



Universidade de Brasília (UnB)
Faculdade do Gama
Engenharia de Software

Princípios de um bom projeto de código

Arthur de Melo Viana - 211029147
Douglas Alves dos Santos - 211029620
Geovanna Maciel Avelino da Costa - 202016328

Brasília
2024

1 INTRODUÇÃO

Refatoração é o processo de modificar o código de um sistema sem alterar seu comportamento externo, mas melhorando sua estrutura interna. Embora tradicionalmente se acredite que o design deva preceder a codificação, a realidade é que, com o tempo, o código pode perder a integridade do design original e tornar-se desordenado. A refatoração pode combater essa tendência transformando código mal projetado em um sistema bem estruturado por meio de pequenas alterações cumulativas. Dessa forma, os projetos não possuem mais uma única etapa inicial, mas passam a ocorrer continuamente durante o desenvolvimento, resultando em um sistema que se mantém robusto ao longo do tempo.

A refatoração feita de forma inapropriada pode levar uma equipe de software a atrasos de dias, até mesmo semanas, tornando-a ainda mais arriscada quando praticada informalmente, sem seguir uma metodologia aceita. Para evitar esses problemas, a refatoração deve ser feita sistematicamente para a criação de um código de alta qualidade.

2 PRINCÍPIOS DE UM BOM PROJETO DE CÓDIGO E MAUS-CHEIROS RELACIONADOS

A criação de um código de alta qualidade é um dos pilares fundamentais do desenvolvimento de software bem-sucedido. Para alcançar esse objetivo, é essencial seguir princípios de bom projeto de código, como simplicidade, elegância, modularidade, boas interfaces, extensibilidade, evitar duplicação, portabilidade, e garantir que o código seja idiomático e bem documentado.

Esses princípios, quando aplicados corretamente, não apenas tornam o código mais fácil de escrever, entender e manter, mas também o tornam mais resiliente às mudanças. No entanto, falhas no cumprimento desses princípios podem levar ao surgimento dos chamados "maus-cheiros de código", conceitos identificados por Martin Fowler, que indicam áreas do software que necessitam de refatoração. A compreensão e a aplicação desses princípios são, portanto, fundamentais para manter a integridade e a eficácia de um projeto de software ao longo de seu ciclo de vida.

- Simplicidade

A simplicidade em um projeto de código se refere à criação de soluções diretas e sem complicações desnecessárias. O código simples é fácil de entender, manter e modificar, evitando a introdução de complexidade que não contribui para o objetivo do software. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Código Duplicado:** A duplicação é inimiga da simplicidade, pois cria redundâncias que tornam o código mais complexo e difícil de manter. A refatoração para remover duplicações simplifica o código.
- **Classe Ociosa:** Elementos de código que não são utilizados aumentam a complexidade sem necessidade. A remoção de código morto torna o código mais simples e limpo.

- Elegância

A elegância no código é a qualidade que faz com que o software seja não apenas funcional, mas também agradável de ler e trabalhar. Isso envolve o uso de padrões de design bem estabelecidos e a aplicação de soluções que são tanto eficientes quanto esteticamente organizadas. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Métodos Longos:** Métodos que são longos demais geralmente carecem de elegância, pois contêm muita lógica em um só lugar. Refatorar métodos longos em métodos menores e mais focados pode trazer de volta a elegância.
- **Intimidade Inadequada:** Uma classe que faz tudo compromete a elegância do código. O código elegante distribui responsabilidades de forma apropriada entre classes.

- Modularidade

Modularidade é a divisão do código em partes menores e independentes, chamadas módulos, cada uma responsável por uma parte específica da funcionalidade do sistema. Isso facilita a manutenção, a escalabilidade e o teste do software. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Inveja dos Dados:** Quando um método numa classe acessa mais dados de outra classe do que da sua própria, indica uma falha na modularidade. Refatorar para respeitar a modularidade ajuda a distribuir melhor as responsabilidades.
- **Classes de Dados:** Classes que apenas armazenam dados sem comportamento associado indicam uma falta de modularidade. Refatorar para atribuir responsabilidades adequadas a essas classes melhora a modularidade.

- Boas Interfaces

Boas interfaces fornecem meios claros e concisos para que as diferentes partes do sistema se comuniquem. Elas expõem apenas o necessário e ocultam a complexidade interna, facilitando o uso e a integração de componentes. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Classes Alternativas com Interfaces Diferentes:** Nomes que não refletem a verdadeira intenção de uma classe ou método comprometem a clareza das interfaces. Corrigir nomes enganosos ajuda a criar interfaces mais intuitivas e claras.

- Extensibilidade

Extensibilidade refere-se à capacidade do código de ser facilmente ampliado para adicionar novas funcionalidades sem a necessidade de grandes alterações na base existente. Um sistema

extensível é projetado com a flexibilidade em mente. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Inveja dos Dados:** Métodos que dependem excessivamente de variáveis globais ou de contexto externo comprometem a extensibilidade. Refatorar para reduzir essas dependências melhora a capacidade de extensão do código.
- **Generalidade Especulativa:** Tentar prever necessidades futuras e adicionar código para isso pode comprometer a extensibilidade. É melhor refatorar para suportar apenas as funcionalidades atuais e evoluir conforme necessário.

- Evitar Duplicação

Evitar duplicação significa eliminar a repetição de lógica e dados no código. Quando algo é duplicado, ele aumenta a complexidade do sistema e o risco de inconsistências. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Código Duplicado:** É um dos maus-cheiros mais diretos relacionados à duplicação. A refatoração para eliminar código duplicado, consolidando lógica em métodos ou classes compartilhadas, melhora a coesão e reduz erros.
- **Cirurgia com Rifle:** Quando uma única alteração exige a modificação de muitas partes diferentes do sistema, isso indica duplicação de lógica. Refatorar para centralizar a lógica pode eliminar essa necessidade de alterações múltiplas.

- Portabilidade

Portabilidade é a facilidade com que o código pode ser executado em diferentes ambientes, plataformas ou configurações. Um código portátil é adaptável e não depende de recursos específicos de um ambiente. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Intermediário:** Dependências excessivas em APIs ou bibliotecas específicas de uma plataforma podem comprometer a portabilidade. Refatorar para reduzir essas dependências melhora a capacidade de mover o código entre diferentes ambientes.

- Código Idiomático e Bem Documentado

Um código idiomático segue as convenções e padrões de sua linguagem de programação, tornando-o mais compreensível para outros desenvolvedores. Além disso, a documentação clara ajuda na manutenção e evolução do código. Agora relacionando com os maus-cheiros de código apresentados por Fowler temos:

- **Excesso de Comentários:** Comentários excessivos podem indicar que o código é complicado demais e difícil de entender. Refatorar para melhorar a clareza e reduzir a necessidade de comentários melhora a qualidade do código.

3 MAUS-CHEIROS PERSISTENTES NO TRABALHO PRÁTICO 2

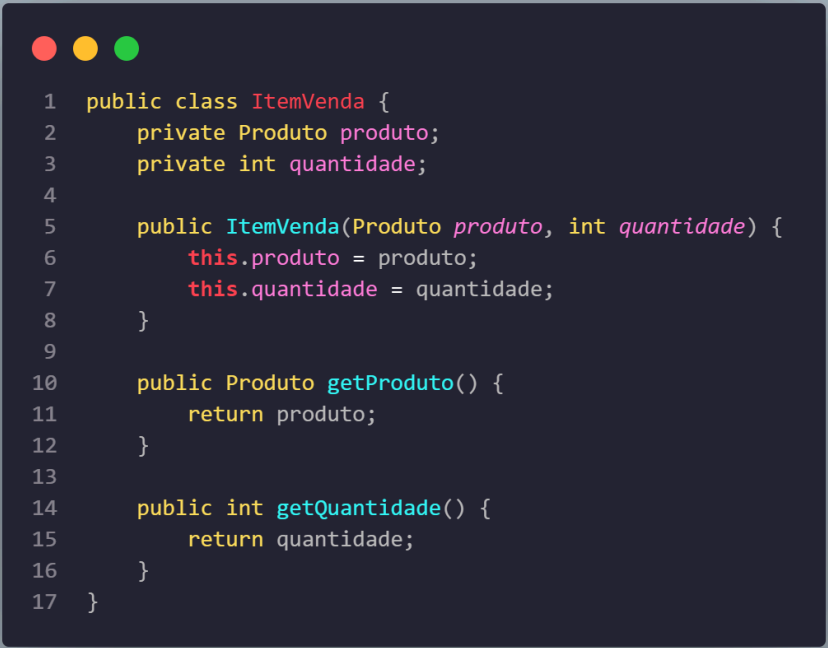
Após o estudo dos bons princípios de código, foram identificados os maus-cheiros citados por Fowler no trabalho prático que se seguem:

→ **Código Duplicado:**

- ◆ **Princípio violado:** Evitar duplicação.
- ◆ **Descrição:** A classe Item e a classe ItemVenda servem a propósitos semelhantes. Ambas armazenam informações sobre itens, incluindo o nome/produto, quantidade e preço. A existência dessas classes duplicadas pode causar inconsistências e dificulta a manutenção.
- ◆ **Operações de Refatoração:** **Substituir Classe por Campo** ou **Fundir Classes**. É possível unificar as duas classes, eliminando a redundância.

```
1  public class Item {
2      private String nome;
3      private int quantidade;
4      private double preco;
5
6      public Item(String nome, int quantidade, double preco) {
7          this.nome = nome;
8          this.quantidade = quantidade;
9          this.preco = preco;
10     }
11
12     public String getNome() {
13         return nome;
14     }
15
16     public int getQuantidade() {
17         return quantidade;
18     }
19
20     public double getPreco() {
21         return preco;
22     }
23
24     public double getTotal() {
25         return quantidade * preco;
26     }
27 }
```

Fonte: Autores



```
1  public class ItemVenda {
2      private Produto produto;
3      private int quantidade;
4
5      public ItemVenda(Produto produto, int quantidade) {
6          this.produto = produto;
7          this.quantidade = quantidade;
8      }
9
10     public Produto getProduto() {
11         return produto;
12     }
13
14     public int getQuantidade() {
15         return quantidade;
16     }
17 }
```

Fonte: Autores

→ **Método Longo (calcularValores):**

- ◆ **Princípio violado:** Simplicidade e responsabilidade única.
- ◆ **Descrição:** O método calcularValores() na classe Venda ainda realiza múltiplas tarefas, como calcular valores de itens, frete, descontos, e impostos, além de atualizar o estado do cliente.
- ◆ **Operações de Refatoração: Extrair Método.** Parte do código já foi extraído para métodos como processarItemVenda, mas outras partes ainda podem ser separadas para aumentar a clareza e facilitar a manutenção.

```
1  private void calcularValores() {
2      valorTotal = 0;
3      valorDesconto = 0;
4      double valorImposto = 0;
5
6      cliente.calcularComprasUltimoMes();
7
8      // Extraí o cálculo do valor de cada item
9      for (ItemVenda item : itens) {
10         processarItemVenda(item);
11     }
12
13     valorFrete = calcularFrete(cliente);
14     valorTotal += valorFrete + valorImposto;
15
16     if (cliente.getTipo().equals("Prime")) {
17         aplicarCashback(metodoPagamento);
18         aplicarDescontoCashback();
19     }
20
21     cliente.adicionarVenda(this);
22 }
```

Fonte: Autores

→ **Complexidade Condicional:**

- ◆ **Princípio violado:** Simplicidade e elegância.
- ◆ **Descrição:** As estruturas condicionais na função `calcularFrete` e em outros métodos (como `aplicarDescontoCashback`) tornam o código mais complexo e difícil de entender.
- ◆ **Operações de Refatoração:** Substituir Condicional por Polimorfismo ou Substituir Condicional por Strategy Pattern. Essas refatorações podem isolar as diferentes regras de cálculo em classes ou estratégias separadas.

```
1  private double calcularFrete(Cliente cliente) {
2      double frete;
3      switch (cliente.getEstado()) {
4          case "DF":
5              frete = 5.00;
6              break;
7          case "Centro-oeste":
8              frete = cliente.isCapital() ? 10.00 : 13.00;
9              break;
10         case "Nordeste":
11             frete = cliente.isCapital() ? 15.00 : 18.00;
12             break;
13         case "Norte":
14             frete = cliente.isCapital() ? 20.00 : 25.00;
15             break;
16         case "Sudeste":
17             frete = cliente.isCapital() ? 7.00 : 10.00;
18             break;
19         case "Sul":
20             frete = cliente.isCapital() ? 10.00 : 13.00;
21             break;
22         default:
23             frete = 0;
24     }
25     if (cliente.getTipo().equals("Prime")) {
26         return 0;
27     } else if (cliente.getTipo().equals("Especial")) {
28         return frete * 0.70;
29     }
30     return frete;
31 }
```

Fonte: Autores

→ **Mudança Divergente:**


- ◆ **Princípio violado:** Responsabilidade única e modularidade.
- ◆ **Descrição:** A classe Venda sofre de mudanças divergentes, onde várias alterações relacionadas a diferentes aspectos do negócio (cálculo de valores, frete, descontos, e estado do cliente) precisam ser feitas em uma única classe.
- ◆ **Operações de Refatoração: Extrair Classe.** Parte do código já foi movida para a classe CalculadoraImpostos, mas outras partes poderiam ser movidas para novas classes, como CalculadoraFrete, CalculadoraDescontos, etc.

```
1 public Venda(Date data, Cliente cliente, List<ItemVenda> itens, String metodoPagamento, boolean usarCashback) {
2     this.data = data;
3     this.cliente = cliente;
4     this.itens = itens;
5     this.metodoPagamento = metodoPagamento;
6     this.impostosItens = new LinkedList<double[]>();
7     this.usarCashback = usarCashback;
8     calcularValores();
9 }
10
11 private void calcularValores() {
12     // Detalhes de implementação ...
13 }
14
15 // Novo método para processar cada item da venda
16 private void processarItemVenda(ItemVenda item) {
17     // Detalhes de implementação ...
18 }
19
20 // Novo método para calcular o desconto de um item
21 private double calcularDescontoItem(Cliente cliente, String metodoPagamento) {
22     // Detalhes de implementação ...
23 }
24
25 // Novo método para aplicar descontos e impostos ao valor de um item
26 private double aplicarDescontoImpostos(double valorItem, double descontoItem, double icms, double impostoMunicipal) {
27     // Detalhes de implementação ...
28 }
29
30 // Método para aplicar o desconto do cashback no valor total
31 private void aplicarDescontoCashback() {
32     // Detalhes de implementação ...
33 }
34
35 private double calcularFrete(Cliente cliente) {
36     // Detalhes de implementação ...
37 }
38
39 private void aplicarCashback(String metodoPagamento) {
40     // Detalhes de implementação ...
41 }
```

Fonte: Autores


→ **Dependência Direta em Strings:**

- ◆ **Princípio violado:** Código idiomático e extensibilidade.
- ◆ **Descrição:** Comparações de strings como `cliente.getTipo().equals("Prime")` e `cliente.getEstado().equals("DF")` são propensas a erros de digitação e dificultam a extensão do código.
- ◆ **Operações de Refatoração: Substituir Constantes Mágicas por Enumerações.** Isso tornaria o código mais claro e reduziria o risco de erros.



```
1  if (cliente.getTipo().equals("Prime")) {
2      aplicarCashback(metodoPagamento);
3      aplicarDescontoCashback();
4  }
```

Fonte: Autores



```
1  if (cliente.getEstado().equals("DF")) {
2      this.icms = 0.18;
3      this.impostoMunicipal = 0.00;
4  } else {
5      this.icms = 0.12;
6      this.impostoMunicipal = 0.04;
7  }
```

Fonte: Autores

4 CONSIDERAÇÕES FINAIS

A refatoração é uma prática essencial no desenvolvimento de software moderno, garantindo que o código se mantenha limpo, eficiente e fácil de manter à medida que o sistema evolui. Como discutido, seguir os princípios de um bom projeto de código e identificar os maus-cheiros relacionados é fundamental para criar e manter um sistema robusto e de alta qualidade.

As operações de refatoração discutidas, como a substituição de classes duplicadas, a extração de métodos e classes, e a aplicação de padrões de design como Strategy, são exemplos de como essas melhorias podem ser aplicadas de forma prática para resolver problemas específicos de código. Além disso, elas não devem ser vistas como uma etapa isolada, mas como uma parte contínua do ciclo de desenvolvimento. Ao refatorar constantemente e em pequenas etapas, o código permanece saudável e preparado para mudanças futuras, reduzindo o risco de problemas que possam impactar o cronograma ou a qualidade final do software.

Finalmente, é crucial lembrar que um projeto de software bem-sucedido exige não apenas habilidade técnica, mas também uma boa compreensão dos princípios de design e uma visão clara dos objetivos do sistema. Dito isso, com uma abordagem cuidadosa e disciplinada, a refatoração contribui significativamente para o sucesso a longo prazo do projeto de software, garantindo que ele permaneça funcional, adaptável e sustentável à medida que cresce e se adapta a novas necessidades.

4 REFERÊNCIAS BIBLIOGRÁFICAS

- 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.