## Entrega 3 Trabalho TPPE Grupo 19

## Alunos:

**Paulo Victor Fonseca Sousa - Matrícula: 211043718**

**Leonardo Gonçalves Machado - Matrícula: 211029405**
**Arthur Grandão de Mello - Matrícula: 211039250**

# 1. Relação entre os Princípios de Bom Projeto e os Maus-Cheiros de Código

O design de software é um processo contínuo que se estende para além da fase de planejamento, incorporando também a codificação. Conforme destacado por Martin Fowler em *Refactoring: Improving the Design of Existing Code* (1999), refatoração é o aperfeiçoamento do código sem modificar seu comportamento externo, o que significa que um bom projeto deve ser constantemente aprimorado.

A seguir, apresentamos a definição de cada princípio de um bom projeto de código e sua relação com os maus-cheiros identificados por Fowler:

## Simplicidade

A simplicidade no código significa torná-lo o mais direto possível para atingir sua função, eliminando complexidades desnecessárias. Código simples é mais fácil de entender, modificar e manter.

**Maus-cheiros relacionados:**

- **Código duplicado:** ocorre quando trechos de código idênticos ou muito similares são repetidos, tornando o sistema mais difícil de atualizar.
- **Classe grande:** classes com múltiplas responsabilidades quebram a simplicidade e aumentam a dificuldade de manutenção.

## Elegância

Código elegante é aquele que expressa claramente sua intenção, utilizando as melhores práticas e convenções da linguagem de programação. Isso o torna mais legível e compreensível para outros desenvolvedores.

**Maus-cheiros relacionados:**

- **Código morto:** trechos de código que não são mais utilizados poluem o sistema e dificultam a leitura.
- **Código muito comentado:** quando muitos comentários são necessários para entender o código, isso indica que ele não é expressivo o suficiente.

## Modularidade

Modularidade significa dividir o código em partes menores, coesas e reutilizáveis. Cada módulo deve ter uma responsabilidade bem definida, reduzindo a interdependência com outros módulos.

**Maus-cheiros relacionados:**

- **Classe grande:** classes que concentram muitas funcionalidades deveriam ser separadas em diferentes módulos.
- **Método longo:** métodos extensos dificultam a reutilização e manutenção do código.

## Boas Interfaces

Interfaces bem projetadas garantem que a comunicação entre módulos ocorra de forma clara e intuitiva. Funções e classes devem expor apenas o necessário, evitando complexidade desnecessária.

**Maus-cheiros relacionados:**

- **Divergência de mudança:** quando mudanças frequentes em uma parte do sistema impactam múltiplos módulos, isso indica que a interface entre eles não está bem definida.
- **Mudança divergente:** ocorre quando uma única classe sofre alterações constantes por motivos distintos, indicando falta de coesão.

## Extensibilidade

Um sistema bem projetado deve permitir a adição de novas funcionalidades sem exigir grandes alterações na base de código. Para isso, é essencial seguir princípios como *Open-Closed* (aberto para extensão, fechado para modificação).

**Maus-cheiros relacionados:**

- **Inveja de função:** quando um método acessa mais dados de outra classe do que da própria, isso indica que a funcionalidade deveria estar em outro local.
- **Classe paralela:** ocorre quando duas classes desempenham funções muito semelhantes, tornando o código redundante.

## Evitar Duplicação

Repetir código aumenta a chance de erros e dificulta a manutenção. Sempre que possível, trechos de código similares devem ser extraídos para métodos reutilizáveis.

**Maus-cheiros relacionados:**

- **Código duplicado:** uma das principais causas de dificuldades na manutenção do software.

## Portabilidade

O código deve ser capaz de rodar em diferentes ambientes sem necessidade de grandes adaptações. Isso envolve o uso adequado de abstrações e padrões de projeto.

**Maus-cheiros relacionados:**

- **Acoplamento excessivo:** quando módulos ou classes são fortemente dependentes entre si, dificultando sua reutilização em outros contextos.

## Código Idiomático e Bem Documentado

O código deve seguir padrões e convenções da linguagem utilizada, tornando-o mais compreensível para outros desenvolvedores. Além disso, deve ser bem documentado, mas sem excesso de comentários desnecessários.

**Maus-cheiros relacionados:**

- **Nomes ruins:** quando variáveis, métodos ou classes têm nomes pouco descritivos, dificultando a compreensão do código.
- **Código morto:** trechos não utilizados poluem o código e devem ser removidos.

## 2. Identificação dos Maus-Cheiros no Trabalho Prático 2 do Grupo 19

### 1. Classe Grande (Large Class)

**Onde ocorre:** Classe `IRPF.java`
**Problema:**
A classe `IRPF` concentra múltiplas responsabilidades, como gerenciamento de rendimentos, deduções, dependentes e cálculo de imposto. Isso viola o **Princípio da Responsabilidade Única (SRP - Single Responsibility Principle)**, dificultando a manutenção e evolução do código.
**Refatoração sugerida:**
**Extrair Classe (Extract Class):** Criar classes especializadas, como:

- `GerenciadorRendimentos` para lidar com rendimentos e seus cálculos.
- `GerenciadorDeducoes` para processar deduções.
- `GerenciadorDependentes` para gerenciar dependentes.
- `CalculadoraImposto` para encapsular a lógica de cálculo do IRPF.

---

### 2. Método Longo (Long Method)

**Onde ocorre:** Método `calcularFaixasImposto()` na classe `IRPF.java`
**Problema:**
Este método contém uma longa cadeia de condicionais (`if-else`) para cálculo de imposto, dificultando a leitura e manutenção.
**Refatoração sugerida:**
**Extrair Método (Extract Method):** Dividir `calcularFaixasImposto()` em métodos menores, como:

- `calcularFaixaBase()`
- `calcularFaixaIntermediaria()`

- `calcularFaixaSuperior()`
  **Aplicar o Padrão Strategy:** Criar classes separadas para cada faixa de imposto, facilitando a manutenção e extensão futura.

---

### 3. Acoplamento Excessivo (Feature Envy)

**Onde ocorre:** Método `baseCalculo()` na classe `IRPF.java`
**Problema:**
O método acessa diretamente diversos getters de outras classes, como `getTotalRendimentosTributaveis()` e `getDeducao()`, indicando que ele **não está na classe mais apropriada**.
**Refatoração sugerida:**
**Mover Método (Move Method):** Transferir `baseCalculo()` para `CalculadoraImposto.java`, onde a lógica de cálculo deveria estar centralizada.

---

### 4. Alterações Espalhadas (Shotgun Surgery)

**Onde ocorre:** Validação de parentesco nos métodos `cadastrarPensaoAlimenticia()` e `getParentesco()`
**Problema:**
A lógica de validação está espalhada por múltiplos locais, tornando o código mais difícil de modificar e aumentando o risco de inconsistências.
**Refatoração sugerida:**
**Consolidar Condicional (Consolidate Conditional Expression):** Criar um método `validarParentescoParaPensao(String parentesco)`, que centralize a validação.

---

### 5. Nomes Pouco Descritivos (Poor Naming)

**Onde ocorre:** Métodos `getOutrasDeducoes()` e `getDeducao(String nome)`
**Problema:**
Métodos com **nomes genéricos** dificultam a compreensão do código.

**Refatoração sugerida:**
**Renomear Método (Rename Method):** Tornar os nomes mais descritivos, como:

- `getTotalDeducoesAdicionais()` (em vez de `getOutrasDeducoes()`).
- `getValorDeducao(String nome)` (em vez de `getDeducao(String nome)`).