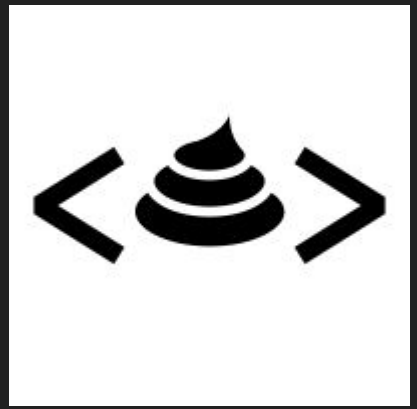
Code smells & Refactoring



Whoami

- David Cortez
- Desarrollador Python.
- Entusiasta de la seguridad informática.

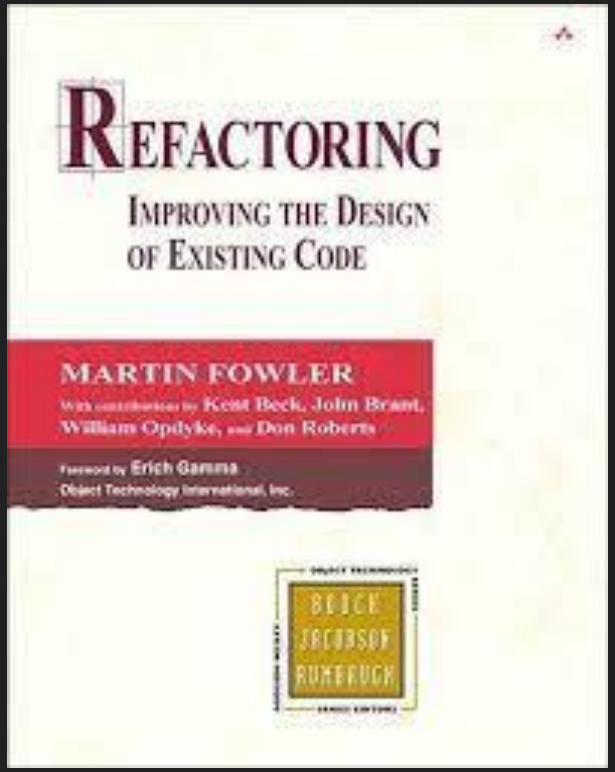
@davcortez

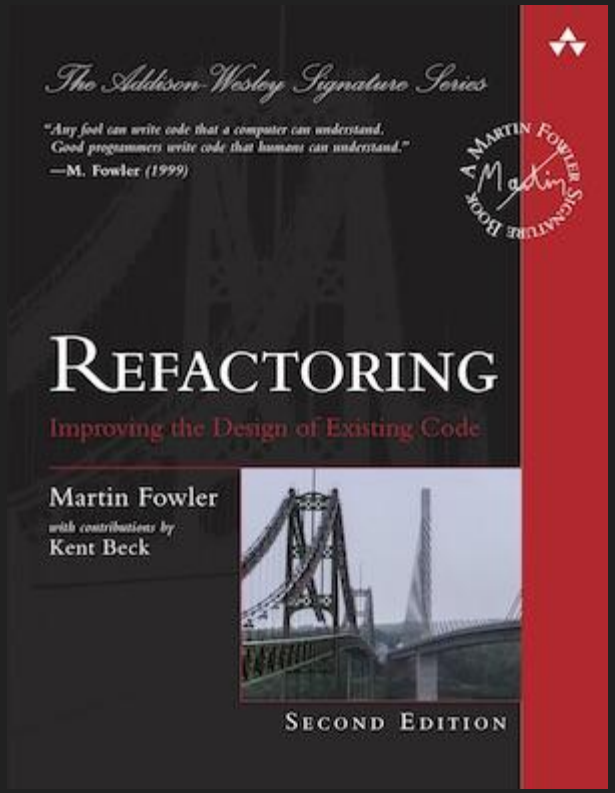
¿Qué son los Code Smells?

- 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.
- Baja calidad del código.
- Alto costo de mantenibilidad.
- Riesgo de introducir otros errores.

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

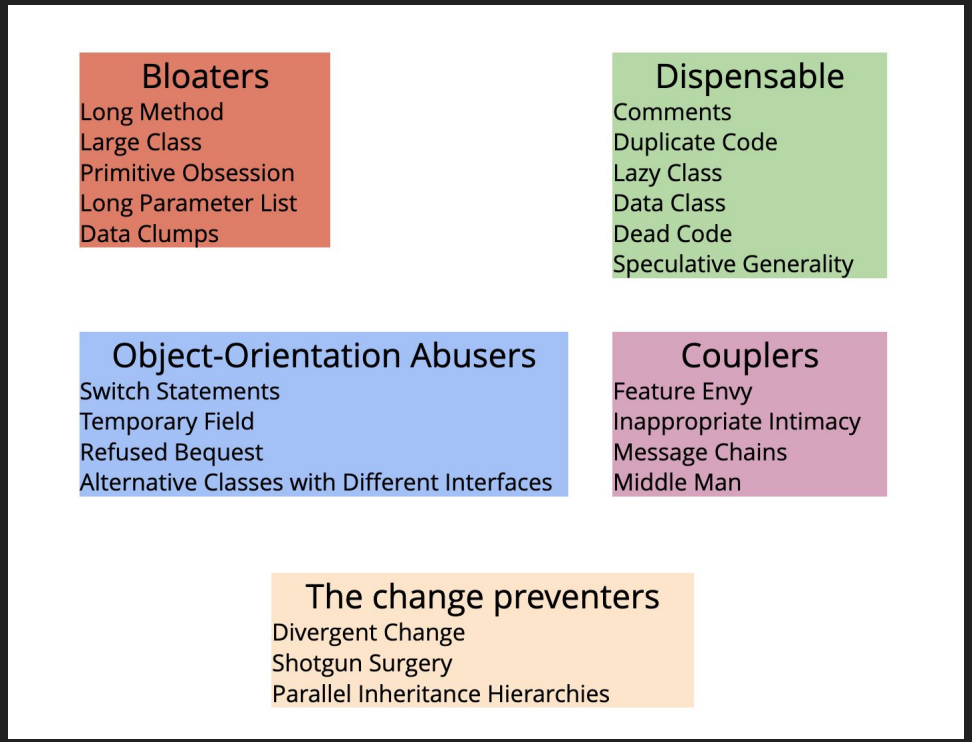
Shotgun Surgery

Speculative Generality

Switch Statements

Temporary Field

A Taxonomy for "Bad Code Smells" - Mantyla & Lassenius



Bloaters

Demasiado grande para entender y manejar.

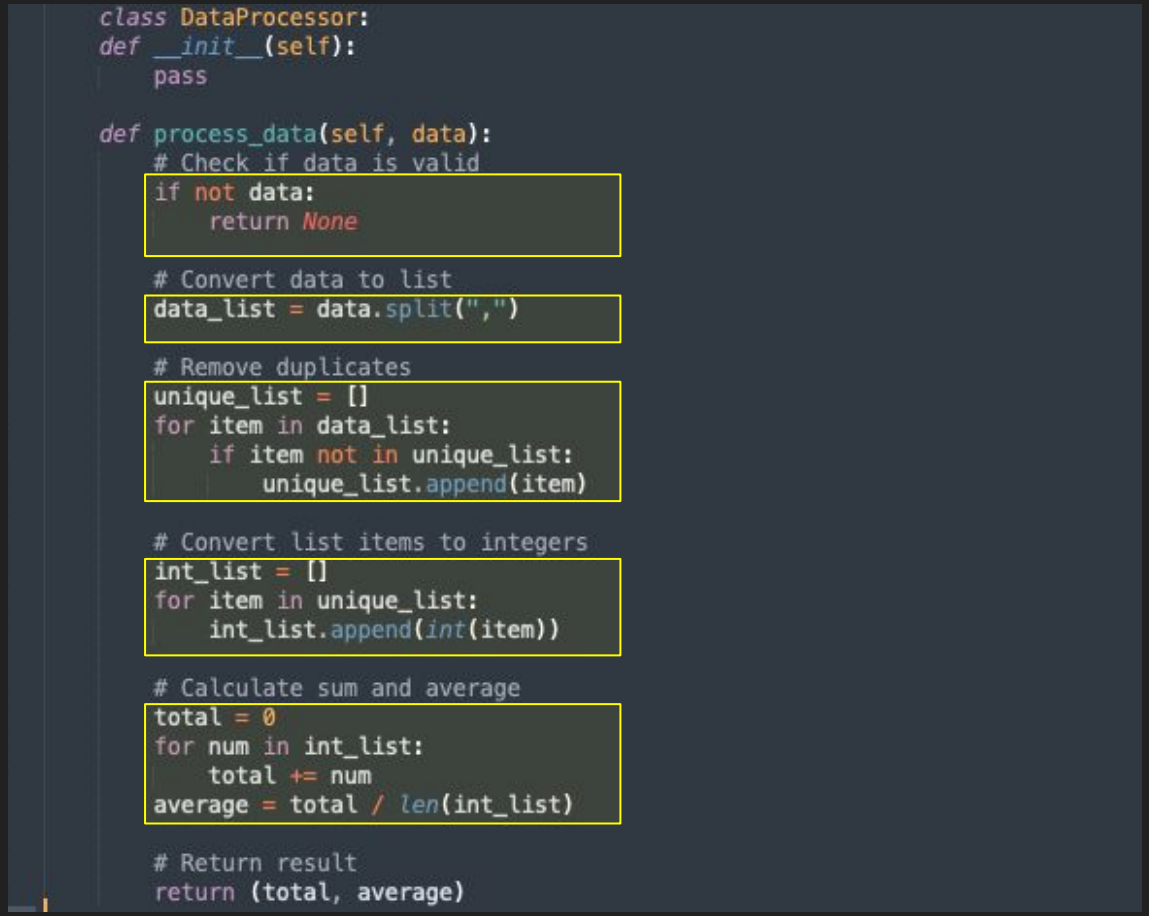
Long Method

Un método que contiene muchas líneas de código.

```python
class DataProcessor:
def __init__(self):
    pass

def process_data(self, 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

Un método que contiene muchas líneas de código.



```python
class DataProcessor:
def __init__(self):
    pass

def process_data(self, 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)
```

Large Class

Una clase que hace muchas cosas.

```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...
```

Large Class

Una clase que hace muchas cosas.

```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

Variables que deberían ser empaquetadas en un objeto.

```
def colorize(red: int, green: int, blue: int) -> str:
    """Returns a string representation of an RGB color."""
    color_hex = f"#{red:02x}{green:02x}{blue:02x}"
    return color_hex
```

Data Clumps

Variables que deberían ser empaquetadas en un objeto.

```python
def colorize(red: int, green: int, blue: int) -> str:
    """Returns a string representation of an RGB color."""
    color_hex = f"#{red:02x}{green:02x}{blue:02x}"
    return color_hex
```

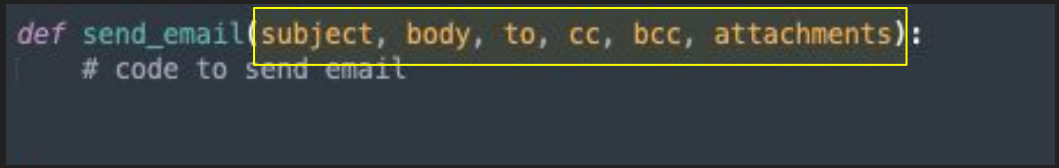
Long Parameter List

Más de 3 o 4 parámetros para un método.

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

Long Parameter List

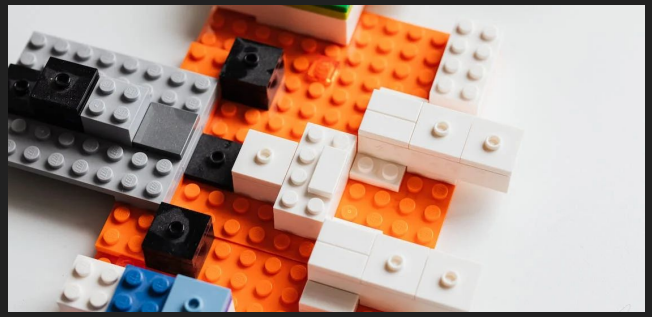
Más de 3 o 4 parámetros para un método.



Primitive Obsession

Cuando el código depende demasiado de datos primitivos.

```
birthday_date: str = "1998-03-04"
name_day_date: str = "2021-03-20"
```



Object-Orientation Abusers

Aplicación incompleta o incorrecta de los principios de OOP.

Switch Statements

Operador switch complejo o una serie de declaraciones if.

```
class Exporter:
    def export(self, export_format: str):
        if export_format == 'wav':
            self.exportInWav()
        elif export_format == 'flac':
            self.exportInFlac()
        elif export_format == 'mp3':
            self.exportInMp3()
        elif export_format == 'ogg':
            self.exportInOgg()
```

Switch Statements

Operador switch complejo o una serie de declaraciones if.

Temporary Field

Campos que solo se usan una vez

```python
@dataclass
class MyDateTime:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
        self.full_date = f"{year}, {month}, {day}"

    def foo(self):
        ...

    def goo(self):
        ...

    def hoo(self):
        ...

    def __str__(self):
        return self.full_date
```

Temporary Field

Campos que solo se usan una vez

```python
@dataclass
class MyDateTime:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
        self.full_date = f"{year}, {month}, {day}"

    def foo(self):
        ...

    def goo(self):
        ...

    def hoo(self):
        ...

    def __str__(self):
        return self.full_date
```

Refused Bequest

Cuando una subclase hereda de una clase padre pero solo utiliza un subconjunto de los métodos implementados en el padre.

```
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
```

Refused Bequest

Cuando una subclase hereda de una clase padre pero solo utiliza un subconjunto de los métodos implementados en el padre.

```ruby
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
```

Alternative Classes with Different Interfaces

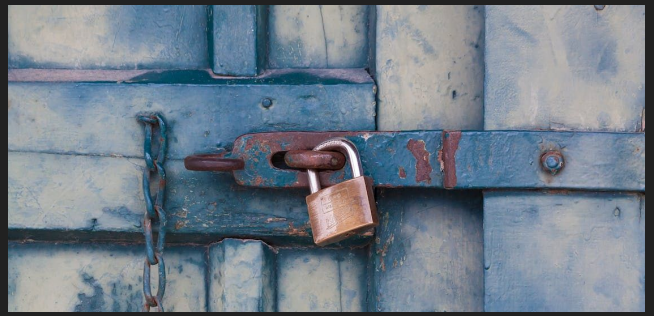
Si dos clases tienen la misma funcionalidad pero implementaciones diferentes.

```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
```

Alternative Classes with Different Interfaces

Si dos clases tienen la misma funcionalidad pero implementaciones diferentes.

```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

Cuando un cambio de código en un lugar nos obliga a realizar muchos cambios en otros lugares.

Shotgun Surgery

Cambios entre clases

```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))
```

Shotgun Surgery

Cambios entre clases

```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

Cambios entre métodos

```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
```

Divergent Changes

Cambios entre métodos

```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

Cuando se implementan interfaces paralelas

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

    def eat(self):
        ...

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

    def give_birth(self):
        ...

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

    def lay_eggs(self):
        ...

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

    def meow(self):
        ...

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

    def cluck(self):
        ...
```

Parallel Inheritance Hierarchies

Cuando se implementan interfaces paralelas

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

    def eat(self):
        ...

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

    def give_birth(self):
        ...

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

    def lay_eggs(self):
        ...

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

    def meow(self):
        ...

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

    def cluck(self):
        ...
```

Dispensables

Cosas innecesarias que pueden removerse.

Comments

Comentarios sin valor: Obsoletos, explican el qué hace más no el porqué.

```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
```

Comments

Comentarios sin valor: Obsoletos, explican el qué hace más no el porqué.

```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

El mismo código
múltiples veces.

```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
```

Duplicate Code

El mismo código
múltiples veces.

```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
```

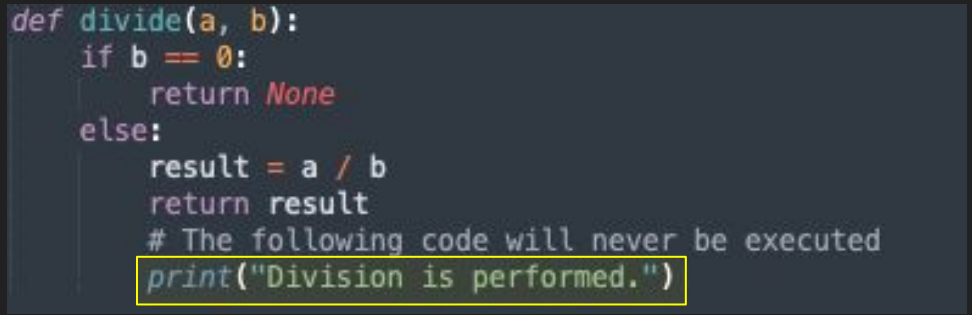
Dead Code

Código que ya no es utilizado.

```python
def divide(a, b):
    if b == 0:
        return None
    else:
        result = a / b
        return result
        # The following code will never be executed
        print("Division is performed.")
```

Dead Code

Código que ya no es utilizado.



Lazy Class

Una clase que no tiene suficientes responsabilidades

```
class Person:
    pass
```

Speculative Generality

Preparación excesiva
para el futuro.

```
class Animal:
    health: int

class Human(Animal):
    name: str
    attack: int
    defense: int

class Swordsman(Human):
    ...

class Archer(Human):
    ...

class Pikeman(Human):
    ...
```

Speculative Generality

Preparación excesiva
para el futuro.

Contexto: Juego de peleas medievales.

```
class Animal:
    health: int

class Human(Animal):
    name: str
    attack: int
    defense: int

class Swordsman(Human):
    ...

class Archer(Human):
    ...

class Pikeman(Human):
    ...
```

Couplers

Demasiado o muy poco acoplamiento.

Feature Envy

Cuando una clase implementa características de otra

```python
@dataclass(frozen=True)
class ShoppingItem:
    name: str
    price: float
    tax: float


class Order:
    ...
    def get_bill_total(self, items: list[ShoppingItem]) -> float:
        return sum([item.price * item.tax for item in items])

    def get_receipt_string(self, items: list[ShoppingItem]) -> list[str]:
        return [f"{item.name}: {item.price * item.tax}$" for item in items]

    def create_receipt(self, items: list[ShoppingItem]) -> float:
        bill = self.get_bill_total(items)
        receipt = self.get_receipt_string(items).join('\n')
        return f"{receipt}\nBill {bill}"
```

Feature Envy

Cuando una clase implementa características de otra

```python
@dataclass(frozen=True)
class ShoppingItem:
    name: str
    price: float
    tax: float


class Order:
    ...
    def get_bill_total(self, items: list[ShoppingItem]) -> float:
        return sum([item.price * item.tax for item in items])

    def get_receipt_string(self, items: list[ShoppingItem]) -> list[str]:
        return [f"{item.name}: {item.price * item.tax}$" for item in items]

    def create_receipt(self, items: list[ShoppingItem]) -> float:
        bill = self.get_bill_total(items)
        receipt = self.get_receipt_string(items).join('\n')
        return f"{receipt}\nBill {bill}"
```

Inappropriate Intimacy

Ocurre cuando dos clases están estrechamente vinculadas entre sí.

```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
```

Inappropriate Intimacy

Ocurre cuando dos clases están estrechamente vinculadas entre sí.

```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
```

Middle Man

Cuando una clase tiene como responsabilidad delegar el trabajo a otra.

```
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)

name = manager.get_employee_name()
```

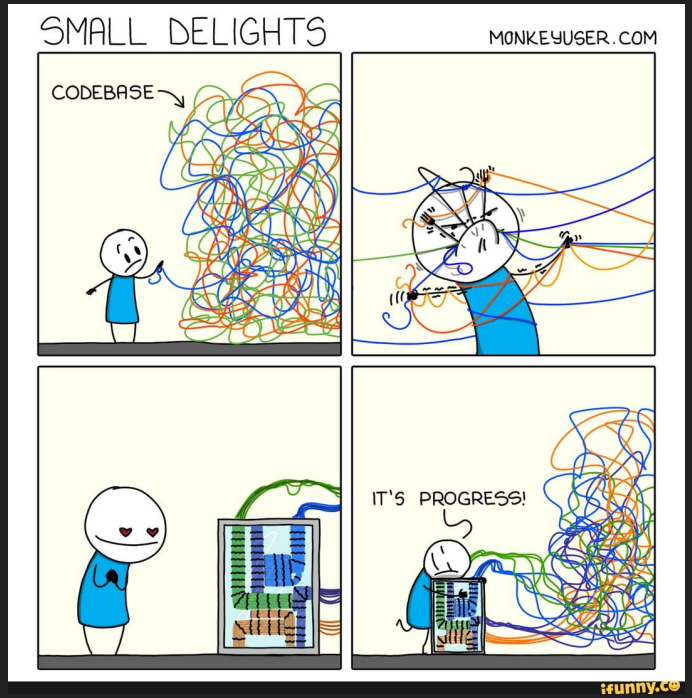
Middle Man

Cuando una clase tiene como responsabilidad delegar el trabajo a otra.

```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)

name = manager.get_employee_name()
```

¿Qué es Refactoring?

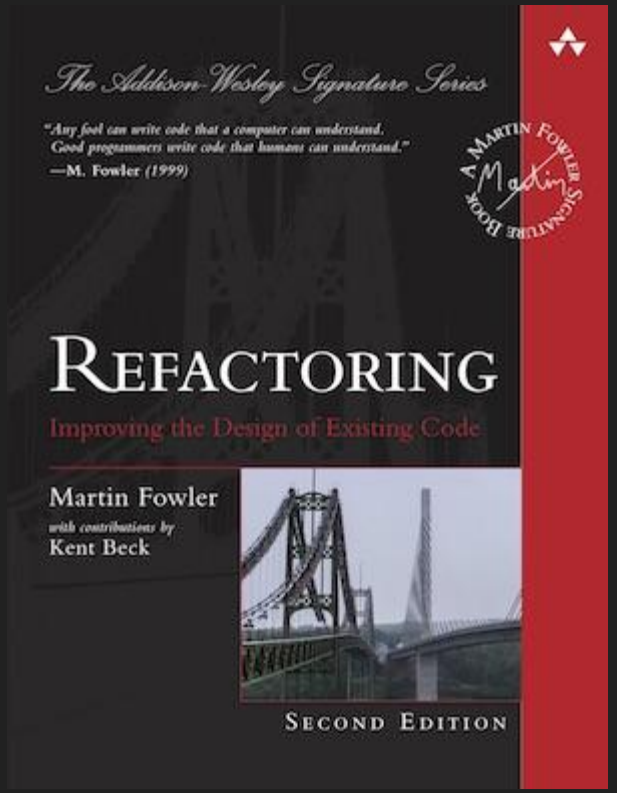
Técnica de reestructuración de código que no modifica su comportamiento externo



¿Qué es Refactoring?

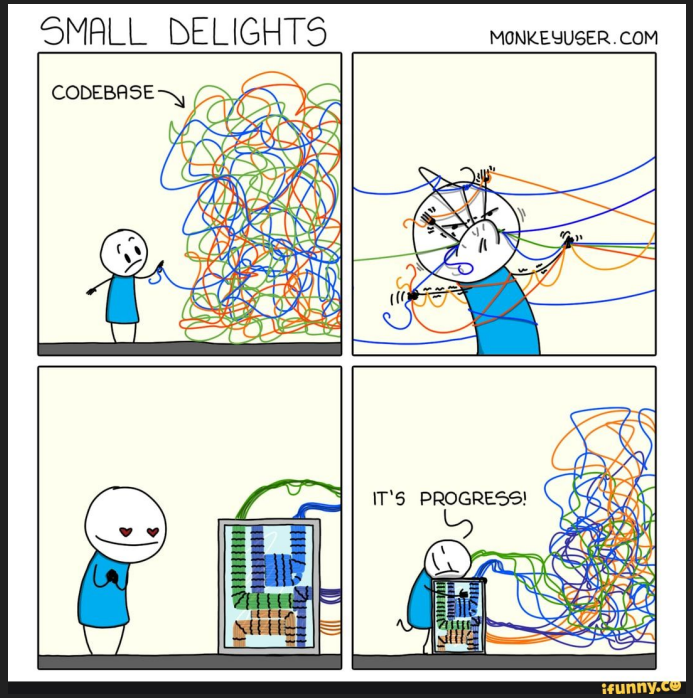
/noun/ A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

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



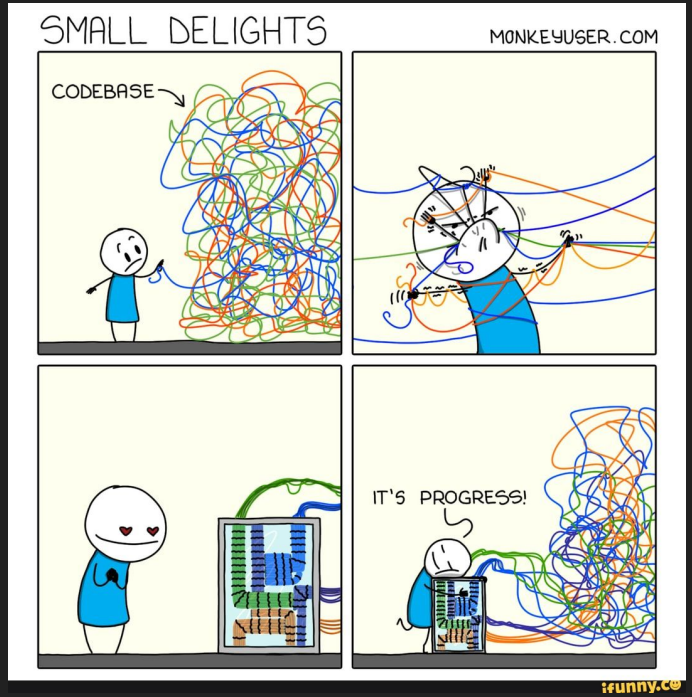
¿Qué no es refactorizar?

- Corregir bugs que se encuentren en el código.
- Optimizar.
- Hacer el código más fácil de testear.
- Reescribir código.



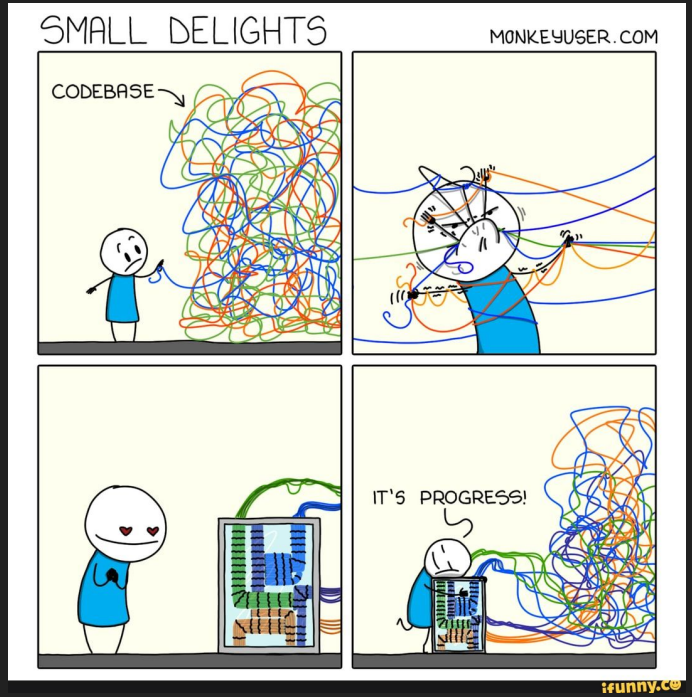
¿Cuándo refactorizar?

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



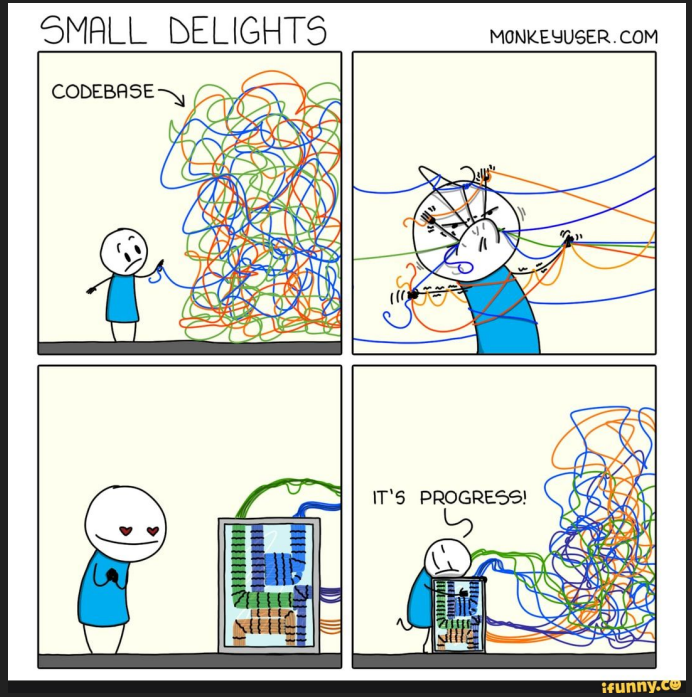
¿Cuándo refactorizar según Fowler? (situaciones)

- Fase preparatoria.
- Comprensividad.
- Incremental.
- De Oportunidad.



¿Cuándo refactorizar según Fowler? (situaciones)

- De forma planificada.
- A largo plazo.
- Revisiones de código.



¿Cómo refactorizar?

Usando las diferentes `recetas` 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

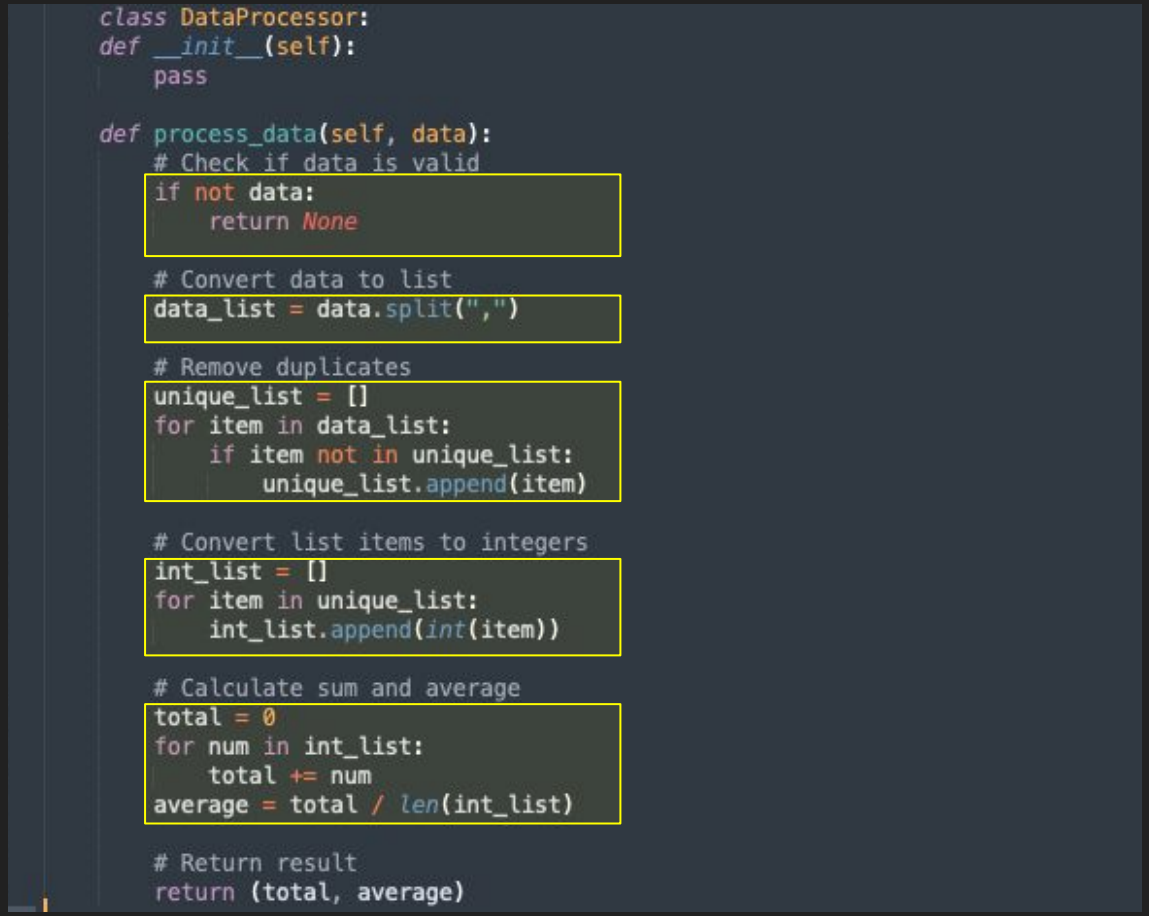
Long Method

Un método que contiene muchas líneas de código.

```python
class DataProcessor:
def __init__(self):
    pass

def process_data(self, 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)
```

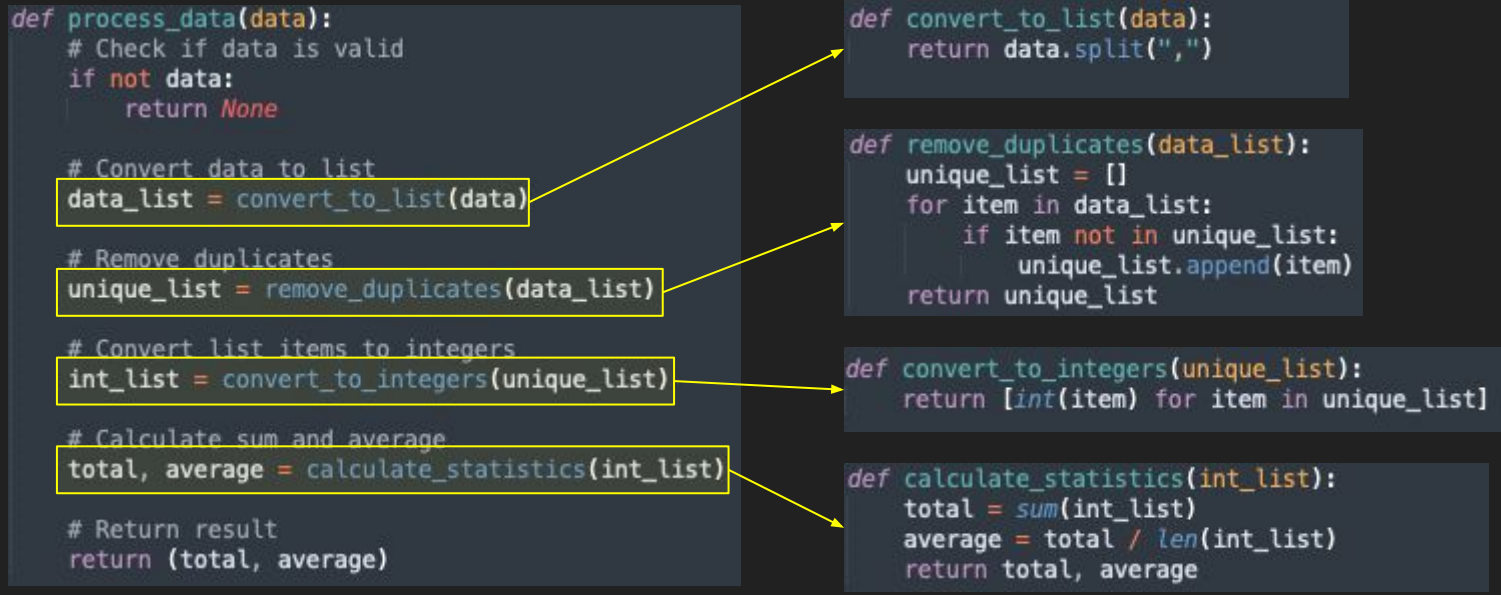
Long Method

Un método que contiene muchas líneas de código.



```python
class DataProcessor:
def __init__(self):
    pass

def process_data(self, 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 usando Extract Method



Magic Numbers

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

Magic Numbers

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

Refactorizar 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
```

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

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

Ahora, a practicar...

Repositorio

Code Smells & Refactoring - Flisol ECU 2023 Talk
https://github.com/davcortez/refactoring-code-smells

@davcortez

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

@davcortez

Recursos

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

@davcortez