



Técnicas De Programação Em Plataformas Emergentes Trabalho Programático 3

Grupo 20

- Clara Marcelino Ribeiro de Sousa - 200036351
- Luiza Esteves dos Santos - 200023411
- Natan Tavares Santana - 200025449

1 - Princípios de Bom Projeto de Código e *Code Smells*

Um código bem escrito é fundamental para a manutenção, escalabilidade e sucesso de um software a longo prazo. Pete Goodliffe, em seu livro *Code Craft: The practice of Writing Excellent Code*, enfatiza a importância de princípios como simplicidade, elegância, modularidade e a preocupação com a legibilidade e documentação do código. Esses princípios são essenciais para criar um código que não apenas funcione, mas que seja fácil de entender e modificar por qualquer programador, não apenas seu autor.

Quando esses princípios são negligenciados surgem os chamados "code smells", conforme descritos por Martin Fowler e Kent Beck em *Refactoring: Improving the design of Existing Code*. *Code smells* são indicadores de que algo pode estar errado no código, como duplicação, complexidade excessiva ou falta de coesão, e embora não quebrem o código imediatamente, apontam para problemas que podem comprometer a qualidade do software. Os princípios de bom código de Goodliffe estão diretamente relacionados aos *code smells* de Fowler e Beck, e o uso desses princípios pode ajudar a evitar ou mitigar sinais de alerta no desenvolvimento de software.

- **Simplicidade**

Segundo Goodliffe a **simplicidade** ao escrever um código é essencial e consiste em evitar complexidade sem a devida necessidade. Um código simples é fácil de ler, manter e testar, enquanto um código complexo pode gerar diversos *code smells*, como o de **generalização especulativa e comentários**.

A generalização especulativa consiste em escrever código para possíveis situações de uso futuras e esse código pode nunca ser necessário. Isso acarreta em trechos de códigos inutilizados, tornando o projeto mais complexo e difícil de manter.

Já o uso de comentários é um indicador que o código não é intuitivo, sendo utilizado para mascarar a complexidade do que foi desenvolvido. O código deve ser autoexplicativo e a presença de comentários deixa claro que esse não é o caso.

- **Elegância**

Outro princípio descrito por Goodliffe para um código bem desenvolvido é a **elegância**. Um código elegante não é barroco, confuso e nem "esperto". Goodliffe reforça que um código bem projetado tem uma beleza em sua estrutura e anda de mãos dadas com a simplicidade. Um código elegante não deve ter **longos métodos, grandes classes, longa lista de parâmetros e dados aglomerados**.

Um método não deve conter muitas linhas de código, e geralmente em sua versão inicial não tem mesmo, porém é muito comum adicionar linhas sem nunca apagar nada, gerando no final um método gigante e com inúmeras responsabilidades. O mesmo ocorre com classes, já que é muito comum programadores adicionarem uma nova *feature* em uma classe já existente, aumentando sua complexidade e tornando cada vez mais difícil a leitura e manutenção.

Também podemos observar claramente a falta de elegância quando temos uma enorme lista de parâmetros de um método (mais de 4 já é considerado um tamanho grande). Isso torna difícil o entendimento, ficando cada vez mais difícil o seu uso a cada novo parâmetro inserido.

Por fim, é bastante comum ver os mesmos dados aglomerados em diferentes partes do código. Quando isso acontece, tais grupos de dados devem ser transformados em uma classe própria, trazendo mais clareza e elegância ao código desenvolvido.

- **Modularidade**

A **modularidade** consiste em dividir um problema em partes independentes e reutilizáveis, em que cada parte deve ser menos complexa que o problema original, mas ao juntar tudo é formada uma solução completa. Um código bem modularizado evita o surgimento de **grandes classes**, **mudanças divergentes** e "**cirurgia com rifle**".

Modularizar o código evita grandes classes ao dividir funcionalidades em módulos menores e mais gerenciáveis. Já as mudanças divergentes ocorrem quando uma classe é constantemente alterada em diferentes modos por diferentes motivos. Quando é necessário fazer uma alteração no código o desejável é que o programador seja capaz de acessar um ponto específico no sistema e facilmente fazer a mudança e quando isso não é possível é perceptível um *code smell* no projeto. Com um código modular é reduzida a necessidade de alterar uma única classe por diversas razões distintas.

A cirurgia com rifle é quando ocorre o contrário do que ocorre com as mudanças divergentes. Esse *code smell* é perceptível quando toda vez que uma mudança é necessária, é preciso fazer várias pequenas mudanças em diferentes classes. Quando isso ocorre, os locais em que as mudanças devem ocorrer podem ser difíceis de encontrar. Com a modularização, alterações em um módulo específico não irão se espalhar por várias partes do código.

- **Boas interfaces**

O princípio de **boas interfaces** se refere à existência de pontos claros de interação, com uso intuitivo entre diferentes sistemas ou diferentes partes de um mesmo sistema. A interface, conforme explicado por Goodliffe, pode ser entendida como uma fronteira através da qual diferentes módulos, classes e componentes interagem. A boa construção de interfaces evita a existência de **longos métodos**, **longa lista de parâmetros**, "**Inveja de recursos**" e **correntes de mensagens**.

Por incentivar a criação de métodos pequenos e específicos, a construção de boas interfaces previne a existência de longos métodos. E por geralmente reduzir a complexidade dos métodos, fica cada vez mais raro a existência de listas longas de parâmetros.

A inveja de recursos ocorre quando uma função se comunica mais com funções e dados de outro módulo do que com o seu próprio. E isso costuma não acontecer com métodos que seguem boas práticas de interface.

A cadeia de objetos pode ser observada quando temos um cliente solicitando um objeto para outro objeto, cujo cliente pede para outro objeto, e assim segue. Ao construir uma boa interface, tais cadeias são evitadas já que a responsabilidade de uma ação deve estar clara e encapsulada.

- **Extensibilidade**

Apesar de um bom projeto permitir a inserção de novas funcionalidades quando necessário, é importante não superdimensionar o código. Goodliffe comenta em seu livro que quando você desenvolve o projeto pensando em **extensibilidade** você terá escrito no final um sistema operacional, e não um programa. É preciso ter equilíbrio com as funcionalidades necessárias, as que com certeza serão necessárias no futuro e as que talvez sejam necessárias, a fim de evitar a **generalização especulativa**. Os princípios de extensibilidade ajudam a evitar tais generalizações por focar em extensões reais e essenciais.

- **Evitar duplicação**

Goodliffe afirma que **código duplicado** é o inimigo de um projeto elegante e simples. Um programa bem escrito não deve possuir duplicatas, nunca deve haver a necessidade de repetição. Esse princípio dialoga diretamente com o primeiro *code smell* de Fowler e Beck, o de **código duplicado**. Caso seja possível identificar a mesma estrutura de código em mais de um lugar do projeto, com certeza o projeto pode ser aperfeiçoado ao unificá-las em uma única estrutura.

- **Portabilidade**

A **portabilidade** se refere à facilidade que um projeto tem em ser transferido e executado em diferentes ambientes, sistemas operacionais ou plataformas, com o mínimo de alterações. Para Goodliffe um bom design não precisa necessariamente ter portabilidade, isso deve ser definido de acordo com os requisitos do projeto. Apesar de não ser recomendado comprometer o código com portabilidade desnecessária, é importante prevenir dependência de plataforma no projeto. Um projeto bem feito deve ser portátil quando é apropriado e deve gerenciar as questões referentes a tal portabilidade quando elas geram algum tipo de problema. Para evitar possíveis dificuldades na portabilidade, devem ser evitados a existência de **longos métodos** e **longas classes**, já que métodos mais curtos e claros são mais portáteis, assim como as classes menores são mais fáceis de portar entre diferentes contextos.

É bastante comum para métodos extensos possuírem lógicas específicas de uma plataforma ou um ambiente específico, o que traz dificuldade ao processo de portabilidade. Da mesma forma que uma classe grande pode ter comportamentos específicos de uma plataforma, gerando o mesmo problema citado com os métodos.

Apesar de ter dito que não é sempre que a portabilidade é necessária, Goodliffe afirma que a melhor forma de lidar com isso é gerenciando a portabilidade do código em seu design, em vez de improvisá-la como uma reflexão tardia.

- **Código idiomático e bem documentado**

Um bom design deve naturalmente ser construído usando boas práticas, tanto na metodologia do design quanto na implementação de **expressões idiomáticas da linguagem**. Isso é fundamental para que o código fique intuitivo e que outros programadores imediatamente entendam a estrutura do projeto. Além de bem escrito, um bom projeto deve

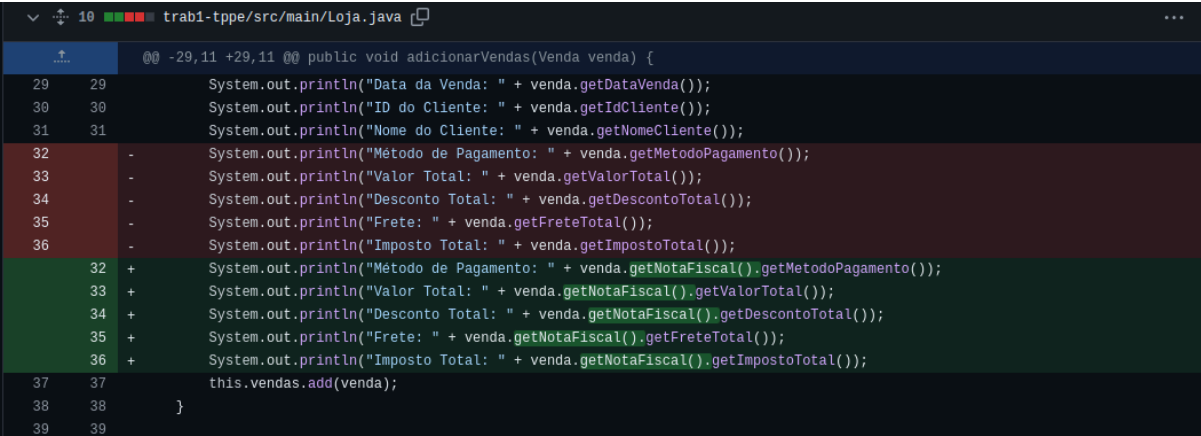
ser **documentado**. Tal documentação deve ser bem feita, mas não extensa, já que o ideal é que o código seja simples o suficiente para ser autoexplicativo. Devemos documentar tomadas de decisões, designs arquiteturais e para fornecer contextos adicionais. Isso evita a presença de **comentários** ao longo do código, já que o mesmo será simples, bem escrito seguindo as convenções da linguagem e com a documentação necessária.

2 - Identificação de maus-cheiros

Após estudar o livro de Martin Fowler, o grupo conseguiu rever o código produzido nos trabalhos 1 e 2 e identificar maus-cheiros que ainda persistem no código. Abaixo está a listagem desses maus-cheiros junto com a explicação e a sugestão de técnicas de refatoração que poderiam resolvê-los:

2.1 - “Cirurgia com Rifle”

Este problema ocorre quando é feita uma mudança em alguma classes e então é necessário aplicar várias pequenas mudanças em várias outras classes. Ao realizar a refatoração de Extrair Classe na classe Venda, foi necessário fazer várias pequenas mudanças no método adicionarVendas da classe Loja (Figura 1). Isto viola o princípio de **Modularidade** e uma sugestão para resolver isso seria aplicando **Mover Método** junto com **Mover Campo**.



```
trabi-tppe/src/main/Loja.java
@@ -29,11 +29,11 @@ public void adicionarVendas(Venda venda) {
29 29      System.out.println("Data da Venda: " + venda.getDataVenda());
30 30      System.out.println("ID do Cliente: " + venda.getIdCliente());
31 31      System.out.println("Nome do Cliente: " + venda.getNomeCliente());
32 -      System.out.println("Método de Pagamento: " + venda.getMetodoPagamento());
33 -      System.out.println("Valor Total: " + venda.getValorTotal());
34 -      System.out.println("Desconto Total: " + venda.getDescontoTotal());
35 -      System.out.println("Frete: " + venda.getFreteTotal());
36 -      System.out.println("Imposto Total: " + venda.getImpostoTotal());
32 +      System.out.println("Método de Pagamento: " + venda.getNotaFiscal().getMetodoPagamento());
33 +      System.out.println("Valor Total: " + venda.getNotaFiscal().getValorTotal());
34 +      System.out.println("Desconto Total: " + venda.getNotaFiscal().getDescontoTotal());
35 +      System.out.println("Frete: " + venda.getNotaFiscal().getFreteTotal());
36 +      System.out.println("Imposto Total: " + venda.getNotaFiscal().getImpostoTotal());
37 37      this.vendas.add(venda);
38 38
39 39  }
```

Figura 1: Commit da refatoração de extrair classe da classe Venda

2.2 - “Inveja de Recursos”

Este mau cheiro ocorre quando um método parece estar “mais interessado” em uma classe em que ele não está, resultando na utilização de vários gets e sets de uma outra classe. Um exemplo disso pode ser encontrado no método adicionarVendas presente na classe Loja (Figura 2). Este método faz duas coisas: a adição da venda no histórico da loja e a impressão da nota fiscal. Observa-se que a parte de impressão da nota fiscal deveria ser um método específico da classe Venda. Isto viola o princípio de **Boas Interfaces** e pode ser resolvido utilizando a técnica **Mover Método**.

```

public void adicionarVendas(Venda venda) {
    // TO-DO: imprimir nota fiscal na tela
    System.out.println("Itens Vendidos:");
    for (ItemVendido item : venda.getItensVendidos()) {
        System.out.println("  Descrição: " + item.getDescricao());
        System.out.println("  ICMS: " + item.getIcms());
        System.out.println("  Municipal: " + item.getMunicipal());
        System.out.println();
    }
    System.out.println("Data da Venda: " + venda.getDataVenda());
    System.out.println("ID do Cliente: " + venda.getIdCliente());
    System.out.println("Nome do Cliente: " + venda.getNomeCliente());
    System.out.println("Método de Pagamento: " + venda.getNotaFiscal().getMetodoPagamento());
    System.out.println("Valor Total: " + venda.getNotaFiscal().getValorTotal());
    System.out.println("Desconto Total: " + venda.getNotaFiscal().getDescontoTotal());
    System.out.println("Frete: " + venda.getNotaFiscal().getFreteTotal());
    System.out.println("Imposto Total: " + venda.getNotaFiscal().getImpostoTotal());
    this.vendas.add(venda);
}

```

Figura 2: Método adicionarVendas da classe Loja

2.3 - Obsessão Primitiva

Este mau-cheiro ocorre quando vários atributos utilizam somente tipos primitivos do próprio Java quando poderia ser usados pequenos objetos para fazer a tipagem de forma mais eficiente e legível. Por exemplo, várias classes deste projeto possuem atributos relacionados à dinheiro (como “preço” na classe Produto; “valorTotal”, “descontoTotal”, “freteTotal” e “impostoTotal” na classe NotaFiscal) e todos usam o tipo primitivo double. Em produto, o método setPreco garante que o número seja um decimal de duas casas (Figura 3) e isto poderia ser transformado em um pequeno objeto que poderia ser utilizado em todos os outros atributos. Este problema viola o princípio de **Modularidade** e poderia ser resolvido com a técnica **Substituir Valor de Dado por Objeto**.

```

public double setPreco(double preco) {
    BigDecimal precoBD = new BigDecimal(Double.toString(preco));
    precoBD = precoBD.setScale(2, RoundingMode.HALF_UP);
    return precoBD.doubleValue();
}

```

Figura 3: método setPreco da classe Produto

2.4 - Lista Longa de parâmetros

Uma lista longa de parâmetros prejudica a legibilidade do código e deixa o método mais difícil de entender, violando os princípios de **Elegância** e **Boas Interfaces**. Na classe Venda, o construtor possui uma lista muito longa de parâmetros que poderia ser **substituído por um objeto** (Figura 4).

```

public Venda(Date dataVenda, int idCliente, String nomeCliente, ItemVendido[] itensVendidos, String metodoPagamento, double valorTotal,
double descontoTotal, double freteTotal, double impostoTotal) {
    this.dataVenda = dataVenda;
    this.idCliente = idCliente;
    this.nomeCliente = nomeCliente;
    this.itensVendidos = itensVendidos;
    this.notaFiscal = new NotaFiscal(metodoPagamento, valorTotal, descontoTotal, freteTotal, impostoTotal);
}

```

Figura 4: Construtor da classe Venda

2.5 - Código Duplicado

O método `setPreco` está sendo chamado no construtor, o que pode causar confusão e viola os princípios da Simplicidade e Elegância, já que o método retorna o valor ajustado, mas a lógica de arredondamento poderia ser movida diretamente para dentro do construtor.

```
public Produto(int id, String descricao, double preco, String medida) {  
    this.id = id;  
    this.descricao = descricao;  
    this.preco = setPreco(preco);  
    this.medida = medida;  
}
```

Figura 5: Construtor da classe Produto

2.6 - Data Class

A classe `Produto` age principalmente como um contêiner de dados, sem muita lógica associada, o que é característico do *code smell* **Data Class**. Isso pode indicar uma falta de encapsulamento adequado e levar a problemas de manutenção e evolução do código. O princípio do **Encapsulamento** acaba sendo violado.

```
public void setId(int id) {  
    this.id = id;  
}  
  
public String getDescricao() {  
    return descricao;  
}  
  
public void setDescricao(String descricao) {  
    this.descricao = descricao;  
}
```

Figura 6: Getters e Setters da classe Produto

Referências

Martin Fowler. Refactoring: Improving the design of Existing Code. Addison-Wesley Professional, 1999.

Pete Goodliffe. Code Craft: The practice of Writing Excellent Code. No Starch Press, 2006.