IIC2113 - Diseño Detallado de Software

CLASE 3 - BUENAS PRÁCTICAS DE DESARROLLO

Pontificia Universidad Católica de Chile
2020-2

Índice

04. Code Smells

Code Smells

"***A code smell is a surface indication that usually corresponds to a deeper problem in the system***" - Martin Fowler.

- El término fue acuñado por Kent Beck y se hizo famoso con el libro Refactoring: Improving the Design of Existing Code de Martin Fowler.
- **No son bugs**: el programa funciona correctamente, pero su débil diseño dificulta el desarrollo e incrementa la posibilidad de generar bugs.

https://refactoring.guru/

Página recomendada para estudiar code smells, refactoring y design patterns

Code Smells

Los Code Smells se pueden clasificarse en 5 tipos:

- Bloaters
- Object-Orientation Abusers
- Change Preventers
- Dispensables
- Couplers

Bloaters

Representan código, métodos y clases que han crecido a tal punto que es difícil trabajar con ellos

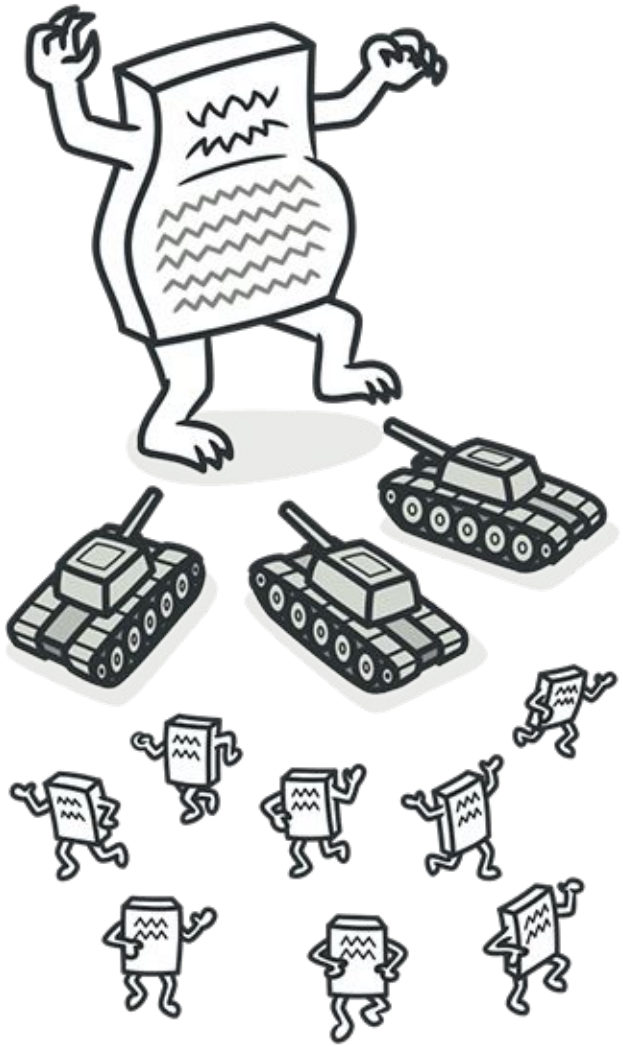


Ilustración extraída de https://refactoring.guru/

Bloaters - Long Method

- **Síntomas**: Un método contiene excesivas líneas de código. Si tiene más de 10 puede haber un problema.
- **Razones del problema**:
  - La misma persona que escribe el código lo mantiene.
  - “Es más fácil” agregar líneas a un método existente que crear nuevos.
- **Soluciones**:
  - Descomponer en más funciones.
  - Reducir variables y parámetros.
  - Mover el método a su propio objeto.

Descomponer en más funciones

```java
// BAD
public void printOwing() {
  printBanner();

  Console.WriteLine("name: " + name);
  Console.WriteLine("amount: " + getOutstanding());
}

// GOOD
public void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

public void printDetails(double outstanding) {
  Console.WriteLine("name: " + name);
  Console.WriteLine("amount: " + outstanding);
}
```

```ruby
# BAD
def print_owing
  print_banner

  puts "name: #{@name}"
  puts "amount: #{get_outstanding}"
end

# GOOD
def print_owing
  print_banner
  print_details(get_outstanding)
end

def print_details(outstanding)
  puts "name: #{@name}"
  puts "amount: #{outstanding}"
end
```

Reducir variables y parámetros

```java
// BAD
public double calculateTotal() {
  double basePrice = quantity * itemPrice;
  if (basePrice > 1000) {
    return basePrice * 0.95;
  }
  else {
    return basePrice * 0.98;
  }
}

// GOOD
public double calculateTotal() {
  if (basePrice() > 1000) {
    return basePrice() * 0.95;
  }
  else {
    return basePrice() * 0.98;
  }
}
double basePrice() {
  return quantity * itemPrice;
}
```

```ruby
# BAD
def calculate_total
  base_price = @quantity * @item_price

  if base_price > 1000
    base_price * 0.95
  else
    base_price * 0.98
  end
end

# GOOD
def calculate_total
  if base_price > 1000
    base_price * 0.95
  else
    base_price * 0.98
  end
end

def base_price
  @base_price |= @quantity * @item_price
end
```

Reducir variables y parámetros

```
1   // BAD
2   amountInvoiced(Date start, Date end)
3
4   // GOOD
5   amountInvoiced(DateRange date)
```

```
1   // BAD
2   int low = daysTempRange.getLow();
3   int high = daysTempRange.getHigh();
4   boolean withinPlan = plan.withinRange(low, high);
5
6   // GOOD
7   boolean withinPlan = plan.withinRange(daysTempRange);
8
```

Mover el método a su propio objeto

```
1  // BAD
2  class Order {
3    // ...
4    public double price() {
5      double primaryBasePrice;
6      double secondaryBasePrice;
7      double tertiaryBasePrice;
8    }
9  }
```

No necesariamente disminuirá la cantidad de líneas.

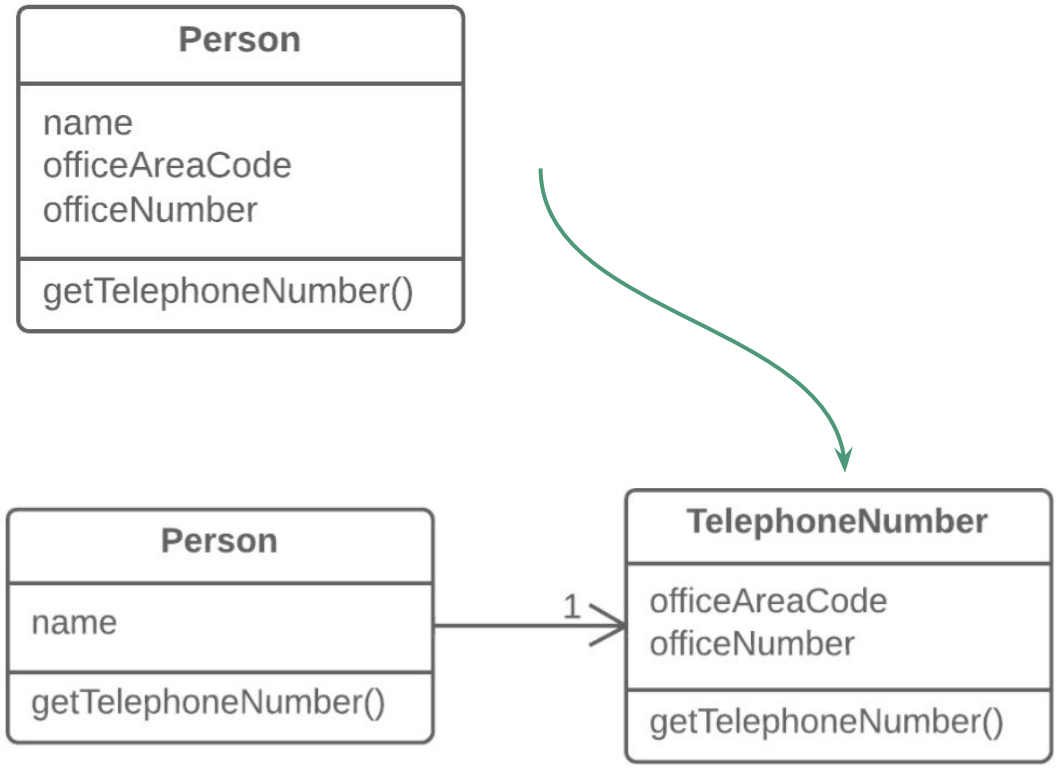
```
1  // GOOD
2  class Order {
3    // ...
4    public double price() {
5      return new PriceCalculator(this).compute();
6    }
7  }
8
9  class PriceCalculator {
10   private double primaryBasePrice;
11   private double secondaryBasePrice;
12   private double tertiaryBasePrice;
13
14   public PriceCalculator(Order order) {
15     // Copy relevant information from the
16     // order object.
17   }
18
19   public double compute() {
20     // Perform long computation.
21   }
22 }
```

Bloaters - Large Class

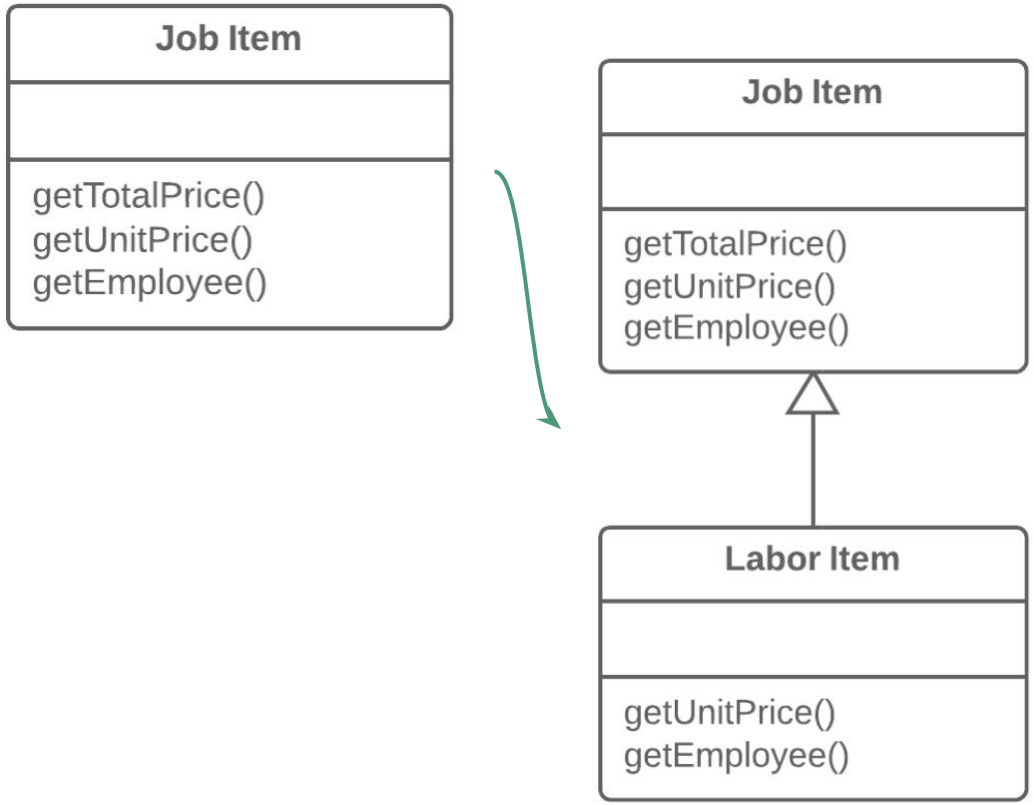
- **Síntomas**: Una clase contiene excesivas líneas de código, atributos y métodos.
- **Razones del problema**:
  - Las clases crecen en conjunto con la aplicación con baja cohesión.
  - “Es más fácil” agregar líneas a un método existente que crear nuevos.
- **Soluciones**:
  - Descomponer en múltiples clases con asociación o composición.
  - Descomponer en múltiples clases con herencia.

Descomponer la clase

Descomponer usando asociación



Descomponer usando herencia

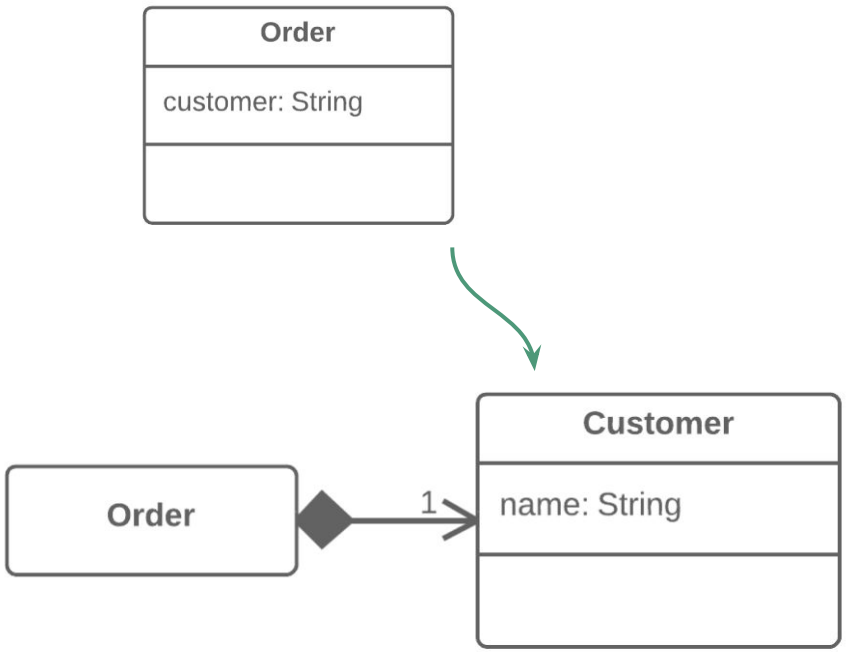


Bloaters - Primitive Obsession

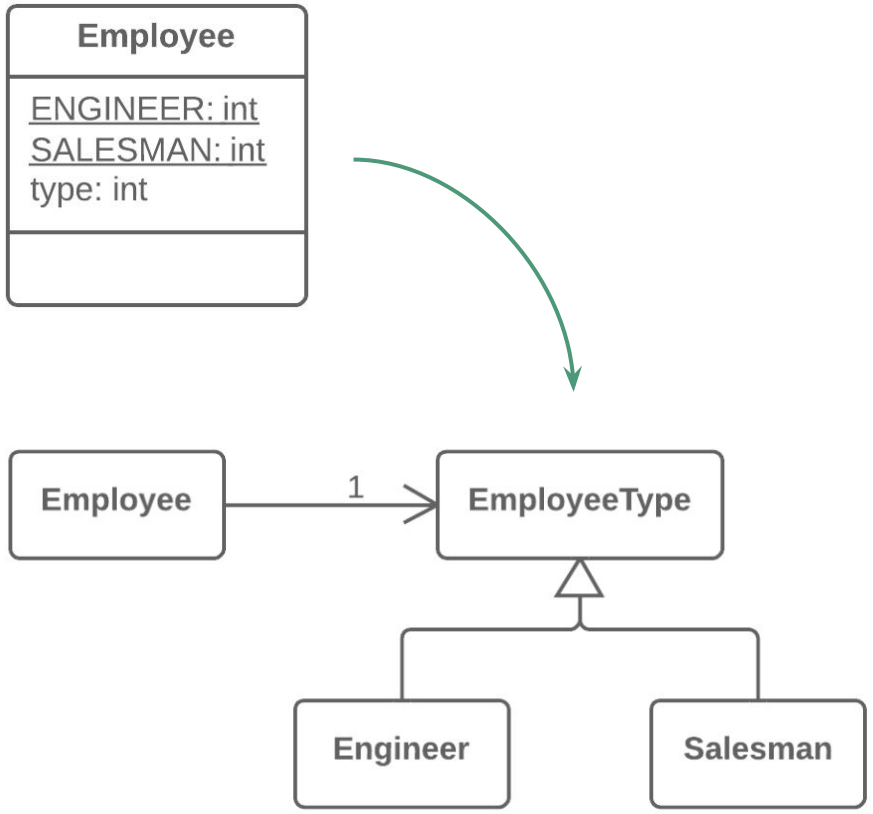
- **Síntomas**:
  - Uso de variables primitivas en vez de objetos pequeños.
  - Uso de constantes para guardar información.
- **Razones del problema**:
  - Se agregan atributos a una clase a medida que se necesitan, sin considerar que se pueden agrupar en un objeto.
  - Mala elección de tipos primitivos para representar una estructura de datos.
- **Soluciones**:
  - Convertir conjuntos de variables en objetos pequeños.

Convertir conjuntos de variables en objetos

Ejemplo 1:



Ejemplo 2:



Bloaters - Long Parameter List

- **Síntomas**: Más de 3 o 4 parámetros por método. Es difícil entender listas extensas de parámetros.
- **Razones del problema**:
  - Se agrupa funcionalidad en un sólo método.
  - Se pasan parámetros a un método directamente, y no a través de objetos que contienen la información o llamar a otros métodos para obtenerlos.
- **Soluciones**:
  - Agrupar conjuntos de variables en objetos pequeños.
  - Conviene ignorarlo si la separación implicaría mucha interdependencia entre los nuevos objetos creados.

Bloaters - Data Clumps

- **Síntomas**: Diferentes partes de una aplicación contienen una definición recurrente de variables.
- **Razones del problema**:
  - Una pobre estructura de la aplicación, o copy-paste programming.
  - No se agrupan valores inseparables en un objeto.
- **Soluciones**:
  - Agrupar conjuntos de variables en objetos pequeños.
  - Conviene ignorarlo si se generan dependencias no deseadas entre clases.