| | | | |
|---|---|---|---|
| **TPPE:** | Turma 01 | **Semestre:** | 2024.2 |
| **Nome:** | Alexia Naara Da Silva Cardoso | **Matrícula:** | 202045007 |
| | Iago De Sousa Campelo Matos | | 202023743 |
| | João Lucas Pinto Vasconcelos | | 190089601 |
| | Leonardo Michalski Miranda | | 190046945 |
| | Maria Eduarda Dos Santos Abritta Ferreira | | 202016945 |
| **Professor:** | André Luiz Peron Martins Lanna | | |

## 1. Princípios de Bom Projeto de Código

### 1.1 Simplicidade

- **Definição:** O código deve fazer o mínimo necessário de forma clara e direta para que o programa funcione corretamente.
- **Relação com maus-cheiros:**
  - **Long Method:** Métodos excessivamente longos prejudicam a legibilidade.
  - **Long Parameter List:** Métodos com muitos parâmetros deixam o código difícil de entender e modificar.
  - **Excesso de Comentários:** Pode ser sinal de que o código não se expressa de forma simples.

### 1.2 Elegância

- **Definição:** Código otimizado, mas sem comprometer a clareza; fácil de manter, testar e seguir boas práticas.
- **Relação com maus-cheiros:**
  - **Excesso de Comentários:** Comentários desnecessários podem indicar que o código não é autoexplicativo.
  - **Long Method:** Dificulta a testabilidade e pode esconder oportunidades de simplificar a lógica.
  - **Large Class e Código Duplicado:** Classes grandes e duplicação de código indicam falta de elegância na organização.

### 1.3 Modularidade

- **Definição:** O software deve ser dividido em módulos ou componentes com responsabilidades bem definidas, facilitando a manutenção e evolução.
- **Relação com maus-cheiros:**

    - **Large Class:** Classes com muitas responsabilidades quebram o princípio da modularidade.
    - **Feature Envy:** Métodos que acessam muitos dados de outra classe apontam para a falta de distinção clara entre os módulos.

### 1.4 Boas Interfaces

- **Definição:** Interfaces claras, consistentes e simples, que ocultam detalhes desnecessários e expõem apenas o que for necessário para o uso externo.
- **Relação com maus-cheiros:**
    - **Inappropriate Intimacy:** Quando classes têm conhecimento excessivo dos detalhes internos de outras, as interfaces não foram definidas de forma ideal.
    - **Message Chains:** Cadeias de chamadas que indicam falta de abstração e um possível design pobre da interface.

### 1.5 Extensibilidade

- **Definição:** A capacidade de um software ser estendido ou adaptado a novas necessidades sem a necessidade de alterar significativamente a base existente.
- **Relação com maus-cheiros:**
    - **Divergent Change:** Quando uma classe sofre alterações por causa de múltiplas razões, seu design dificulta a extensão.
    - **Shotgun Surgery:** Alterações que exigem a modificação de muitos componentes indicam que o código não foi projetado para permitir extensões de forma isolada.

### 1.6 Evitar Duplicação

- **Definição:** Eliminar código duplicado para centralizar a lógica, facilitando a manutenção e reduzindo a probabilidade de erros.
- **Relação com maus-cheiros:**
    - **Código Duplicado:** Fórmulas, estruturas de decisão ou qualquer bloco de código que se repete podem levar a inconsistências e dificuldade para alterar a lógica central.

### 1.7 Portabilidade

- **Definição:** O software deve ser escrito de forma a ser executado em diferentes ambientes e plataformas com o mínimo de alterações.
- **Relação com maus-cheiros:**
    - **Dependência de Ambiente/Plataforma:** Código fortemente acoplado a recursos ou APIs de um ambiente específico reduz a portabilidade e dificulta a adaptação para outros cenários.

### 1.8 Código Deve Ser Idiomático e Bem Documentado

- **Definição:** O código precisa seguir as convenções e práticas comuns à linguagem de programação utilizada, além de ser bem documentado para facilitar a compreensão e manutenção.

- **Relação com maus-cheiros:**
    - **Nomes Inadequados e Comentários Excessivos/Insuficientes:** Se o código não segue as convenções idiomáticas próprias da linguagem, pode evidenciar confusão na sua organização e, por vezes, o uso inadequado (ou excesso) de comentários para explicar lógica não intuitiva.

## 2. Identificação dos Maus-Cheiros Persistentes no Trabalho Prático 2

**Análise por Classe**

**Deducao.java**

- **Code Smells:**
    - *Duplicação de Dados:* Listas separadas para nomes e valores de deduções poderiam ser substituídas por uma única lista de objetos (por exemplo, DeducaoItem).
    - *Falta de Encapsulamento:* Expor os arrays internos por meio de getters pode levar a alterações indesejadas.
    - *Baixa Coesão:* A classe junta responsabilidades de armazenamento de deduções e cálculo do total.
- **Operações de Refatoração Aplicáveis:**
    - Extrair uma classe DeducaoItem para encapsular nome e valor.
    - Substituir as duas listas por uma lista de DeducaoItem.
    - Remover ou proteger getters que expõem as estruturas internas.

**Dependente.java**

- **Code Smells:**
    - *Classe Anêmica:* Age somente como um DTO, com getters e setters, sem comportamento próprio.
    - *Falta de Validação:* Construtor sem validações para garantir integridade dos dados.
- **Operações de Refatoração Aplicáveis:**
    - Adicionar validações no construtor.
    - Incluir comportamentos relacionados aos dependentes, se aplicável, para agregar responsabilidades relacionadas à dedução.

**Imposto.java**

- **Code Smells:**
    - *Código Duplicado:* Uso repetido de lógica para cálculo de imposto com arrays hard-coded.
    - *Falta de Flexibilidade:* Faixas e alíquotas embutidas no código dificultam adaptações.
    - *Método Longo:* Lógica complexa que pode ser dividida.
- **Operações de Refatoração Aplicáveis:**
    - Extrair a lógica de cálculo para uma classe separada, como uma CalculadoraImposto.
    - Aplicar o padrão Strategy para permitir diferentes estratégias de cálculo.

    - Substituir arrays hard-coded por constantes ou configurações externas.

**ImpostoFaixa3.java**

- **Code Smells:**
    - *Classe Desnecessária:* Se a implementação não for genérica, pode causar a criação de diversas classes específicas para cada faixa.
    - *Acoplamento Forte:* Dependência direta na classe IRPF limita a reutilização e os testes.
- **Operações de Refatoração Aplicáveis:**
    - Generalizar a lógica de cálculo para que a classe seja reutilizável para outras faixas.
    - Aplicar interfaces ou uma classe base para desacoplar a lógica do IRPF.

**IRPF.java**

- **Code Smells:**
    - *Classe Grande:* Responsável por muitas atividades (rendimentos, dependentes, deduções, contribuições, pensões e cálculos de imposto).
    - *Código Duplicado:* Operações repetitivas na manipulação de arrays, como em criarRendimento e cadastrarDependente.
    - *Métodos Longos:* Alguns métodos agregam muita lógica sem separação clara de responsabilidades.
    - *Uso de Arrays Primitivos:* Dificulta manutenção; uso de coleções (List, Map) seria mais adequado.
    - *Falta de Encapsulamento:* Exposição direta de atributos que podem ser manipulados de fora da classe.
- **Operações de Refatoração Aplicáveis:**
    - Dividir a classe em responsabilidades menores (por exemplo, RendimentoManager, DependenteManager, DeducaoManager).
    - Substituir arrays por coleções Java, facilitando a manipulação de dados.
    - Extrair métodos repetitivos para uma classe de utilitários.
    - Aplicar o padrão Builder para simplificar a criação de objetos complexos.

**Rendimento.java**

- **Code Smells:**
    - *Classe Anêmica:* Funciona como um DTO, apenas com getters e setters.
    - *Falta de Validação:* Ausência de validações no construtor.
- **Operações de Refatoração Aplicáveis:**
    - Adicionar validações para garantir a consistência dos dados.
    - Incluir comportamentos ou métodos que encapsulam lógicas relacionadas ao rendimento, se pertinente.

**Problemas Gerais Observados**

- **Duplicação de Código:** Principalmente na manipulação de arrays e na lógica de cálculo de imposto.

- **Falta de Modularidade:** Classes com múltiplas responsabilidades dificultam a manutenção e testes.
- **Uso de Arrays Primitivos:** Em vez de utilizar coleções, o que compromete a escalabilidade e adaptabilidade do código.
- **Falta de Encapsulamento:** Exposição de atributos internos que podem ser modificados externamente.
- **Métodos Longos e Complexos:** Viola o princípio da responsabilidade única, dificultando a compreensão e manutenção.

## Conclusão

Este relatório apresenta uma análise detalhada dos princípios de bom projeto de código e dos maus-cheiros identificados no Trabalho Prático 2. As operações de refatoração sugeridas visam melhorar a qualidade do código, tornando-o mais simples, modular, extensível e fácil de manter. A implementação dessas melhorias contribuirá para um software mais robusto e alinhado com as melhores práticas de desenvolvimento.

## Referências

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code* .

Pressman, R. S. (2010). *Software Engineering: A Practitioner's Approach* .

Sommerville, I. (2010). *Software Engineering* .