ATA 1 - Code Smells e Refatorações

Grupo 2:

- Lucas de Oliveira Rodrigues Alves
- Matheus Yuiti Moriy Miata
- Vitor Milanez

Índice

Referências

- https://refactoring.guru/pt-br/refactoring/smells
- https://coodesh.com/blog/dicionario/o-que-e-code-smell/
- https://refactoring.guru/pt-br/refactoring/techniques
- https://refactoring.com/
- https://engsoftmoderna.info/cap9.html

Code Smells

Code Smells

O que é?

- Smell = cheiro
- Algo não “cheira bem” no seu código.
- Qualquer característica no código-fonte de um programa que possivelmente indica um problema mais profundo.
- Metáfora indica bem o problema -> um cheiro é fácil de detectar, principalmente, para bons farejadores
- Code Smells não são bugs, contudo, eles indicam pontos fracos que podem ocasionar falhas no presente ou no futuro.

Code Smells

Como corrigir?

- Melhor é prevenir. Por isso, deve-se ser providenciada a correção do código-fonte quando um Code Smell identifica que algo não vai bem.
- O problema mais profundo pode ser descoberto por meio de ciclos curtos de feedback -> código é refatorado em etapas menores e controladas.
- Observa-se o design para examinar se há novos Code Smells espalhados. Isso, portanto, indica a necessidade de refatoração.

Tipos de Code Smells

Bloaters

Bloaters são códigos, métodos e classes, que cresceram para proporções tão grandes que fazem ser difíceis de serem utilizadas.

Normalmente, não aparecem imediatamente, se acumulando ao longo do tempo com a evolução do programa.

- **Métodos Longo**
- **Classes Grande**
- **Obsessão Primitiva**
- **Lista Longa de Parâmetros**
- **Aglomerados de Dados**

Exemplo

Método Longo

```java
public class OrderProcessor {
    public void processOrder(Order order) {
        // 1. Validação do pedido
        if (order == null) {
            throw new IllegalArgumentException(s:"Pedido não pode ser nulo");
        }
        if (order.getItems() == null || order.getItems().isEmpty()) {
            throw new IllegalArgumentException(s:"Pedido sem itens");
        }
        if (order.getCustomer() == null) {
            throw new IllegalArgumentException(s:"Cliente não associado");
        }

        // 2. Cálculo do total
        double total = 0;
        for (Item item : order.getItems()) {
            if (item.getPrice() <= 0) {
                throw new IllegalArgumentException(s:"Preço do item inválido");
            }
            total += item.getPrice() * item.getQuantity();
        }
        if (total > 1000) {
            total *= 0.9; // Aplica 10% de desconto
        }

        // 3. Geração de log
        System.out.println("Processando pedido #" + order.getId());
        System.out.println("Cliente: " + order.getCustomer().getName());
        System.out.println("Total: R$" + total);

        // 4. Atualização de estoque
        for (Item item : order.getItems()) {
            Inventory inventory = InventoryService.lookup(item.getProductId());
            inventory.reduceStock(item.getQuantity());
            System.out.println("Atualizado estoque: " + item.getProductId());
        }

        // 5. Notificação
        EmailService.send(
            order.getCustomer().getEmail(),
            "Seu pedido #" + order.getId() + " foi processado",
            "Total: R$" + total
        );
    }
}
```

Exemplo

Obsessão Primitiva

```
public class Order {
    private String deliveryAddress;

    public void setDeliveryAddress(String deliveryAddress) {
        this.deliveryAddress = deliveryAddress;
    }

    public String getDeliveryAddress() {
        return deliveryAddress;
    }
}
```

Object-Orientation Abusers

Esses smells surgem quando os princípios fundamentais da Orientação a Objetos (como polimorfismo, encapsulamento e abstração) são ignorados ou aplicados incorretamente.

- **Classes Alternativas com Interfaces Diferentes**
- **Herança Recusada**
- **Instruções Switch**
- **Campo Temporário**

Exemplo

Instruções Switch

```
public class Payment {
    public void process(String paymentType) {
        switch (paymentType) {
            case "CREDIT_CARD": /* lógica */ break;
            case "PAYPAL": /* lógica */ break;
            // + novos tipos exigem modificar esta classe!
        }
    }
}
```

Exemplo

Classes Alternativas com Interfaces Diferentes

```java
public class PixProcessor {
    public void processPayment() {
        // lógica
    }

    public void processRefund() {
        // lógica
    }
}
```

```java
public class CreditCardProcessor {
    public void processPayment() {
        // lógica
    }

    public void processRefund() {
        // lógica
    }
}
```

Change Preventers

Esses smells estão relacionados com casos em que, ao alterar algo em uma parte do código, muitas mudanças em outras partes devem ser feitas também.

- **Mudança Divergente**
- **Cirurgia com Espingarda**
- **Hierarquias de Herança Paralela**

Exemplo

Mudança Divergente

```java
public class RelatorioFinanceiro {
    public String gerarRelatorioHTML(List<Transacao> transacoes) {
        // lógica...
    }

    public double calcularTotal(List<Transacao> transacoes) {
        // lógica...
    }

    public void salvarRelatorioEmArquivo(String relatorio, String caminhoArquivo) {
        // lógica...
    }
}
```

Dispensables

Um dispensable é algo desnecessário, cuja a ausência tornaria o código mais limpo, eficiente e fácil de entender

- **Comentários**
- **Código Duplicado**
- **Classes de Dados**
- **Classes Preguiçosas**
- **Código Morto**
- **Generalidade Especulativa**

Exemplo

Código Duplicado

```java
// Classe 1: Processador de Pedidos
public class OrderProcessor {
    public void processOrder(Order order) {
        // Validação de e-mail
        if (!order.getCustomerEmail().matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$")) {
            throw new IllegalArgumentException("E-mail do cliente inválido");
        }
        // Lógica de processamento...
    }
}

// Classe 2: Gerador de Fatura
public class InvoiceGenerator {
    public void generateInvoice(Order order) {
        // Validação de e-mail
        if (!order.getCustomerEmail().matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$")) {
            throw new IllegalArgumentException(s:"E-mail do cliente inválido");
        }
        // Lógica de geração da fatura...
    }
}
```

Couplers

Trata-se do alto acoplamento -> a forma de medir o quão dependente uma classe é das outras.

- **Inveja de Funcionalidade**
- **Intimidade Inapropriada**
- **Classes de Biblioteca Incompleta**
- **Cadeias de Mensagens**
- **Intermediário**

Exemplo

Cadeias de Mensagens

```
public class CheckoutService {
    public void finalizeOrder(Order order) {
        // Cadeia longa de chamadas (acoplamento excessivo)
        String customerCity = order.getCustomer().getAddress().getCity();
        String customerCountry = order.getCustomer().getAddress().getCountry();

        // lógica...
    }
}
```

Refatorações

Refatoração

O que é?

- “Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.” - Martin Fowler.
- São pequenas mudanças que podem levar a uma grande reestruturação e melhora da qualidade do código existente e maior flexibilidade para adição de novas funcionalidades.

Refatoração

Quando usar?

- Nas fases de Implementação, Testes e Manutenção do ciclo de vida do software.
- Em níveis mais baixos, na adição de novas funcionalidades, code reviews, alto número de bugs/dificuldade em corrigi-los.

Técnicas de Refatoração

Composing Methods

Foco em refatorar como os métodos são estruturados e organizados dentro de uma classe, com o objetivo de torná-los mais coesos, legíveis e fáceis de entender.

- **Extrair método**
- **Método inline**
- **Extrair variável**
- **Variável Inline**
- **Substituir variável por método**
- **Dividir variável temporária**
- **Remover atribuições a variáveis**
- **Substituir método por uma classe**
- **Substituir algoritmo**

Exemplo

Método Longo

```java
public class OrderProcessor {
    public void processOrder(Order order) {
        // 1. Validação do pedido
        if (order == null) {
            throw new IllegalArgumentException(s:"Pedido não pode ser nulo");
        }
        if (order.getItems() == null || order.getItems().isEmpty()) {
            throw new IllegalArgumentException(s:"Pedido sem itens");
        }
        if (order.getCustomer() == null) {
            throw new IllegalArgumentException(s:"Cliente não associado");
        }

        // 2. Cálculo do total
        double total = 0;
        for (Item item : order.getItems()) {
            if (item.getPrice() <= 0) {
                throw new IllegalArgumentException(s:"Preço do item inválido");
            }
            total += item.getPrice() * item.getQuantity();
        }
        if (total > 1000) {
            total *= 0.9; // Aplica 10% de desconto
        }

        // 3. Geração de log
        System.out.println("Processando pedido #" + order.getId());
        System.out.println("Cliente: " + order.getCustomer().getName());
        System.out.println("Total: R$" + total);

        // 4. Atualização de estoque
        for (Item item : order.getItems()) {
            Inventory inventory = InventoryService.lookup(item.getProductId());
            inventory.reduceStock(item.getQuantity());
            System.out.println("Atualizado estoque: " + item.getProductId());
        }

        // 5. Notificação
        EmailService.send(
            order.getCustomer().getEmail(),
            "Seu pedido #" + order.getId() + " foi processado",
            "Total: R$" + total
        );
```

Refatoração -> Extração de Métodos

Método Longo

```java
public class OrderProcessor {
    public void processOrder(Order order) {
        validateOrder(order);
        double total = calculateTotal(order);
        logOrderDetails(order, total);
        updateInventory(order);
        notifyCustomer(order, total);
    }

    private void validateOrder(Order order) {
        // lógica...
    }

    private double calculateTotal(Order order) {
        // lógica...
    }

    private void logOrderDetails(Order order, double total) {
        // lógica...
    }

    private void updateInventory(Order order) {
        // lógica...
    }

    private void notifyCustomer(Order order, double total) {
        // lógica...
    }
}
```

Moving Features between Objects

Foco em refatorar a distribuição de responsabilidades entre classes, com o objetivo de garantir que cada classe tenha uma responsabilidade clara e única e que o acoplamento entre classes seja minimizado.

- **Mover método**
- **Mover campo**
- **Extrair classe**
- **Classe inline**
- **Esconder delegações**
- **Remover intermediário**
- **Introduzir método utilitário**
- **Introduzir extensão local**

Exemplo

Mudança Divergente

```java
public class RelatorioFinanceiro {
    public String gerarRelatorioHTML(List<Transacao> transacoes) {
        // lógica...
    }

    public double calcularTotal(List<Transacao> transacoes) {
        // lógica...
    }

    public void salvarRelatorioEmArquivo(String relatorio, String caminhoArquivo) {
        // lógica...
    }
}
```

Refatoração

Mudança Divergente

```java
// Classe 1: Responsável apenas por formatação
public class FormatadorRelatorio {
    public String formatarHTML(List<Transacao> transacoes) {
        // lógica...
    }
}

// Classe 2: Responsável apenas por cálculos
public class CalculadorFinanceiro {
    public double calcularTotal(List<Transacao> transacoes) {
        // lógica...
    }
}

// Classe 3: Responsável apenas por persistência
public class ArmazenadorRelatorio {
    public void salvarEmArquivo(String conteudo, String caminho) {
        // lógica...
    }
}
```

```java
// Classe principal (coordenação)
public class RelatorioFinanceiro {
    private FormatadorRelatorio formatador;
    private CalculadorFinanceiro calculador;
    private ArmazenadorRelatorio armazenador;

    // lógica...
}
```

Exemplo

Código Duplicado

```java
// Classe 1: Processador de Pedidos
public class OrderProcessor {
    public void processOrder(Order order) {
        // Validação de e-mail
        if (!order.getCustomerEmail().matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$")) {
            throw new IllegalArgumentException("E-mail do cliente inválido");
        }
        // Lógica de processamento...
    }
}

// Classe 2: Gerador de Fatura
public class InvoiceGenerator {
    public void generateInvoice(Order order) {
        // Validação de e-mail
        if (!order.getCustomerEmail().matches("^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$")) {
            throw new IllegalArgumentException(s:"E-mail do cliente inválido");
        }
        // Lógica de geração da fatura...
    }
}
```

Refatoração - Extração de classe

Código Duplicado

```java
// Classe utilitária centralizada
public final class EmailValidator {
    private static final String EMAIL_REGEX = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+$";

    public static void validate(String email) {
        if (email == null || !email.matches(EMAIL_REGEX)) {
            throw new IllegalArgumentException(s:"E-mail inválido");
        }
    }
}

// Classes refatoradas (sem duplicação)
public class OrderProcessor {
    public void processOrder(Order order) {
        EmailValidator.validate(order.getCustomerEmail()); // Reuso da validação
        // Lógica de processamento...
    }
}
public class InvoiceGenerator {
    public void generateInvoice(Order order) {
        EmailValidator.validate(order.getCustomerEmail()); // Reuso da validação
        // Lógica de geração da fatura...
    }
}
```

Organizing Data

Foco em refatorar como os dados são estruturados e acessados, com o objetivo de tornar a manipulação dos dados mais seguro, flexível e fácil.

- **Encapsulamento de campo**
- **Substituir valor com objeto**
- **Mudar valor para referência**
- **Mudar referência para valor**
- **Substituir Array com objeto**
- **Duplicar dado observado**
- **Mudar associação unidirecional para bidirecional**

Organizing Data

- Mudar associação bidirecional para unidirecional
- Substituir número mágico por constante
- Encapsular campo
- Encapsular coleção
- Substituir campos relacionados com classe, subclasse ou State/Strategy
- Substituir subclasse com campos

Exemplo

Obsessão Primitiva

```
public class Order {
    private String deliveryAddress;

    public void setDeliveryAddress(String deliveryAddress) {
        this.deliveryAddress = deliveryAddress;
    }

    public String getDeliveryAddress() {
        return deliveryAddress;
    }
}
```

Refatoração - Substituir Campo com Objeto

Obsessão Primitiva

```java
public class Address {
    private String street;
    private String city;
    private String zip;

    // Getters and Setters
}
```

```java
public class Delivery {
    private Address address;

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}
```

Simplifying Conditional Expressions

Foco em refatorar as expressões condicionais complexas, com o objetivo de torná-las mais simples e legíveis e evitar o aumento da complexidade à medida que o sistema evolui.

- **Decompor condicionais**
- **Consolidar expressões condicionais**
- **Consolidar fragmentos de condicionais duplicadas**
- **Remover flags de controle**
- **Substituir condicionais aninhadas com Guard Clauses**
- **Substituir condicionais com polimorfismo**
- **Introduzir objetos null**
- **Introduzir asserções**

Exemplo

Instruções Switch

```
public class Payment {
    public void process(String paymentType) {
        switch (paymentType) {
            case "CREDIT_CARD": /* lógica */ break;
            case "PAYPAL": /* lógica */ break;
            // + novos tipos exigem modificar esta classe!
        }
    }
}
```

Refatoração - Substituição por Polimorfismo

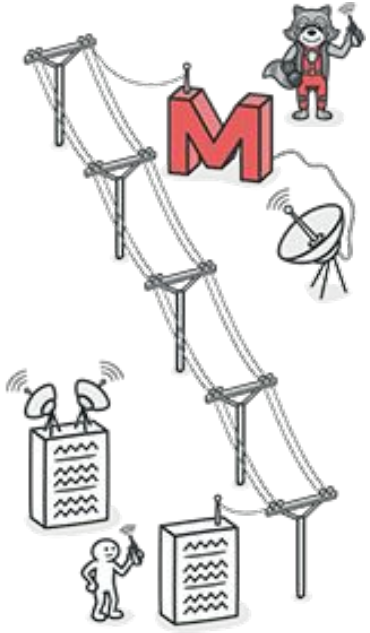
Instruções Switch

```java
// Interface comum
public interface Payment {
    void process();
}

// Implementações concretas
public class CreditCardPayment implements Payment {
    @Override public void process() {
        // lógica...
    }
}

public class PayPalPayment implements Payment {
    @Override public void process() {
        // lógica...
    }
}
```

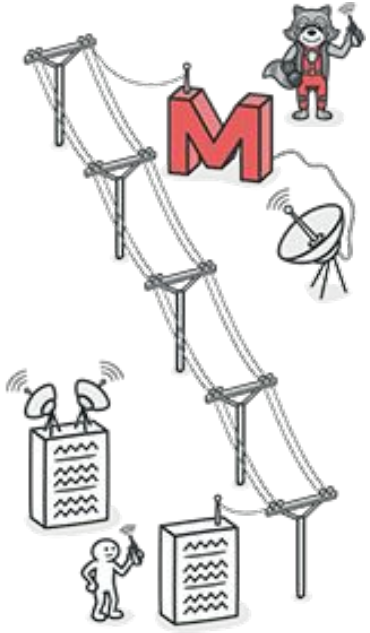
Simplifying Method Calls



Foco em refatorar as chamadas de métodos, com o objetivo de torná-las mais claras, concisas e fáceis de usar. Em troca, simplifica as interfaces para interação entre classes.

- **Renomear método**
- **Adicionar parâmetro**
- **Remover parâmetro**
- **Separar métodos setter e getter**
- **Combinar métodos similares com a adição de parâmetros necessários**
- **Preservar o objeto**

Simplifying Method Calls



- Substituir parâmetro com chamada de método
- Introduzir parâmetro de objeto
- Remover método setter
- Substituir construtor com Factory Method
- Substituir código de erro com exceção
- Substituir exceção com teste

Exemplo

Cadeias de Mensagens

```
public class CheckoutService {
    public void finalizeOrder(Order order) {
        // Cadeia longa de chamadas (acoplamento excessivo)
        String customerCity = order.getCustomer().getAddress().getCity();
        String customerCountry = order.getCustomer().getAddress().getCountry();

        // lógica...
    }
}
```

Refatoração - Esconder Delegações

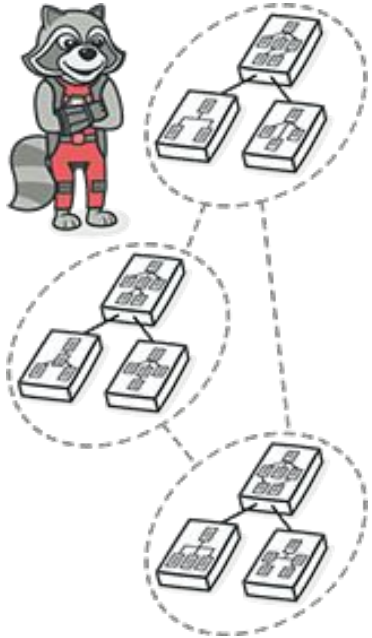
Cadeias de Mensagens

```java
public class Order {
    private Customer customer;

    // Novo método encapsulando a cadeia
    public String getCustomerCity() {
        return customer.getAddress().getCity();
    }
    public String getCustomerCountry() {
        return customer.getAddress().getCountry();
    }
}
```

```java
// Classe refatorada
public class CheckoutService {
    public void finalizeOrder(Order order) {
        // Cadeia reduzida (menos acoplamento)
        String customerCity = order.getCustomerCity();
        String customerCountry = order.getCustomerCountry();
        // lógica
    }
}
```

Dealing with Generalization



Foco em refatorar como as hierarquias de herança são projetadas e utilizadas, com o objetivo de garantir que a herança seja usada de forma apropriada e eficaz, promovendo reutilização e evitando designs rígidos

- **Subir um campo**
- **Subir um método**
- **Subir o corpo do construtor**
- **Descer um campo**
- **Descer um método**
- **Extrair subclasse**
- **Extrair superclasse**
- **Extrair interface**
- **Unir a hierarquia**
- **Formar Template Method**
- **Substituir herança com delegação**
- **Substituir delegação com herança**

Exemplo

Classes Alternativas com Interfaces Diferentes

```java
public class PixProcessor {
    public void processPayment() {
        // lógica
    }

    public void processRefund() {
        // lógica
    }
}
```

```java
public class CreditCardProcessor {
    public void processPayment() {
        // lógica
    }

    public void processRefund() {
        // lógica
    }
}
```

Refatoração - Extrair Interface

Classes Alternativas com Interfaces Diferentes

```java
interface PaymentProcessor {
    void processPayment();

    void processRefund();
}
```

```java
public class CreditCardProcessor implements PaymentProcessor {
    @Override
    public void processPayment() {
        // lógica
    }

    @Override
    public void processRefund() {
        // lógica
    }
}
```

Refatoração - Extrair Interface

Classes Alternativas com Interfaces Diferentes

```java
interface PaymentProcessor {
    void processPayment();

    void processRefund();
}
```

```java
public class PixProcessor implements PaymentProcessor {
    @Override
    public void processPayment() {
        // lógica
    }

    @Override
    public void processRefund() {
        // lógica
    }
}
```

Obrigado!