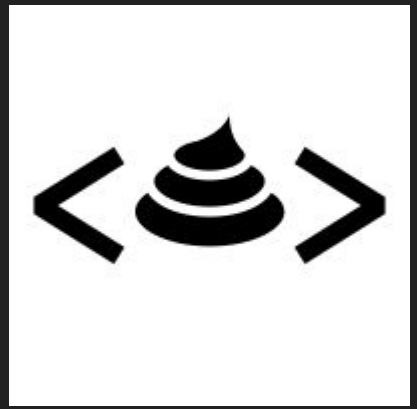
Code smells & Refactoring



Whoami

I'm developer/security researcher who likes to make/break things 🛠️ and teach people.

Github
@davcortez | @hexod0t

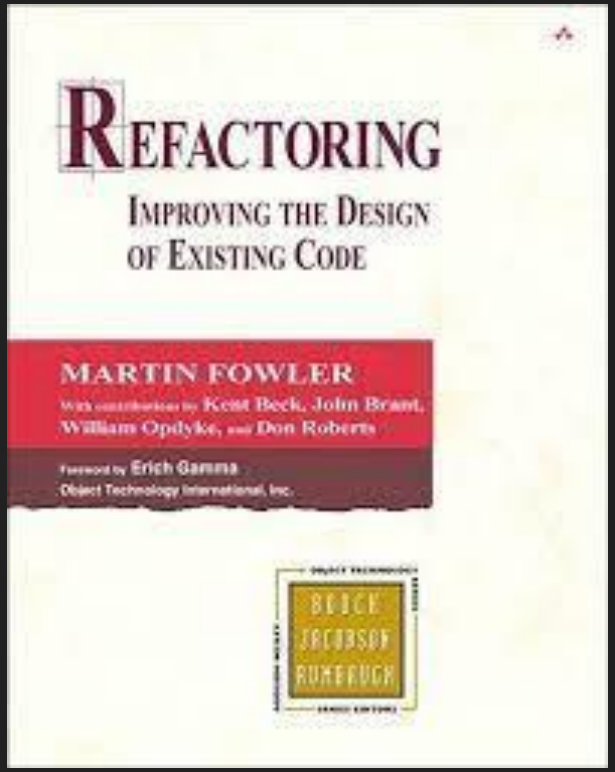
Linkedin
@david-cortez-alban

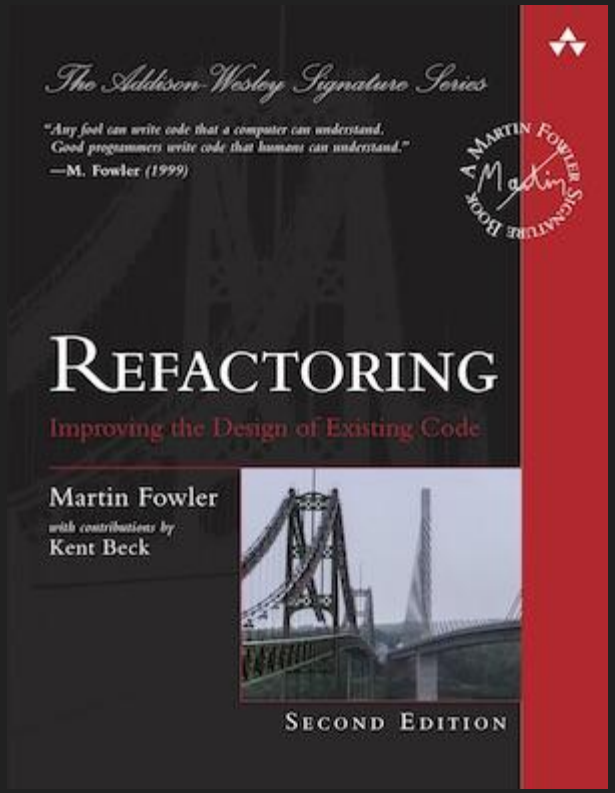
¿Qué son los Code Smells?

- Término agnóstico del lenguaje de programación.
- No son un bug de programación.
- Indican deficiencias en el diseño.

Kent Beck

Martin Fowler


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
BECK
JACOBSON
RUMBAUGH


The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
—M. Fowler (1999)
A MARTIN FOWLER SIGNATURE BOOK
Martin
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

¿Por qué deberíamos preocuparnos?

- Poca Legibilidad (pobre).
- Baja productividad.
- Riesgo de introducir otros errores (bug pollution).
- Demoras en liberar código.
- Mantenibilidad (pobre).

Classic Smells

Alternative Classes w/ Different Interfaces

Comments

Data Class

Data Clumps

Divergent Change

Duplicated Code

Feature Envy

Inappropriate Intimacy

Incomplete Library Client

Large Class

Lazy Class

Long Method

Long Parameter List

Message Chains

Middle Man

Parallel Inheritance Hierarchies

Primitive Obsession

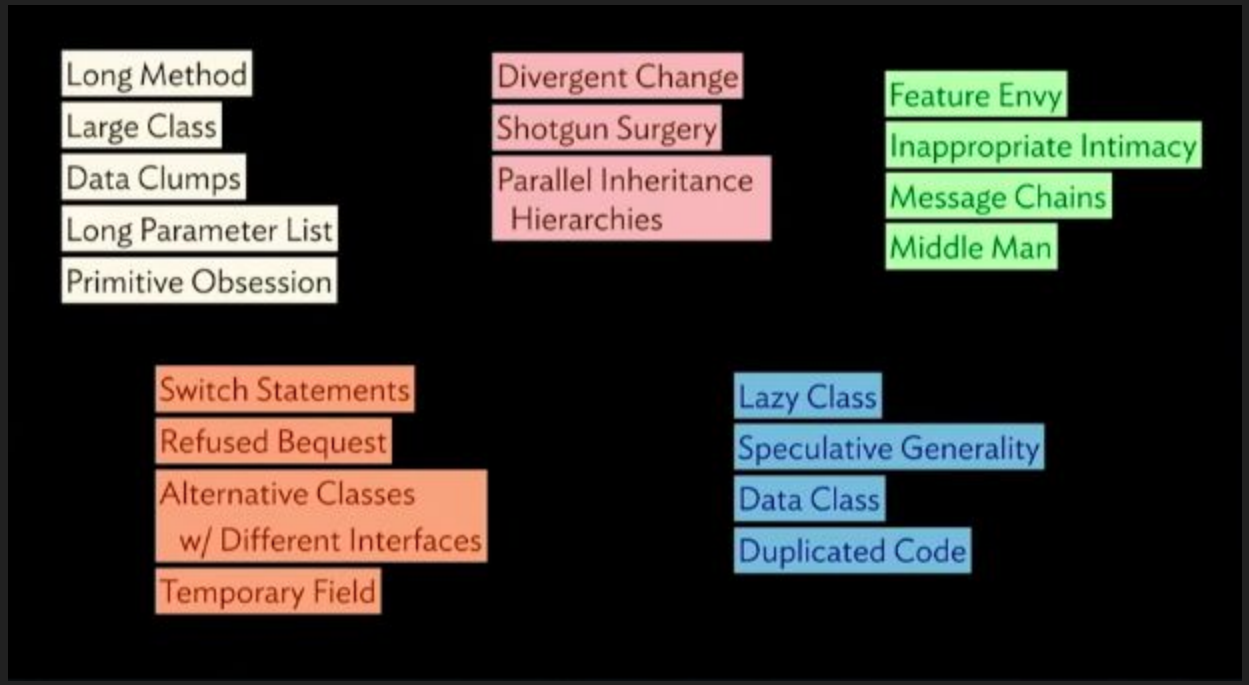
Refused Bequest

Shotgun Surgery

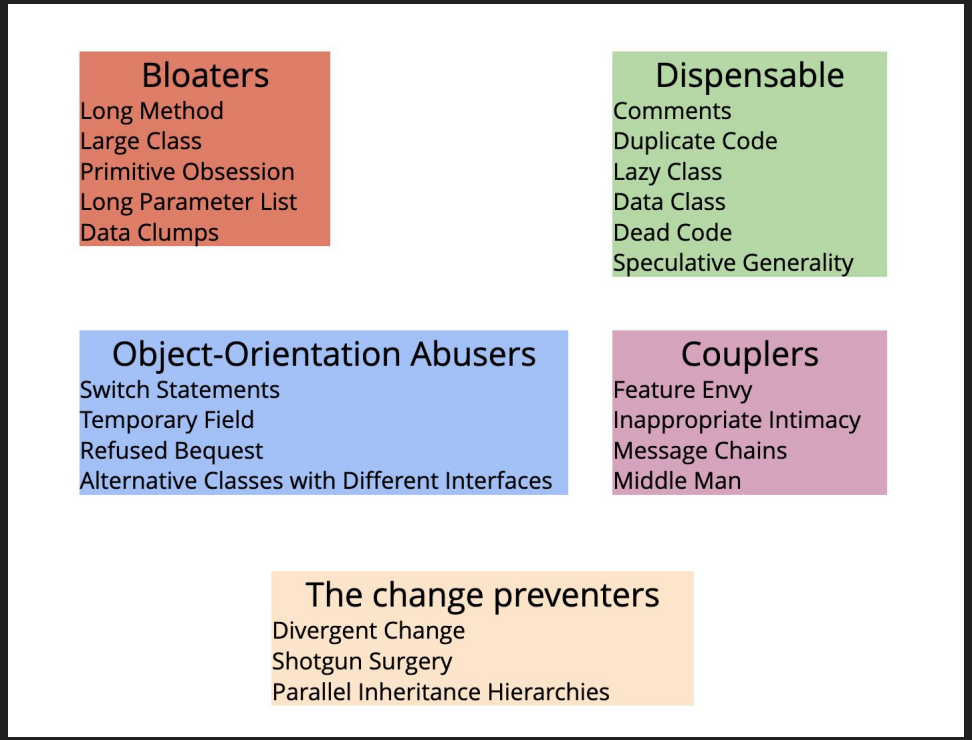
Speculative Generality

Switch Statements

Temporary Field


Long Method
Large Class
Data Clumps
Long Parameter List
Primitive Obsession
Divergent Change
Shotgun Surgery
Parallel Inheritance Hierarchies
Feature Envy
Inappropriate Intimacy
Message Chains
Middle Man
Switch Statements
Refused Bequest
Alternative Classes w/ Different Interfaces
Temporary Field
Lazy Class
Speculative Generality
Data Class
Duplicated Code

A Taxonomy for "Bad Code Smells" - Mika Mantyla



Bloaters

Es código, métodos y clases que su tamaño ha aumentado en grandes proporciones y son difíciles de entender y trabajar.

Long Method

```python
def process_data(data):
    # Check if data is valid
    if not data:
        return None

    # Convert data to list
    data_list = data.split(",")

    # Remove duplicates
    unique_list = []
    for item in data_list:
        if item not in unique_list:
            unique_list.append(item)

    # Convert list items to integers
    int_list = []
    for item in unique_list:
        int_list.append(int(item))

    # Calculate sum and average
    total = 0
    for num in int_list:
        total += num
    average = total / len(int_list)

    # Return result
    return (total, average)
```

Long Class

```python
class User:
    def __init__(self, name, age, email, password):
        self.name = name
        self.age = age
        self.email = email
        self.password = password
        self.is_admin = False

    def set_admin(self, is_admin):
        self.is_admin = is_admin

    def reset_password(self, new_password):
        # code to reset password

    def send_email(self, subject, body):
        # code to send email

    def validate_user(self):
        # code to validate user information

    def get_user_data(self):
        # code to retrieve user data

    def update_user_data(self, data):
        # code to update user data

    # many more methods...
```

Data Clumps

```python
def calculate_order_total(order):
    # get data from order dictionary
    customer_name = order["customer_name"]
    customer_email = order["customer_email"]
    customer_address = order["customer_address"]
    items = order["items"]

    # perform calculations
    subtotal = sum(item["price"] * item["quantity"] for item in items)
    tax = subtotal * 0.1
    shipping = 5.0

    # output results
    print(f"Customer Name: {customer_name}")
    print(f"Customer Email: {customer_email}")
    print(f"Customer Address: {customer_address}")
    print(f"Subtotal: {subtotal}")
    print(f"Tax: {tax}")
    print(f"Shipping: {shipping}")
    print(f"Total: {subtotal + tax + shipping}")
```

Long Parameter List

```
def send_email(subject, body, to, cc, bcc, attachments):
    # code to send email
```

Primitive Obsession

```python
def validate_user(username, password):
    if not isinstance(username, str):
        return False
    if not isinstance(password, str):
        return False
    if len(username) < 5 or len(username) > 20:
        return False
    if len(password) < 8 or len(password) > 30:
        return False
    # code to validate user credentials
```

Object-Orientation Abusers

Este tipo de code smell se da cuando se aplica de forma incorrecta o incompleta los principios de la programación orientada a objetos.

Switch Statements

```python
def calculate(operation, x, y):
    if operation == "add":
        return x + y
    elif operation == "subtract":
        return x - y
    elif operation == "multiply":
        return x * y
    elif operation == "divide":
        if y == 0:
            raise ValueError("Cannot divide by zero")
        return x / y
    else:
        raise ValueError(f"Invalid operation: {operation}")
```

Temporary Field

```python
class Order:
    def __init__(self, items):
        self.items = items
        self.total_price = None

    def calculate_total_price(self):
        total = 0
        for item in self.items:
            total += item.price
        self.total_price = total

    def print_order(self):
        for item in self.items:
            print(f"{item.name}: {item.price}")
        print(f"Total price: {self.total_price}")
```

```javascript
function sum (a, b) {
  var total = a + b;
  return total;

}
```

Refused Bequest

```
class Vehicle:
    def start_engine(self):
        raise NotImplementedError

    def stop_engine(self):
        raise NotImplementedError

    def drive(self):
        raise NotImplementedError

class Car(Vehicle):
    def start_engine(self):
        # some implementation

    def stop_engine(self):
        # some implementation

class Bicycle(Vehicle):
    def drive(self):
        # some implementation

    # bicycle doesn't have an engine,
    #so start_engine and stop_engine methods are not implemented
```

Alternative Classes with Different Interfaces

```python
class Shape:
    def area(self):
        raise NotImplementedError

    def perimeter(self):
        raise NotImplementedError

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height

    def get_perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius ** 2

    def calculate_circumference(self):
        return 2 * 3.14 * self.radius
```

Change Preventers

Este tipo de code smell se da cuando se necesita cambiar algo en una parte del código, y se deben hacer muchos cambios en otras partes. Ocasionando que el sistema se vuelva complejo y costoso de mantener.

Shotgun Surgery

```python
class User:
    def save(self):
        try:
            # save user to the database
            pass
        except Exception as e:
            # log the error to a file
            with open("error.log", "a") as f:
                f.write(str(e))

class Order:
    def process(self):
        try:
            # process the order
            pass
        except Exception as e:
            # log the error to a file
            with open("error.log", "a") as f:
                f.write(str(e))
```

Divergent Changes

```python
class PaymentProcessor:
    def __init__(self, payment_method):
        self.payment_method = payment_method

    def process_payment(self, amount):
        if self.payment_method == "credit_card":
            # code to process payment using credit card
            return True
        elif self.payment_method == "paypal":
            # code to process payment using PayPal
            return True
        elif self.payment_method == "stripe":
            # code to process payment using Stripe
            return True
        else:
            # code to handle invalid payment method
            return False

    def cancel_payment(self, transaction_id):
        if self.payment_method == "credit_card":
            # code to cancel payment using credit card
            return True
        elif self.payment_method == "paypal":
            # code to cancel payment using PayPal
            return True
        elif self.payment_method == "stripe":
            # code to cancel payment using Stripe
            return True
        else:
            # code to handle invalid payment method
            return False
```

Parallel Inheritance Hierarchies

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def eat(self):
        print(f"{self.name} is eating")

    def sleep(self):
        print(f"{self.name} is sleeping")
```

```python
class Mammal(Animal):
    def __init__(self, name):
        super().__init__(name)

    def give_birth(self):
        print(f"{self.name} is giving birth")


class Bird(Animal):
    def __init__(self, name):
        super().__init__(name)

    def lay_eggs(self):
        print(f"{self.name} is laying eggs")
```

```python
class Cat(Mammal):
    def __init__(self, name):
        super().__init__(name)

    def meow(self):
        print(f"{self.name} is meowing")


class Dog(Mammal):
    def __init__(self, name):
        super().__init__(name)

    def bark(self):
        print(f"{self.name} is barking")


class Chicken(Bird):
    def __init__(self, name):
        super().__init__(name)

    def cluck(self):
        print(f"{self.name} is clucking")


class Duck(Bird):
    def __init__(self, name):
        super().__init__(name)

    def quack(self):
        print(f"{self.name} is quacking")
```

Dispensables

Este tipo de code smell se da cuando existe o se agrega código sin importancia y que no se necesita volviendo el código ineficiente y difícil de entender.

Comments

```python
def calculate_total(items):
    """
    This function calculates the total amount of a list of items.

    :param items: A list of items.
    :return: The total amount.
    """
    total = 0
    # Loop over all the items and add their price to the total.
    for item in items:
        price = item['price']
        total += price

    # If the total amount is over $100, apply a 10% discount.
    if total > 100:
        total = total * 0.9

    return total
```

Duplicate Code

```python
def calculate_circle_area(radius):
    area = 3.14 * radius ** 2
    return area

def calculate_sphere_area(radius):
    area = 4 * 3.14 * radius ** 2
    return area
```

Dead Code

```python
def calculate_total(items):
    total = 0
    for item in items:
        price = item['price']
        total += price

    # This if statement is no longer needed,
    # but it remains in the code.
    if total < 0:
        print("Error: Total should be positive!")

    return total
```

Lazy Class

```
class Person:
    pass
```

```
class Calculator {
  constructor() {}
}

const calculator = new Calculator();
```

Speculative Generality

```python
class Shape:
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Rectangle(Shape):
    def draw(self):
        print("Drawing a rectangle")

class Triangle(Shape):
    def draw(self):
        print("Drawing a triangle")
```

Couplers

Este tipo de code smell contribuye a un acoplamiento excesivo entre clases o muestran lo que sucede cuando el acoplamiento es reemplazado por una delegación excesiva.

Feature Envy

```python
class Order:
    def __init__(self, customer, total):
        self.customer = customer
        self.total = total

    def print_invoice(self):
        return f"Name: {self.customer.name}, Total: {self.total}"

class Customer:
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone

    def get_order_total(self, order):
        return order.total

order = Order(Customer("John", "john@gmail.com", "123-456-7890"), 100)
customer = Customer("John", "john@gmail.com", "123-456-7890")
total = customer.get_order_total(order)
```

Inappropriate Intimacy

```python
class Customer:
    def __init__(self, name, address):
        self.name = name
        self.address = address
        self.billing_address = None

    def set_billing_address(self, billing_address):
        self.billing_address = billing_address

    def send_invoice(self):
        invoice = Invoice(self.billing_address)
        # send invoice to the customer

class Invoice:
    def __init__(self, billing_address):
        self.billing_address = billing_address

    def generate(self):
        # generate invoice
```

Message Chains

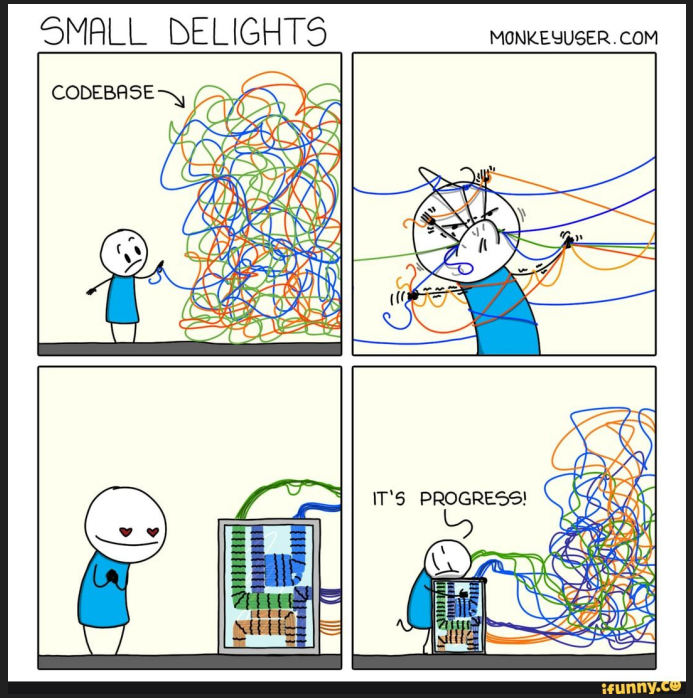
```python
class Address:
    def __init__(self, street, city, state, zip_code):
        self.street = street
        self.city = city
        self.state = state
        self.zip_code = zip_code

class Customer:
    def __init__(self, name, email, phone, address):
        self.name = name
        self.email = email
        self.phone = phone
        self.address = address

class Order:
    def __init__(self, customer, total):
        self.customer = customer
        self.total = total

    def get_customer_address(self):
        return f"{self.customer.address.street}, {self.customer.address.city}, {self.customer.address.state}, {self.customer.address.zip_code}"

order = Order(Customer("John", "john@gmail.com", "123-456-7890", Address("123 Main St", "Anytown", "CA", "12345")), 100)
address = order.get_customer_address()
```

Middle Man

```python
class Manager:
    def __init__(self, employee):
        self.employee = employee

    def get_employee_name(self):
        return self.employee.name

class Employee:
    def __init__(self, name):
        self.name = name

employee = Employee("John Doe")
manager = Manager(employee)

# Instead of calling employee.name directly,
# the client has to go through the manager object
name = manager.get_employee_name()
```

¿Qué es Refactoring?

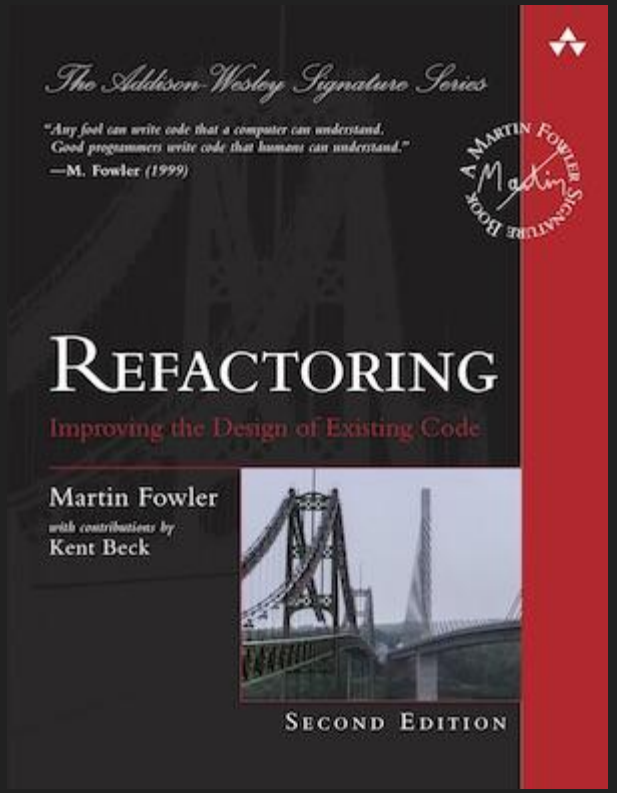
Técnica de reestructuración de código que no modifica su comportamiento externo



¿Qué es Refactoring?

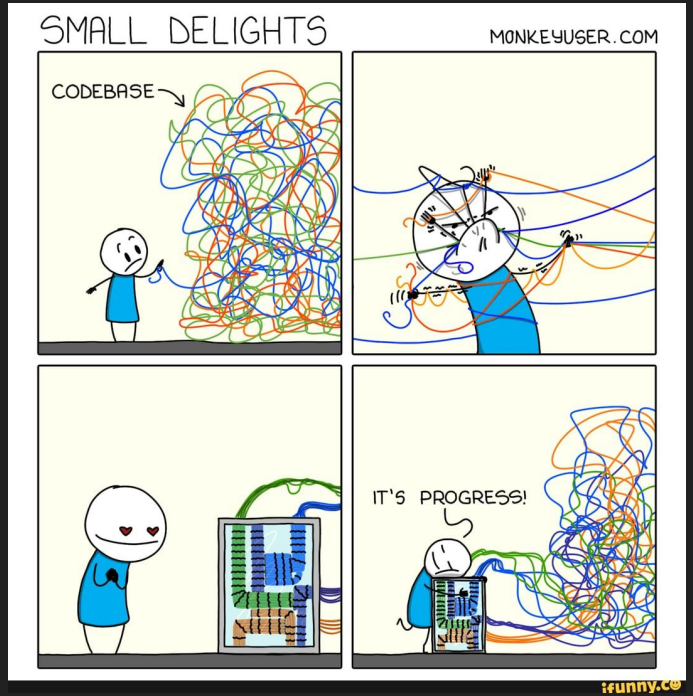
/noun/ A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

/verb/ to restructure software by applying a series of refactorings without changing its observable behaviour



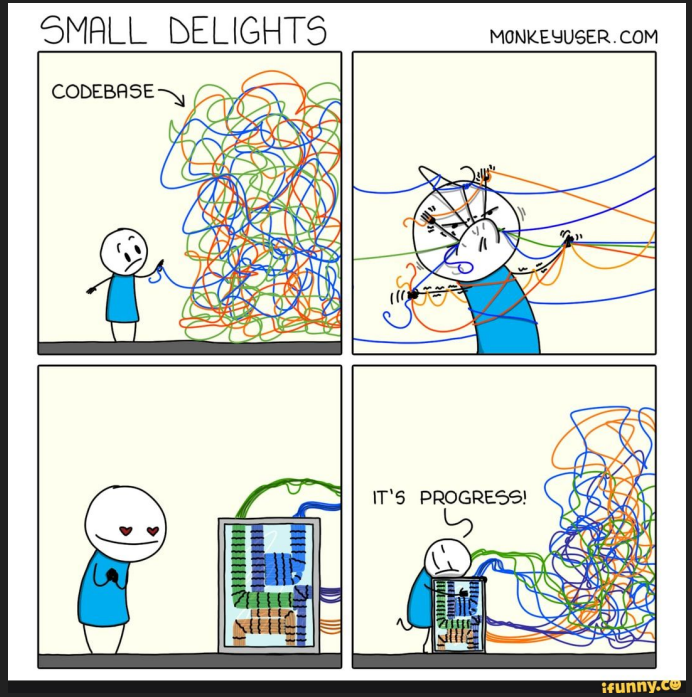
¿Qué no es refactorizar?

- Corregir bugs que se encuentren en el código.
- Optimizar.
- Hacer el código más testeable.
- Cuando se reestructuran muchas cosas a la vez para que el código quede limpio.
- Cuando se habla de `refactorizar la documentación`.



¿Cuándo refactorizar?

Antes de incluir un nuevo feature o actualizar una parte del código existente.



¿Cómo refactorizar?

Usando las diferentes recipes para cada caso según corresponda.

Refactoring Recipes

Change Function Declaration

Add Parameter • Change Signature • Remove Parameter • Rename Function • Rename Method

Combine Functions into Class

Combine Functions into Transform

Encapsulate Variable

Encapsulate Field • Self-Encapsulate Field

Extract Function

Extract Method

Extract Variable

Introduce Explaining Variable

Inline Function

Inline Method

Inline Variable

Inline Temp

Introduce Parameter Object

Rename Variable

Ejemplos

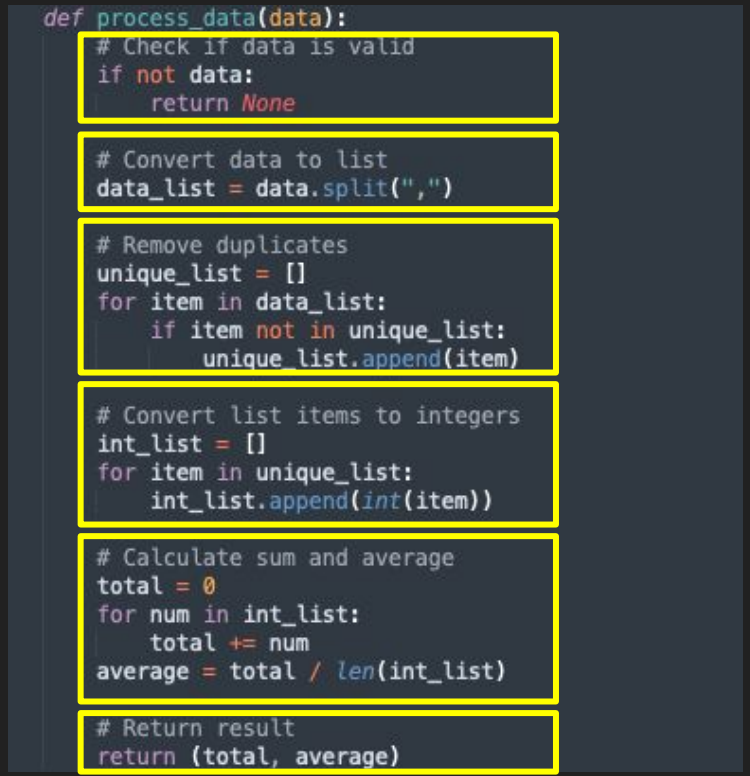
Long Method

```python
def process_data(data):
    if not data:
        return None

    data_list = data.split(",")

    unique_list = []
    for item in data_list:
        if item not in unique_list:
            unique_list.append(item)

    int_list = []
    for item in unique_list:
        int_list.append(int(item))

    total = 0
    for num in int_list:
        total += num
    average = total / len(int_list)

    return (total, average)
```

Long Method

```python
def process_data(data):
    # Check if data is valid
    if not data:
        return None

    # Convert data to list
    data_list = data.split(",")

    # Remove duplicates
    unique_list = []
    for item in data_list:
        if item not in unique_list:
            unique_list.append(item)

    # Convert list items to integers
    int_list = []
    for item in unique_list:
        int_list.append(int(item))

    # Calculate sum and average
    total = 0
    for num in int_list:
        total += num
    average = total / len(int_list)

    # Return result
    return (total, average)
```

Long Method


def process_data(data):
# Check if data is valid
if not data:
return None
# Convert data to list
data_list = data.split(",")
# Remove duplicates
unique_list = []
for item in data_list:
if item not in unique_list:
unique_list.append(item)
# Convert list items to integers
int_list = []
for item in unique_list:
int_list.append(int(item))
# Calculate sum and average
total = 0
for num in int_list:
total += num
average = total / len(int_list)
# Return result
return (total, average)

Refactoring Long Method usando Extract Method

```python
def process_data(data):
    # Check if data is valid
    if not data:
        return None

    # Convert data to list
    data_list = convert_to_list(data)

    # Remove duplicates
    unique_list = remove_duplicates(data_list)

    # Convert list items to integers
    int_list = convert_to_integers(unique_list)

    # Calculate sum and average
    total, average = calculate_statistics(int_list)

    # Return result
    return (total, average)
```

```python
def convert_to_list(data):
    return data.split(",")
```

```python
def remove_duplicates(data_list):
    unique_list = []
    for item in data_list:
        if item not in unique_list:
            unique_list.append(item)
    return unique_list
```

```python
def convert_to_integers(unique_list):
    return [int(item) for item in unique_list]
```

```python
def calculate_statistics(int_list):
    total = sum(int_list)
    average = total / len(int_list)
    return total, average
```

Magic Numbers

```
def potentialEnergy(mass, height):
    return mass * height * 9.81
```

Magic Numbers

```
def potentialEnergy(mass, height):
    return mass * height * 9.81
```

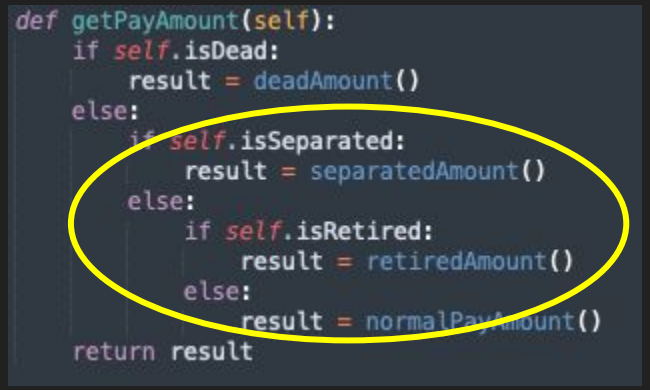
Refactorizar Magic Numbers con Symbolic Constant

```
GRAVITATIONAL_CONSTANT = 9.81

def potentialEnergy(mass, height):
    return mass * height * GRAVITATIONAL_CONSTANT
```

Condicionales anidados

```
def getPayAmount(self):
    if self.isDead:
        result = deadAmount()
    else:
        if self.isSeparated:
            result = separatedAmount()
        else:
            if self.isRetired:
                result = retiredAmount()
            else:
                result = normalPayAmount()
    return result
```

Refactorizar Condicionales anidados con Guard Clauses



Refactorizar Condicionales anidados con Guard Clauses

```
def getPayAmount(self):
    if self.isDead:
        return deadAmount()
    if self.isSeparated:
        return separatedAmount()
    if self.isRetired:
        return retiredAmount()
    return normalPayAmount()
```

Recursos

- Libros
  - Refactoring: Improving the design of the existing code - Martin Fowler
    - 1 y 2 edición
  - Clean Code - Robert C. Martin
    - cap. 17 - code smells
- Sitios web
  - Refactoring
    - https://refactoring.com/catalog/
  - Refactoring Guru
    - https://refactoring.guru/es
- Artículos
  - Bad Code Smells Taxonomy - Mmantyla
    - https://mmantyla.github.io/BadCodeSmellsTaxonomy
  - Smells Refactoring Quick Reference Guide
    - https://www.industriallogic.com/img/blog/2005/09/smellstorefactorings.pdf

Recursos

Repositorio

Code Smells & Refactoring - Flisol ECU 2023 Talk

https://github.com/davcortez/refactoring-code-smells