

# **MADS**

## **S8: Refactoring**

### **(Bloque 2 - XP)**

# El post de la semana



**Codesai**  
@codesaidev

...

Entramos en la última semana para inscribirse en esta edición abierta de nuestro curso Code Smells & Refactoring con Java!  
[codesai.com/cursos/refacto...](http://codesai.com/cursos/refacto...) (23, 24, 25, 30 y 31 octubre) Escribe a contact@codesai.com

Anímate!

Hay 50% de descuento para colectivos poco representados.

[Curso Codesai](#)

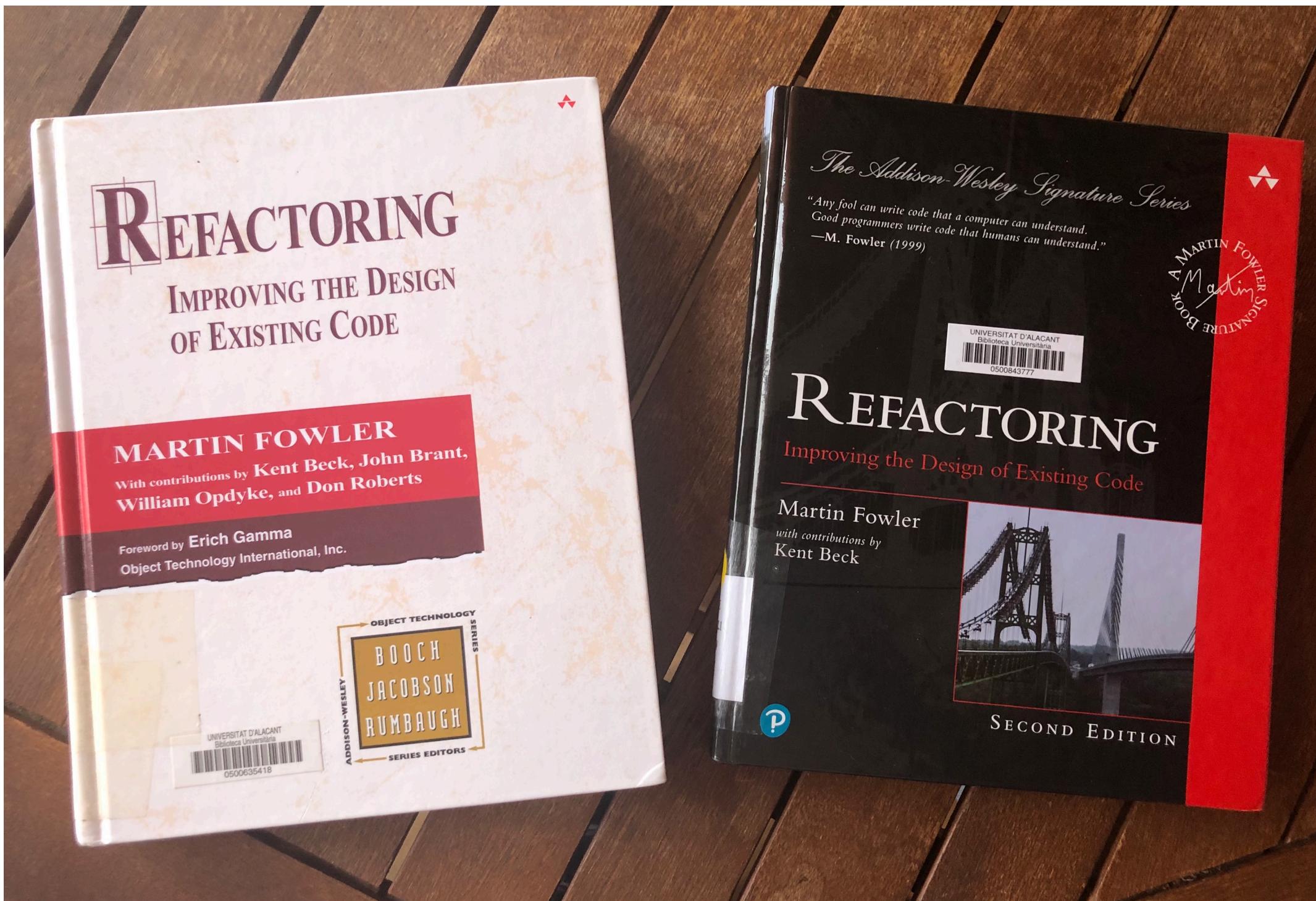
# Definición

Una refactorización (*refactoring* en inglés) de un programa consiste en la realización de una **transformación** de su código fuente **sin cambiar su funcionamiento**.

Se modifica la *estructura* del programa con el objetivo de mejorar su calidad:

- más legible
- más modificable
- mejorar la cohesión
- reducir las dependencias

# Libros de Martin Fowler



# Estándarización

- El trabajo fundamental de Fowler ha sido el de estandarizar las refactorizaciones, dándoles un nombre propio a cada una y clasificándolas.

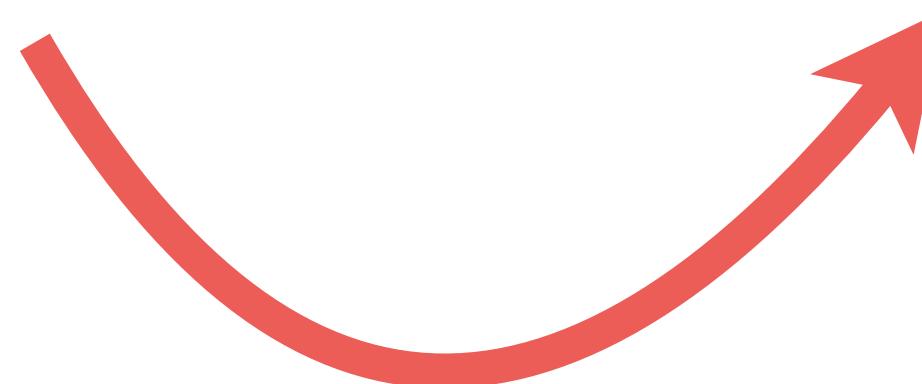
**Using the Catalog ►**

Tags	Change Function Declaration	Remove Dead Code
<input type="checkbox"/> basic <input type="checkbox"/> encapsulation <input type="checkbox"/> moving-features <input type="checkbox"/> organizing-data <input type="checkbox"/> simplify-conditional-logic <input type="checkbox"/> refactoring-apis <input type="checkbox"/> dealing-with-inheritance <input type="checkbox"/> collections <input type="checkbox"/> delegation <input type="checkbox"/> errors <input type="checkbox"/> extract <input type="checkbox"/> parameters <input type="checkbox"/> fragments <input type="checkbox"/> grouping-function <input type="checkbox"/> immutability <input type="checkbox"/> inline <input type="checkbox"/> remove <input type="checkbox"/> rename <input type="checkbox"/> split-phase <input type="checkbox"/> variables	Add Parameter • Change Signature • Remove Parameter • Rename Function • Rename Method	
Change Reference to Value	Remove Flag Argument	
Change Value to Reference	Remove Middle Man	
Collapse Hierarchy	Remove Setting Method	
Combine Functions into Class	Remove Subclass	
Combine Functions into Transform	Rename Field	
Consolidate Conditional Expression	Rename Variable	
Decompose Conditional	Replace Command with Function	

# Ejemplo - Change Function Declaration

```
public class Movie {  
    public double compute(int number) {  
        ...  
    }  
  
    ...  
    int daysRented = 4;  
    double price = movie.compute(daysRented);  
    ...
```

```
public class Movie {  
    public double getCharge(int numberOfDays) {  
        ...  
    }  
  
    ...  
    int daysRented = 4;  
    double price = movie.getCharge(daysRented);  
    ...
```



# ¿Cómo nos aseguramos de que no se cambia el funcionamiento?

- La forma más habitual es mediante una batería de tests que prueba el código que se refactoriza.
- Basta con comprobar, una vez hecha la refactorización, que los tests siguen pasando.

La acción de mejorar el diseño [del código] sin cambiar su conducta se denomina refactorización. La idea tras la refactorización es que podemos hacer el software más mantenible sin cambiar su conducta si escribimos tests para asegurarnos de que la conducta existente no cambia y realizamos pequeños cambios para verificarlo todo a lo largo del proceso.

Michael Feathers (2004), Working Effectively with Legacy Code

# Pequeños pasos (1)

- Es habitual ejecutar descomponer una refactorización grande en pequeños pasos, realizando varias refactorizaciones pequeñas.
- Así la probabilidad de introducir errores es menor.
- Ejemplo:

```
double price() {  
    // price is base price - quantity discount + shipping  
    return quantity * itemPrice -  
        Math.max(0, quantity - 500) * itemPrice * 0.05 +  
        Math.min(quantity * itemPrice * 0.1, 100.0);  
}
```

## Pequeños pasos (2)

- Aplicamos el refactoring *Extract Method*

```
double price() {  
    // price is base price - quantity discount + shipping  
    return basePrice() -  
        Math.max(0, quantity - 500) * itemPrice * 0.05 +  
        Math.min(basePrice() * 0.1, 100.0);  
  
}  
  
private double basePrice() {  
    return quantity * itemPrice;  
}
```

## Pequeños pasos (3)

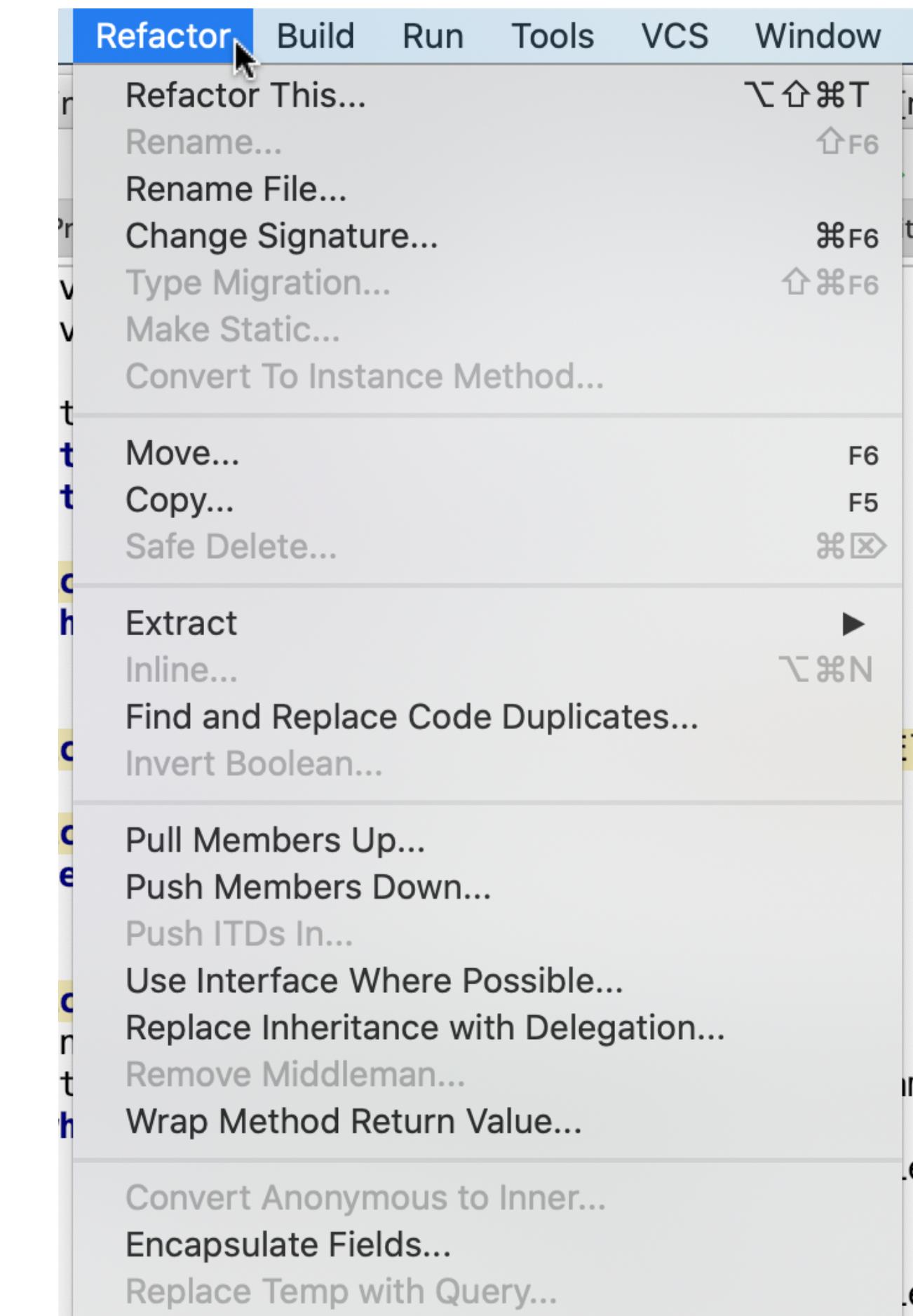
```
double price() {  
    // price is base price - quantity discount + shipping  
    return basePrice() -  
        Math.max(0, quantity - 500) * itemPrice * 0.05 +  
        shipping();  
  
}  
  
private double shipping() {  
    return Math.min(basePrice() * 0.1, 100.0);  
}  
  
private double basePrice() {  
    return quantity * itemPrice;  
}
```

## Pequeños pasos (4)

```
double price() {  
    return basePrice() - discount() + shipping();  
}  
  
private double discount() {  
    Math.max(0, quantity - 500) * itemPrice * 0.05  
}  
  
private double shipping() {  
    return Math.min(basePrice() * 0.1, 100.0);  
}  
  
private double basePrice() {  
    return quantity * itemPrice;  
}
```

# Refactorización en los IDEs

- La popularidad de las técnicas de refactorización y su estandarización ha hecho posible su incorporación a la gran mayoría de IDEs.
- Entre los más avanzados se encuentra IntelliJ



# 61 refactorizaciones

- En la primera edición del libro de Fowler se presentan 72 refactorizaciones.
- En la segunda se reducen a 61.

## List of Refactorings

Change Function Declaration (124)  
Change Reference to Value (252)  
Change Value to Reference (256)  
Collapse Hierarchy (380)  
Combine Functions into Class (144)  
Combine Functions into Transform (149)  
Consolidate Conditional Expression (263)  
Decompose Conditional (260)  
Encapsulate Collection (170)  
Encapsulate Record (162)  
Encapsulate Variable (132)  
Extract Class (182)  
Extract Function (106)  
Extract Superclass (375)  
Extract Variable (119)  
Hide Delegate (189)  
Inline Class (186)  
Inline Function (115)  
Inline Variable (123)  
Introduce Assertion (302)  
Introduce Parameter Object (140)  
Introduce Special Case (289)  
Move Field (207)  
Move Function (198)  
Move Statements into Function (213)  
Move Statements to Callers (217)  
Parameterize Function (310)  
Preserve Whole Object (319)  
Pull Up Constructor Body (355)  
Pull Up Field (353)  
Pull Up Method (350)  
Push Down Field (361)  
Push Down Method (359)  
Remove Dead Code (237)  
Remove Flag Argument (314)  
Remove Middle Man (192)  
Remove Setting Method (331)  
Remove Subclass (369)  
Rename Field (244)  
Rename Variable (137)  
Replace Command with Function (344)  
Replace Conditional with Polymorphism (272)  
Replace Constructor with Factory Function (334)  
Replace Derived Variable with Query (248)  
Replace Function with Command (337)  
Replace Inline Code with Function Call (222)  
Replace Loop with Pipeline (231)  
Replace Nested Conditional with Guard Clauses (266)  
Replace Parameter with Query (324)  
Replace Primitive with Object (174)  
Replace Query with Parameter (327)  
Replace Subclass with Delegate (381)  
Replace Superclass with Delegate (399)  
Replace Temp with Query (178)  
Replace Type Code with Subclasses (362)  
Separate Query from Modifier (306)  
Slide Statements (223)  
Split Loop (227)  
Split Phase (154)  
Split Variable (240)  
Substitute Algorithm (195)

# Code smells

- Son indicadores de mal diseño; cosas que deben ser refactorizadas.
- Los tres más comunes (veremos más cuando hablemos de SOLID)
  - Nombre misterioso
  - Código duplicado
  - Función larga

He desarrollado el hábito de escribir funciones muy cortas, típicamente de sólo unas pocas líneas de código. Para mí, cualquier función con más de media docena de líneas empieza a oler, y tengo bastantes funciones con una única línea de código.

Martin Fowler, Refactoring

# Algunos ejemplos de refactoring

- Extract Method
- Move Method
- Replace Temp with Query
- Parametrize Function

# Extract Method - Código inicial

```
void printOwing(double previousAmount) {  
  
    Enumeration e = orders.elements();  
    double outstanding = previousAmount * 1.2;  
  
    printBanner();  
  
    // calculate outstanding  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        outstanding += each.getAmount();  
    }  
  
    printDetails(outstanding);  
}
```

# Extract Method - Código refactorizado

```
void printOwing(double previousAmount) {  
    printBanner();  
    double outstanding = getOutstanding(previousAmount * 1.2);  
    printDetails(outstanding);  
}  
  
double getOutstanding(double initialValue) {  
    double result = initialValue;  
    Enumeration e = orders.elements();  
    while (e.hasMoreElements()) {  
        Order each = (Order) e.nextElement();  
        result += each.getAmount();  
    }  
    return result;  
}
```

# Move Method - Código inicial

```
public class Account {  
    private AccountType type;  
    private int daysOverdrawn;  
  
    ...  
    double overdraftCharge() {  
        if (type.isPremium()) {  
            double result = 10;  
            if (daysOverdrawn > 7) result += (daysOverdrawn - 7) * 0.85;  
            return result;  
        }  
        else return daysOverdrawn * 1.75;  
    }  
  
    double bankCharge() {  
        double result = 4.5;  
        if (daysOverdrawn > 0) result += overdraftCharge();  
        return result;  
    }  
}
```

# Move Method - Código refactorizado

```
public class Account {
    ...
    double overdraftCharge() {
        return type.overdraftCharge(daysOverdrawn);
    }

    double bankCharge() {
        double result = 4.5;
        if (daysOverdrawn > 0) result += overdraftCharge();
        return result;
    }
}

public class AccountType {
    ...
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7) result += (daysOverdrawn - 7) * 0.85;
            return result;
        }
        else return daysOverdrawn * 1.75;
    }
}
```

# Replace Temp with Query - Código inicial

```
double getPrice() {  
    int basePrice = quantity * itemPrice;  
    double discountFactor;  
    if (basePrice > 1000) discountFactor = 0.95;  
    else discountFactor = 0.98;  
    return basePrice * discountFactor;  
}
```

Suponemos que `getPrice()` es un método y que las variables `quantity` y `itemPrice` son variables de instancia.

# Replace Temp with Query - Paso 1

```
double getPrice() {  
    double discountFactor;  
    if (basePrice() > 1000) discountFactor = 0.95;  
    else discountFactor = 0.98;  
    return basePrice() * discountFactor;  
}  
  
private int basePrice() {  
    return quantity * itemPrice;  
}
```

## Paso 2 - Aplicamos Extract Method

```
double getPrice() {  
    double discountFactor = discountFactor()  
    return basePrice() * discountFactor;  
}  
  
private int basePrice() {  
    return quantity * itemPrice;  
}  
  
private double discountFactor() {  
    if (basePrice() > 1000) return 0.95;  
    else return 0.98;  
}
```

## Paso 3 - Aplicamos Inline Variable

```
double getPrice() {  
    return basePrice() * discountFactor();  
}  
  
private int basePrice() {  
    return quantity * itemPrice;  
}  
  
private double discountFactor() {  
    if (basePrice() > 1000) return 0.95;  
    else return 0.98;  
}
```

# Parametrize Function - Código inicial

```
public class Book {  
    ...  
    public void addReservation(Customer customer) {  
        this.reservations.push(customer);  
    }  
  
    public void addReservationWithPriority(Customer customer) {  
        this.priorityReservations.push(customer);  
    }  
}  
  
...  
book.addReservation(aCustomer);  
...  
// cliente prioritario  
book.addReservationWithPriority(prioritaryCustomer);  
...
```

# Parametrize Function - Código final

```
public class Book {  
    ...  
    public void addReservation(Customer customer, boolean priority) {  
        if (priority) {  
            this.priorityReservations.push(customer);  
        } else {  
            this.reservations.push(customer);  
        }  
    }  
    ...  
    book.addReservation(aCustomer, false);  
    ...  
    // cliente prioritario  
    book.addReservation(prioritaryCustomer, true);  
    ...
```

# Ejemplo completo: alquiler video club

El programa tiene tres clases principales:

- `Movie` : clase que define una película, con un título de película y un código de precio.
- `Rental` : clase que define un alquiler, con la película y el número de días alquilada.
- `Customer` : clase que representa el cliente, con la lista de alquileres que ha realizado y el cálculo de la cuenta.

En concreto, las reglas de negocio en las que se basa el cálculo del total de dinero a pagar por el alquiler son las siguientes:

1. Las películas normales cuestan 2€ y tienen un plazo de devolución máximo de 2 días.
2. Las películas para niños cuestan 1,5€ y tienen un plazo de devolución máximo de 3 días.
3. Por cada día extra fuera de plazo se aplica una penalización de 1,5€.
4. Las novedades cuestan 3€ por cada día.

En cuanto a los puntos de bonificación que se obtienen:

1. Se obtiene un punto de bonificación por cada película alquilada.
2. Si se alquila una novedad más de 1 día se obtiene un punto extra de bonificación.

# Ejemplo de ejecución

```
public class Main {  
    public static void main(String[] args) {  
        Movie tenet = new Movie("Tenet", Movie.NEW_RELEASE);  
        Movie busan = new Movie("Train to Busan", Movie.REGULAR);  
        Movie padre = new Movie("Padre no hay más que uno", Movie.CHILDRENS);  
  
        Rental rental1 = new Rental(tenet, 2);  
        Rental rental2 = new Rental(busan, 2);  
        Rental rental3 = new Rental(padre, 1);  
  
        Customer customer = new Customer("domingogallardo");  
  
        customer.addRental(rental1);  
        customer.addRental(rental2);  
        customer.addRental(rental3);  
  
        System.out.println(customer.statement());  
    }  
}
```

```
Rental Record for domingogallardo  
    Tenet    6.0  
    Train to Busan  2.0  
    Padre no hay más que uno      1.5  
Amount owed is 9.5  
You earned 4 frequent renter points
```

# Tests (1)

```
public class VideoClubTests {  
    @Test  
    public void testCrearMovie() {  
        Movie tenet = new Movie("Tenet", Movie.NEW_RELEASE);  
        assertEquals(tenet.getTitle(), "Tenet");  
    }  
  
    @Test  
    public void testPriceCode() {  
        Movie tenet = new Movie("Tenet", Movie.NEW_RELEASE);  
        Movie busan = new Movie("Train to Busan", Movie.REGULAR);  
        Movie padre = new Movie("Padre no hay más que uno", Movie.CHILDRENS);  
  
        assertEquals(tenet.getPriceCode(), Movie.NEW_RELEASE);  
        assertEquals(busan.getPriceCode(), Movie.REGULAR);  
        assertEquals(padre.getPriceCode(), Movie.CHILDRENS);  
    }  
}
```

## Tests (2)

```
@Test
public void testRental() {
    Movie tenet = new Movie("Tenet", Movie.NEW_RELEASE);
    Rental rental = new Rental(tenet, 2);
    assertThat(rental.getMovie().getTitle(), equalTo("Tenet"));
    assertThat(rental.getDaysRented(), equalTo(2));
}

@Test
public void testCustomer() {
    Customer customer = new Customer("domingogallardo");
    assertThat(customer.getName(), equalTo("domingogallardo"));
}
```

# Tests (3)

```
@Test
public void testCustomerEmptyStatement() {
    Customer customer = new Customer("domingogallardo");
    String statement = customer.statement();
    assertThat(statement, allOf(
        containsString("Rental Record for domingogallardo"),
        containsString("Amount owed is 0.0"),
        containsString("You earned 0 frequent renter points")));
}
```

# Tests (y 4)

```
@Test
public void testCustomerFullStatement() {
    Customer customer = new Customer("domingogallardo");
    Movie tenet = new Movie("Tenet", Movie.NEW_RELEASE);
    Movie busan = new Movie("Train to Busan", Movie.REGULAR);
    Movie padre = new Movie("Padre no hay más que uno", Movie.CHILDRENS);

    Rental rental1 = new Rental(tenet, 2);
    Rental rental2 = new Rental(busan, 2);
    Rental rental3 = new Rental(padre, 1);

    customer.addRental(rental1);
    customer.addRental(rental2);
    customer.addRental(rental3);

    assertThat(customer.statement(), allOf(
        containsString("Rental Record for domingogallardo"),
        stringContainsInOrder("Tenet", "6.0"),
        stringContainsInOrder("Train to Busan", "2.0"),
        stringContainsInOrder("Padre no hay más que uno", "1.5"),
        containsString("Amount owed is 9.5"),
        containsString("You earned 4 frequent renter points")));
}
```

# Refactorización

- Queremos añadir la funcionalidad que nos permita imprimir la factura en HTML.
- El código no está preparado para que sea fácil hacerlo. Hay que refactorizar.
- [Listado de commits](#) con la refactorización.

# Refactorización para imprimir la cuenta en HTML

- Paso 1: Extract Method
- Paso 2: Rename Variable
- Paso 3: Move Method
- Paso 4: Inline Method
- Paso 5: Replace Temp with Query
- Paso 6: Extract + Move Method
- Paso 7: Replace Temp with Query
- Paso 8: Añadir método que genera el HTML

# Refactorización para introducir herencia y polimorfismo

- Paso 1: Move Method
- Paso 2: Move Method
- Paso 3: Encapsulate Variable
- Paso 4: Replace Type Code with State
- Paso 5: Move Method
- Paso 6: Replace Conditional with Polymorphism
- Paso 7: Replace Conditional with Polymorphism

main ▾

[apuntes-mads / apuntes / 08-refactoring / refactoring.md](#)[Go to file](#)

...



domingogallardo Añadidos tests

Latest commit 70eba49 12 minutes ago [History](#)

1 contributor

2069 lines (1636 sloc) | 61.4 KB

[Code](#) [Raw](#) [Blame](#) [Edit](#) [Delete](#)

# Refactoring

Una refactorización (*refactoring* en inglés) de un programa consiste en la realización de una transformación de su código fuente sin cambiar su funcionamiento. A diferencia de un cambio del código en el que se añaden o modifican sus funcionalidades, en una refactorización únicamente se modifica la *estructura* del programa.

Por ejemplo, cuando cambiamos el nombre de un método y cambiamos el nombre de todas las invocaciones a ese método estamos haciendo un ejemplo concreto de la refactorización [Change Function Declaration](#).

Es el caso del ejemplo siguiente. El método `compute` de la siguiente clase `Movie` es muy poco descriptivo.