

# Técnicas de Programação em Plataformas Emergentes

## Trabalho Prático - Entrega 3

### Grupo 2 - Integrantes:

- Bruno Henrique Moreira - 19/0134275
- Guilherme França Dib - 19/0108088
- João Gabriel Elvas - 19/0109599
- Milena Beatriz Aires - 19/0093625
- Rafael Bosi - 21/1029559

## Avaliação e Refatoração do Código da Classe IRPF

### Introdução

Neste documento, exploramos os princípios de um bom projeto de código e sua relação com os maus-cheiros descritos por Martin Fowler. Inicialmente, definimos cada princípio e sua importância para a qualidade do software. Em seguida, analisamos como a ausência desses princípios pode levar a problemas estruturais no código, comprometendo sua manutenção e evolução. Por fim, destacamos alguns maus-cheiros identificados em nosso projeto.

### Definições

- Simplicidade** – O código deve ser escrito da maneira mais clara e direta possível, sem complexidade desnecessária. Soluções simples são mais fáceis de entender, manter e modificar.
- Elegância** – Refere-se a um código que resolve problemas de forma eficiente e clara, utilizando os melhores padrões e práticas. Um código elegante é conciso, bem estruturado e evita redundâncias.
- Modularidade** – O sistema deve ser dividido em partes independentes (módulos) que podem ser desenvolvidas, testadas e mantidas separadamente. Isso melhora a reutilização do código e a organização do projeto.

- D. **Boas interfaces** – As interfaces dos módulos e componentes devem ser bem definidas e fáceis de usar. Um bom design de interface reduz a dependência entre módulos, facilitando a manutenção e evolução do software.
- E. **Extensibilidade** – O código deve ser projetado de forma a permitir a adição de novas funcionalidades sem a necessidade de grandes modificações no sistema existente. Isso pode ser alcançado por meio de princípios como o Open/Closed Principle da programação orientada a objetos.
- F. **Evitar duplicação** – Código repetitivo aumenta a chance de erros e dificulta a manutenção. Aplicar o princípio DRY ("Don't Repeat Yourself") significa consolidar funcionalidades semelhantes em um único local.
- G. **Portabilidade** – O código deve ser escrito de forma a poder ser executado em diferentes ambientes ou sistemas operacionais com o mínimo de esforço para adaptação. Isso é essencial para aumentar a longevidade e reutilização do software.
- H. **Código idiomático e bem documentado** – O código deve seguir as convenções e boas práticas da linguagem utilizada, tornando-o mais legível e compreensível para outros desenvolvedores. Além disso, deve conter comentários e documentação adequados para facilitar sua manutenção e evolução.

## Correlações com Fowler

- A. **Simplicidade** → *Complexidade desnecessária, Código morto, Código especulativo*
  - a. Complexidade desnecessária ocorre quando um código é mais complicado do que precisa ser, contrariando o princípio da simplicidade.
  - b. Código morto refere-se a partes do código que não são mais utilizadas, mas continuam presentes no sistema, tornando-o mais difícil de entender.
  - c. Código especulativo é a implementação de funcionalidades que podem nunca ser usadas, adicionando complexidade desnecessária.
- B. **Elegância** → *Código desorganizado, Métodos longos, Classes grandes*
  - a. Código desorganizado é aquele que não segue boas práticas de estruturação, prejudicando a leitura e a manutenção.
  - b. Métodos longos indicam falta de divisão de responsabilidades, tornando o código difícil de compreender e modificar.
  - c. Classes grandes centralizam muitas responsabilidades, tornando-as difíceis de testar e manter.

**C. Modularidade → *Divergência de funcionalidade, Dispersão de funcionalidades, Classes preguiçosas***

- a. Divergência de funcionalidade ocorre quando um mesmo módulo sofre **modificações frequentes para atender a diferentes propósitos.**
- b. Dispersão de funcionalidades acontece quando **funcionalidades relacionadas estão espalhadas em vários locais do código,** dificultando a manutenção.
- c. Classes preguiçosas são classes que **não justificam sua existência** porque realizam muito pouco, sendo um sinal de modularização inadequada.

**D. Boas interfaces → *Mudanças divergentes, Parâmetros longos, Inconsistência nas nomenclaturas***

- a. Mudanças divergentes ocorrem quando **várias alterações em uma interface são necessárias para cada nova funcionalidade,** sinalizando um design frágil.
- b. Parâmetros longos **dificultam a compreensão e o uso de métodos,** tornando o código menos intuitivo.
- c. Inconsistência nas nomenclaturas **gera confusão e dificulta a compreensão do código.**

**E. Extensibilidade → *Código rígido, Código frágil, Código impenetrável***

- a. Código rígido é difícil de modificar porque **pequenas mudanças exigem muitas alterações em diferentes partes do sistema.**
- b. Código frágil se **quebra facilmente ao realizar pequenas modificações,** tornando a evolução do software arriscada.
- c. Código impenetrável é de **difícil compreensão,** dificultando sua reutilização e extensão.

**F. Evitar duplicação → *Código duplicado***

- a. Código duplicado é um dos principais maus-cheiros, pois **repetições tornam a manutenção mais difícil** e aumentam as chances de inconsistências e bugs.

**G. Portabilidade → *Dependências excessivas, Código específico de plataforma***

- a. Dependências excessivas tornam o código **menos flexível e mais difícil de adaptar a** novos contextos.
- b. Código específico de plataforma **reduz a reutilização, dificultando a portabilidade d**o software para outros ambientes.

## H. Código idiomático e bem documentado → *Nomes ruins, Comentários excessivos ou ausentes, Código confuso*

- a. Nomes ruins dificultam o entendimento da função do código, tornando-o menos intuitivo.
- b. Comentários excessivos ou ausentes tornam difícil compreender a intenção do código, especialmente em sistemas complexos.
- c. Código confuso geralmente ocorre por falta de boas práticas idiomáticas da linguagem utilizada.

## Segunda parte

Analisando o arquivo IRPF.java, podemos identificar alguns maus-cheiros e os princípios de bom projeto que estão sendo violados. Abaixo estão os maus-cheiros identificados, os princípios violados e as operações de refatoração aplicáveis:

### Maus-cheiros Identificados

#### A. Complexidade desnecessária:

- a. A classe IRPF possui muitos atributos e métodos, tornando-a complexa e difícil de entender.

#### B. Classes grandes:

- a. A classe IRPF centraliza muitas responsabilidades, como cadastro de rendimentos, dependentes, deduções, cálculo de imposto, etc.

#### C. Métodos longos:

- a. Alguns métodos, como criarRendimento e cadastrarDeducaoIntegral, possuem lógica repetitiva e poderiam ser simplificados.

#### D. Código duplicado:

- a. A lógica de adicionar elementos a arrays é repetida em vários métodos (criarRendimento, cadastrarDependente, cadastrarDeducaoIntegral).

#### E. Dispersão de funcionalidades:

- a. Funcionalidades relacionadas ao cálculo de imposto, cadastro de rendimentos, dependentes e deduções estão todas na mesma classe, dificultando a manutenção.

#### F. Código rígido:

- a. A classe IRPF é difícil de modificar, pois pequenas mudanças podem exigir alterações em várias partes do código.

## Princípios Violados

### A. Simplicidade:

- a. A classe **IRPF é complexa e possui métodos longos e repetitivos.**
- B. Elegância:**
  - a. A classe não segue boas práticas de estruturação, resultando em código desorganizado e métodos longos.
- C. Modularidade:**
  - a. **A classe IRPF não está dividida em partes independentes,** dificultando o desenvolvimento, teste e manutenção.
- D. Boas interfaces:**
  - a. A classe possui **métodos com muitos parâmetros e lógica repetitiva,** dificultando a compreensão e uso.
- E. Extensibilidade:**
  - a. A classe é rígida e difícil de modificar, comprometendo a adição de novas funcionalidades.
- F. Evitar duplicação:**
  - a. **A lógica de adicionar elementos a arrays é repetida em vários métodos.**

## Operações de Refatoração Aplicáveis

- A. Extrair Classe:**
  - a. **Extrair funcionalidades relacionadas ao cadastro de rendimentos, dependentes e deduções** em classes separadas.
  - b. **Exemplo:** Criar classes `CadastroRendimentos()`, `CadastroDependentes()` e `CadastroDeduccoes()`.
- B. Extrair Método:**
  - a. **Extrair a lógica de adicionar elementos a arrays em um método separado.**
  - b. **Exemplo:** Criar um método `adicionarElemento()` para adicionar elementos a arrays.
- C. Substituir Método por Objeto-Método:**
  - a. **Encapsular a lógica de cálculo de imposto em uma classe separada.**
  - b. **Exemplo:** Criar uma classe `CalculadoraImposto()` para calcular o imposto.
- D. Reduzir Complexidade:**
  - a. Simplificar métodos longos e complexos, dividindo-os em métodos menores e mais específicos.
- E. Remover Código Duplicado:**
  - a. **Consolidar a lógica repetitiva em métodos reutilizáveis.**

## Exemplo de refatoração

Antes:

```
public void criarRendimento(String nome, boolean tributavel, float valor) {  
    //Adicionar o nome do novo rendimento  
    String[] temp = new String[nomeRendimento.length + 1];  
    for (int i=0; i<nomeRendimento.length; i++)  
        temp[i] = nomeRendimento[i];  
    temp[nomeRendimento.length] = nome;  
    nomeRendimento = temp;  
  
    //adicionar tributavel ou nao no vetor  
    boolean[] temp2 = new boolean[rendimentoTributavel.length + 1];  
    for (int i=0; i<rendimentoTributavel.length; i++)  
        temp2[i] = rendimentoTributavel[i];  
    temp2[rendimentoTributavel.length] = tributavel;  
    rendimentoTributavel = temp2;  
  
    //adicionar valor rendimento ao vetor  
    float[] temp3 = new float[valorRendimento.length + 1];  
    for (int i=0; i<valorRendimento.length; i++) {  
        temp3[i] = valorRendimento[i];  
    }  
    temp3[valorRendimento.length] = valor;  
    valorRendimento = temp3;  
  
    this.numRendimentos += 1;  
    this.totalRendimentos += valor;  
}
```

Depois:

```
public void criarRendimento(String nome, boolean tributavel, float valor) {  
    nomeRendimento = adicionarElemento(nomeRendimento, nome);  
    rendimentoTributavel = adicionarElemento(rendimentoTributavel, tributavel);  
    valorRendimento = adicionarElemento(valorRendimento, valor);  
    this.numRendimentos += 1;  
    this.totalRendimentos += valor;  
}  
  
private <T> T[] adicionarElemento(T[] array, T elemento) {  
    T[] temp = Arrays.copyOf(array, array.length + 1);  
    temp[array.length] = elemento;  
    return temp;  
}
```

*Para deixar o código mais organizado e fácil de trabalhar!*