Padrões e Desenho de Software - Code Smells

Autor: Vinícius Benite Ribeiro [82773], 2020-06-01

Professor: José Luís Oliveira

DETI - Universidade de Aveiro

1 Introdução

1.1 Objetivo

O objetivo deste breve relatório é explorar conceitos sobre *Code Smells,* em Engenharia de Software, nomeadamente os Bloaters, violações sobre os princípios de POO e os Changes Preventers e também, apresentar um exemplo e solução a um code smell específico.

1.2 O que é Code Smell?

Geralmente, um code smell é um problema superficial que, geralmente, pode corresponder a problemas maiores no código[1].

Há dois pontos importantes a notar aqui: primeiramente, code smell é algo que é fácil de se notar ao escrever código novo e, nem sempre, code smell é um indicativo que o código apresenta um erro mais grave. Por exemplo, um método muito longo pode ser totalmente coerente e necessário, mas é considerado como code smell.

Pessoas inexperientes em engenharia de software podem notar smells com facilidade, mesmo que elas não saibam que esse smell pode levar a um problema maior ou não. Sendo assim, a detecção de code smells e como resolvê-los, é uma ótima forma de “melhorar” novos programadores

2 Tipos de Code Smells

2.1 Bloaters

Bloaters são métodos, classes ou lista de parâmetros que são muito grandes, tornando-se difícil a sua manutenção. Tornam-se evidentes quanto maior o tempo de vida do código, especialmente se ninguém tratar deste problema. Geralmente, a solução para estes tipos de problemas é extrair o método, tornando as classes e métodos menores e mais fáceis de ler.

```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOuts
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstan
}
```

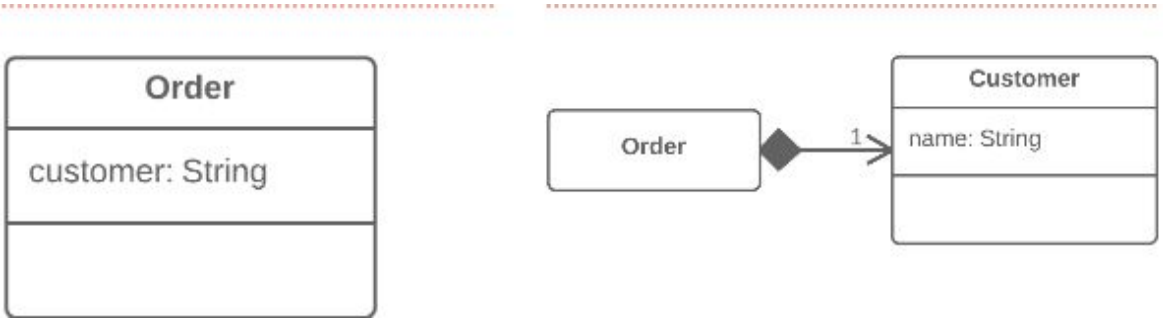
Refactoring Guru

Data clumps também se enquadra nesta categoria. Diversas partes do código contém variáveis idênticas, por exemplo, conexões à base de dados.

Devemos olhar para o código que apresenta esse sintoma e pensar em mover esse código para uma nova classe. Outra solução é notar que uma classe anda a fazer o trabalho de duas. Podemos dividir os métodos e objetos dessa classe:

Refactoring Guru

Finalmente, temos o Primitive Obsesion. Primitivos são usados, geralmente, para simular tipos. É muito mais fácil e rápido criar um objeto primitivo do que uma classe nova. Ao longo do processo de coding, podemos acabar com vários objetos primitivos, tornando as classes muito grandes desnecessariamente. Criar uma nova classe e colocar os campos e seus comportamentos nela e guardar o objeto na sua classe original pode ser a solução.



Refactoring Guru

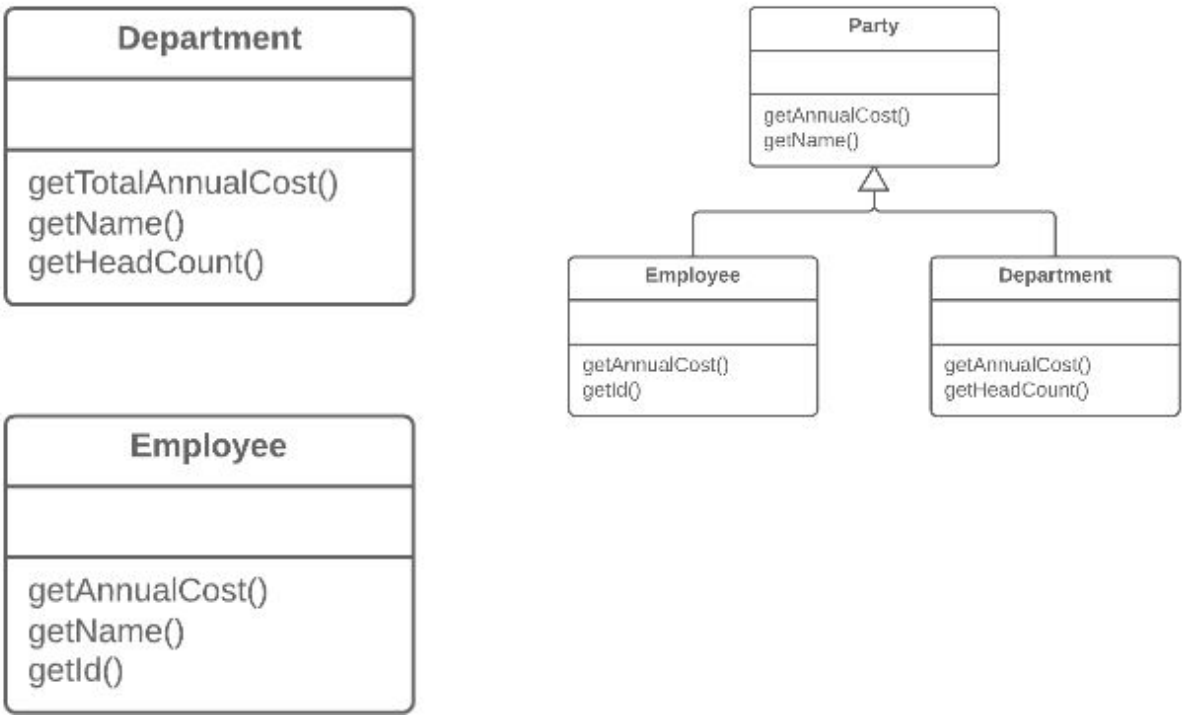
2.2 Violações sobre os princípios de POO

Enquadram-se nesta categoria todos as violações dos princípios de programação orientada a objetos.

Classes diferentes com métodos iguais (com nomes diferentes) é um problema. O programador pode não saber que tal método já existia antes de criá-lo.

Se classes filhas não usam todos os métodos da superclasse, algo está errado. Os métodos podem ser inúteis ou precisam ser reescritos. Chamamos esse smell de Refused Bequest.

Se a herança não faz sentido nenhum, devemos substituir a herança por delegação. Caso contrário, devemos nos livrar dos métodos desnecessários na subclasse e colocá-los em outra subclasse, sendo que ambas derivam da superclasse anterior.



Refactoring Guru

Muitos switches no código podem indicar que algo está errado. Se uma nova condição for adicionada, temos que procurar e alterar os switches no código todo. Geralmente, se pensarmos em usar um switch, polimorfismo pode ser a solução.

Objetos temporários são criados em programas que requerem uma grande quantidade de inputs. Por ser mais fácil, o programador cria objetos para guardar dados na classe. Esses objetos só serão usados nessa classe e somente nela. Esse tipo de código é difícil de ler. Uma solução prática é substituir objetos

por métodos:

```
class Order {
  // ...
  public double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // Perform long computation.
  }
}
```

```
class Order {
  // ...
  public double price() {
    return new PriceCalculator(this).comp
  }
}

class PriceCalculator {
  private double primaryBasePrice;
  private double secondaryBasePrice;
  private double tertiaryBasePrice;

  public PriceCalculator(Order order) {
    // Copy relevant information from the
    // order object.
  }

  public double compute() {
    // Perform long computation.
  }
}
```

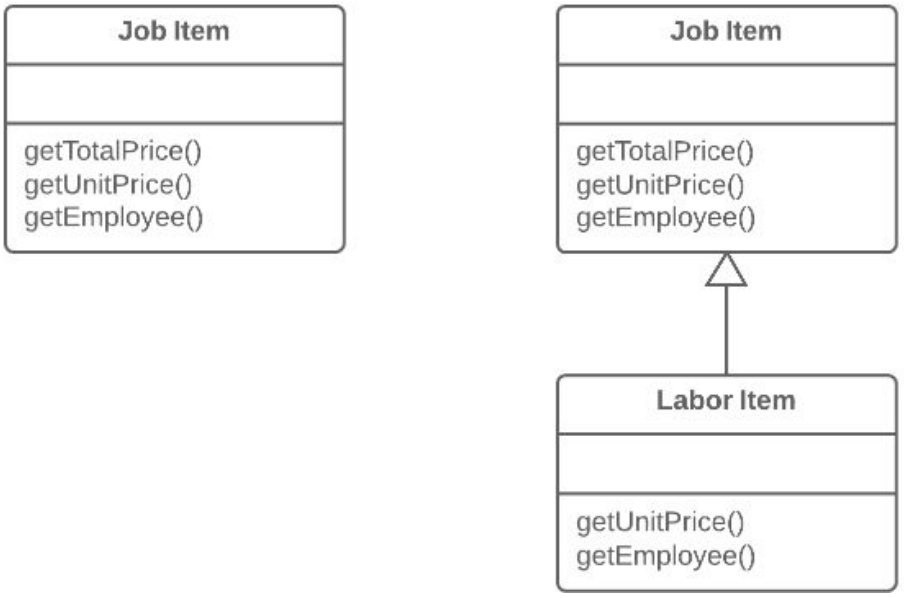
Refactoring Guru

2.3 Change preventers

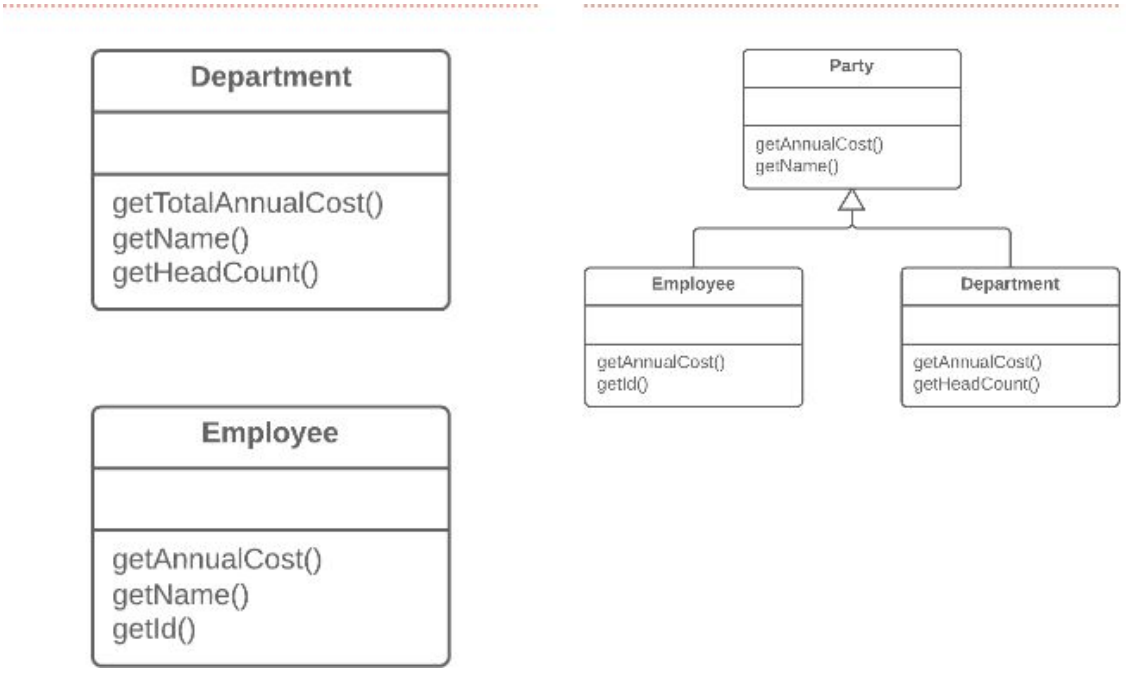
Esse tipo de smell significa que, se você fizer uma mudança em um determinado ponto do código, você tem que fazer essa mudança em outros pontos dependentes. IDE's modernas provêm soluções de refactors eficientes, porém, em códigos muito grandes, esse smell pode-se tornar complicado.

Exemplos deste smell são o divergent changes e shotgun surgery: o primeiro é quando temos diversas mudanças em uma única classe, por exemplo, se alterarmos um método desta classe, temos que alterar o restante dos métodos. Shotgun surgery é quando temos diversas mudanças em várias classes, simultaneamente.

Relativamente ao divergent changes, extrair subclasses e/ou superclasses é a solução:



Extrair subclasse



Extrair superclasse

Hierarquia paralela de herança é outro exemplo. Quando criamos uma subclasse para uma classe, somos obrigados a criar outra subclasse para outra classe qualquer. Conforme a árvore de herança cresce no programa, torna-se cada vez mais difícil fazer mudanças no código, causando duplicação de código e

desorganização. Podemos mover atributos e/ou métodos de classe para classe para tentar minimizar esse problema. Porém, ao fazer isso, às vezes, acabamos com um código ainda mais confuso. Portanto, é preciso saber quando ignorar esse problema.

3 Exemplos práticos

3.1 Switch [3]

Temos um enumerado de empregados e sua respectiva classe com métodos para calcular o salário e o bônus anual, com 2 switches, um em cada método.

```
public enum EmployeeType
{

    Worker,

    Supervisor,

    Manager

}
```

```
public float CalculateYearBonus()

{

    switch (employeeType)

    {

        case EmployeeType.Worker:

            return 0;

        case EmployeeType.Supervisor:

            return salary + salary * 0.7F;

        case EmployeeType.Manager:

            return salary + salary * 1.0F;

    }

    return 0.0F;
}
```

```
1  public class Employee
2
3  {
4
5      private float salary;
6
7      private float bonusPercentage;
8
9      private EmployeeType employeeType;
10
11
12
13     public Employee(float salary, float bonusPercentage, EmployeeType employeeType)
14
15     {
16
17         this.salary = salary;
18
19         this.bonusPercentage = bonusPercentage;
20
21         this.employeeType = employeeType;
22
23     }
24
25
26
27     public float CalculateSalary()
28
29     {
30
31         switch (employeeType)
32
33         {
34
35             case EmployeeType.Worker:
36
37                 return salary;
38
39             case EmployeeType.Supervisor:
40
41                 return salary + (bonusPercentage * 0.5F);
42
43             case EmployeeType.Manager:
44
45                 return salary + (bonusPercentage * 0.7F);
46
47         }
48
49         return 0.0F;
50
51     }
52 }
```

Podemos simplificar o código acima utilizando o padrão Strategy. O código torna-se mais conciso e legível. Só precisamos de uma interface e uma classe de empregados:

```
public interface IRemunerationCalculator

{

    float CalculateSalary(float salary);

    float CalculateYearBonus(float salary);

}
```

```
public class Employee
{

    private float salary;

    private IRemunerationCalculator remunerationCalculator;



    public Employee(float salary, IRemunerationCalculator remunerationCalculator)

    {
        this.salary = salary;

        this.remunerationCalculator = remunerationCalculator;

    }



    public float CalculateSalary()

    {

        return remunerationCalculator.CalculateSalary(salary);

    }



    public float CalculateYearBonus()

    {

        return remunerationCalculator.CalculateYearBonus(salary);

    }
}
```

4 Referências e recursos

Project resources

- Git: https://github.com/viniciusbenite/pds-tp

Reference materials

[1] https://martinfowler.com/bliki/CodeSmell.html

[2] https://refactoring.guru/refactoring/smells

[3] https://dzone.com/articles/code-smellspart-i