivo de los cambios

Si en nuestro código tenemos alguna acumulación dentro de un bucle, podemos usar streams junto a al método “reduce” mejorando la intención de la acumulación y sin la necesidad de variables auxiliares

```
sample.java ▾
```

```
Users > a23andresfmm > Desktop > Example.java > ...
```

```
import java.util.List;
public class Example {
    public int sum(List<Integer> numbers) {
        int total = 0;
        for (int number : numbers) {
            total += number;
        }
        return total;
    }

    public class Example {
        public int sum(List<Integer> numbers) {
            return numbers.stream().reduce(0, Integer::sum);
        }
    }
}
```

REEMPLAZAR BUCLES

→ Motivo de los cambios

Al realizar búsquedas específicas, podremos sustituir el bucle tradicional por un “Stream” y el método “filter”, esto hace que sea mas conciso cuando queremos realizar alguna operacion de filtrado de datos

```
Users > a23andresfmm > Desktop > Example.java > ...
1 import java.util.List;
2 import java.util.stream.Collectors;
3
4 public class Example {
5     public List<String> filterLongNames(List<String> names) {
6         List<String> result = new ArrayList<>();
7         for (String name : names) {
8             if (name.length() > 5) {
9                 result.add(name);
10            }
11        }
12        return result;
13    }
14
15
16
17 public class Example {
18     public List<String> filterLongNames(List<String> names) {
19         return names.stream()
20                  .filter(name -> name.length() > 5)
21                  .collect(Collectors.toList());
22    }
23
24 }
```



CONSEJOS AL SUSTITUIR BUCKLES

- 01** *Usar Stream si se puede*
Pueden ser mas complejos de entender , pero son **mas efectivos y concisos** para estas tareas.
- 02** *Evitar complejidad en el Stream*
Si en el Stream vemos que su cuerpo es complejo, lo **separaremos en métodos**, para facilitar la legibilidad.
- 03** *Priorizar la claridad*
Los Streams son efectivos y poderosos, pero **si nuestro bucle for es mas conciso, lo usaremos**.

EXTRACCION

→ Motivo de los cambios

Podemos extraer en métodos diferentes la lógica del metodo, así es mas claro y cada método posee una responsabilidad, procesarOrden, validarOrden y procesarPago.

```
C:\Users\azuanresimli\Desktop\__ OrdenProcessor.java | OrderProcessor | processOrder(String)
1  public class OrderProcessor {
2      public void processOrder(String orderDetails) {
3          System.out.println("Validating order...");
4          if (orderDetails == null || orderDetails.isEmpty()) {
5              System.out.println("Invalid order.");
6              return;
7          }
8
9          System.out.println("Processing payment...");
10         // Payment logic here
11         System.out.println("Payment successful!");
12     }
13 }
14
15 public class OrderProcessor {
16     public void processOrder(String orderDetails) {
17         if (!validateOrder(orderDetails)) {
18             return;
19         }
20         processPayment();
21     }
22
23     private boolean validateOrder(String orderDetails) {
24         System.out.println("Validating order...");
25         if (orderDetails == null || orderDetails.isEmpty()) {
26             System.out.println("Invalid order.");
27             return false;
28         }
29         return true;
30     }
31
32     private void processPayment() {
33         System.out.println("Processing payment...");
34         // Payment logic here
35         System.out.println("Payment successful!");
36     }
37 }
38 }
```

EXTRACCION

→ Motivo de los cambios

También podemos extraer cálculos complejos, en vez de tener un método con toda la lógica de calculo, podemos extraer el código para cada calculo, así es mas funcional y facilita el realizar cambios y podremos reutilizarlos en otra parte de código si es necesario.

```
users > a23andresfmm > Desktop > SalaryCalculator.java > ...
public class SalaryCalculator {
    public double calculateNetSalary(double grossSalary) {
        double tax = grossSalary * 0.2;
        double insurance = grossSalary * 0.05;
        return grossSalary - tax - insurance;
    }
}

public class SalaryCalculator {
    public double calculateNetSalary(double grossSalary) {
        double tax = calculateTax(grossSalary);
        double insurance = calculateInsurance(grossSalary);
        return grossSalary - tax - insurance;
    }

    private double calculateTax(double grossSalary) {
        return grossSalary * 0.2;
    }

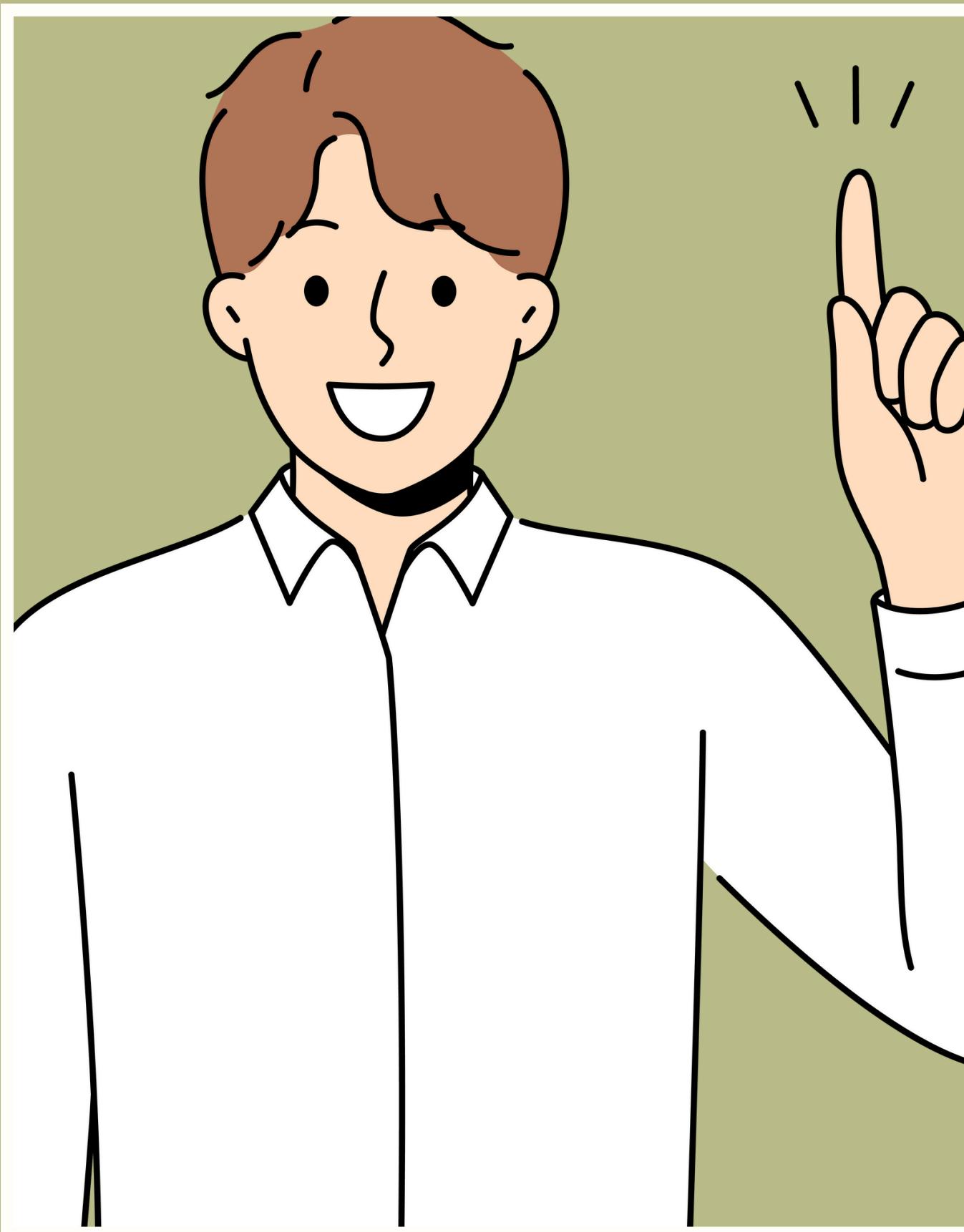
    private double calculateInsurance(double grossSalary) {
        return grossSalary * 0.05;
    }
}
```

EXTRACCION

→ Motivo de los cambios

Si tenemos condiciones anidadas, podemos extraerlas en un método independiente y llamarlo cuando lo precisemos, así mejoramos la legibilidad del calculo principal al ser mas simple el código.

```
public class DiscountService {  
    public double calculateDiscount(double price, String customerType) {  
        if (customerType.equals(anObject:"VIP")) {  
            return price * 0.2;  
        } else if (customerType.equals(anObject:"Regular")) {  
            return price * 0.1;  
        } else {  
            return 0;  
        }  
    }  
  
    public class DiscountService {  
        public double calculateDiscount(double price, String customerType) {  
            return price * getDiscountRate(customerType);  
        }  
  
        private double getDiscountRate(String customerType) {  
            switch (customerType) {  
                case "VIP": return 0.2;  
                case "Regular": return 0.1;  
                default: return 0;  
            }  
        }  
    }
```



CONSEJOS AL EXTRAER

01 Minimizar dependencia

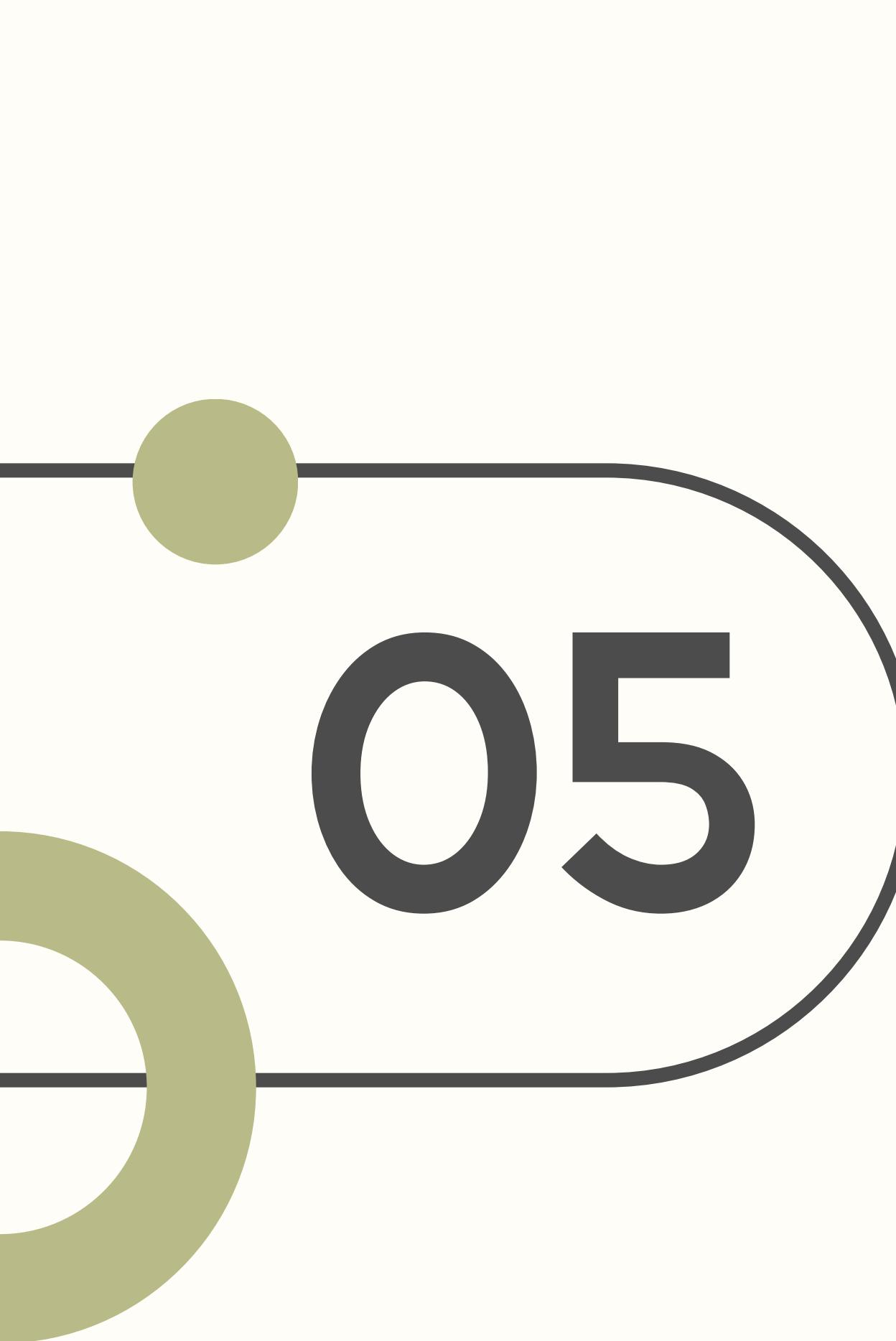
Los métodos que se extraen se debe intentar que tengan la **menor cantidad de parámetros** para que sea reutilizable

02 Asegurar el comportamiento

Verificar con **pruebas unitarias** que la funcionalidad no ha cambiado después de la extracción

03 Aplicar gradualmente

La extracción **se hace poco a poco**, si hay métodos extensos, se realizaran extracciones en **bloques pequeños** para asegurar la funcionalidad



05

Refactorización: Conclusiones

Conclusiones

La refactorización es **esencial** para que un **código sea limpio, conciso y optimo**, nos aporta muchas ventajas a la hora de que nuestro código sea **mantenible** y **ayuda a la utilización** de otros desarrolladores.

Sin embargo, **si nuestro código funciona bien**, y no presenta **“Code Smells”**, código de **mala calidad** o no posee **test unitarios** para comprobar su funcionamiento, realizar la refactorización puede ser **riesgoso y contraproducente**.

Debemos **analizar** cada caso y sopesar si, en efecto, es recomendable realizarla en nuestro proyecto.

Muchas
GRACIAS

