Análise de Princípios de Bom Projeto de Código e Code Smells

Luis Henrique Luz Costa - 180066161

Pedro Lucas Garcia - 190115548

Taynara Cristina Ribeiro Marcellos - 211031833

1. Definição dos Princípios e sua Relação com Code Smells

1.1 Simplicidade

Definição: Um código simples é fácil de entender, direto e sem complexidades desnecessárias. Ele deve evitar sobrecarga de lógica e estruturas complicadas.

Code Smells Relacionados:

Shotgun Surgery: Código espalhado em vários lugares, tornando difícil modificar algo sem alterar diversas partes.
Divergent Change: Uma única classe sofre alterações frequentes por razões diferentes.

1.2 Elegância

Definição: Código elegante resolve problemas de forma concisa e eficiente, utilizando boas práticas de design.

Code Smells Relacionados:

Long Method: Métodos excessivamente longos que dificultam a compreensão.

Large Class: Classes que acumulam muitas responsabilidades.

1.3 Modularidade

Definição: Código modular divide-se em componentes independentes e reutilizáveis, facilitando manutenção e escalabilidade.

Code Smells Relacionados:

God Class: Uma única classe centraliza várias responsabilidades.
Feature Envy: Um método acessa mais atributos de outra classe do que da própria.

1.4 Boas Interfaces

Definição: Interfaces devem ser bem projetadas, intuitivas e fornecer apenas os métodos necessários para interação entre módulos.

Code Smells Relacionados:

Inappropriate Intimacy: Classes conhecem detalhes internos umas das outras.
Message Chains: Chamadas excessivas de métodos encadeados.

1.5 Extensibilidade

Definição: Um código bem projetado pode ser facilmente modificado ou expandido sem grandes retrabalhos.

Code Smells Relacionados:

Rigid Hierarchy: Estruturas de herança que dificultam mudanças.
Refused Bequest: Uma subclasse não utiliza métodos herdados da superclasse.

1.6 Evitar Duplicação

Definição: Código duplicado aumenta esforço de manutenção e risco de inconsistências.

Code Smells Relacionados:

Duplicated Code: Mesma lógica repetida em vários locais.
Speculative Generality: Código genérico sem necessidade real.

1.7 Portabilidade

Definição: Código portátil funciona em diferentes ambientes sem precisar de grandes adaptações.

Code Smells Relacionados:

Hardcoded Values: Uso de valores fixos em vez de configurações dinâmicas.
Platform Dependencies: Código acoplado a um ambiente específico.

1.8 Código Idiomático

Definição: Código que segue as convenções e práticas recomendadas da linguagem utilizada.

Code Smells Relacionados:

Switch Statements: Uso de estruturas switch em vez de polimorfismo.
Alternative Classes with Different Interfaces: Classes que fazem o mesmo, mas de maneiras distintas.

1.9 Código Bem Documentado

Definição: Código deve ter comentários claros, documentando propósito e funcionamento sem redundâncias.

Code Smells Relacionados:

Comments: Uso de comentários para explicar código mal escrito.
Obsolete Comments: Comentários que não refletem mais o código atual.

2. Análise dos Code Smells no Trabalho Prático 2

Análise de Maus-Cheiros na Classe IRPF.java

A classe IRPF apresenta diversos maus-cheiros de código que violam princípios de bom projeto. Abaixo estão os principais problemas identificados e suas respectivas justificativas.

2.1.1 Tamanho excessivo da classe (God Class)

A classe IRPF tem muitas responsabilidades distintas, como cadastro de rendimentos, dependentes, deduções e cálculo de imposto.

Princípio violado: Princípio da Responsabilidade Única (SRP Single Responsibility Principle).

Solução sugerida: Dividir a classe em classes menores e especializadas, como Rendimento, Dependente, Deducao e CalculadoraIRPF.

2.1.2 Uso excessivo de arrays para armazenar dados

A classe mantém múltiplas listas de dados como *nomeRendimento*, *valorRendimento*, *nomesDependentes*, entre outras, sem encapsulamento adequado.

Princípio violado: Encapsulamento e Coesão.

Solução sugerida: Criar classes específicas para representar rendimentos, deduções e dependentes, utilizando List<> ao invés de arrays primitivos.

2.1.3 Repetição de código (Duplicated Code)

Métodos como *criarRendimento*, *cadastrarDependente* e *cadastrarDeducaoIntegral* seguem um padrão repetitivo ao expandir arrays manualmente.

Princípio violado: DRY (Don't Repeat Yourself).

Solução sugerida: Utilizar coleções como List<> para evitar manipulação manual de arrays.

2.1.4 Uso de constantes numéricas explícitas (Magic Numbers)

Há diversos números fixos no código, como 2259.20f, 2826.65f, 3751.05f, etc.

Princípio violado: Princípio da Clareza e Manutenibilidade.

Solução sugerida: Criar constantes nomeadas para representar faixas de imposto e valores fixos.

2.1.5 Métodos longos e de difícil compreensão

Métodos como *calcularImpostoTotal* e *cadastrarDeducaoIntegral* realizam múltiplas operações e são difíceis de entender rapidamente.

Princípio violado: Princípio da Legibilidade e Simplicidade.

Solução sugerida: Refatorar esses métodos em funções menores e mais específicas.

2.1.6 Dependência desnecessária na classe CalculadoraAliquota

O método *calcularAliquotaEfetiva* instancia *CalculadoraAliquota* diretamente, criando um acoplamento forte.

Princípio violado: Inversão de Dependência (DIP Dependency Inversion Principle).

Solução sugerida: Utilizar injeção de dependência para desacoplar a classe.

Análise de Maus-Cheiros na Classe CalculadoraAliquota.java:

2.2.1 Uso de Tipos de Dados Primitivos (Primitive Obsession)

O código utiliza float para representar valores financeiros, o que pode levar a problemas de precisão devido à natureza da aritmética de ponto flutuante.

Sugestão: Utilizar BigDecimal para cálculos monetários e evitar possíveis erros de arredondamento.

2.2.2 Falta de Tratamento de Exceções

O código assume que os valores passados no construtor sempre serão válidos (não negativos ou inválidos).

Sugestão: Incluir validações no construtor para garantir que totalRendimentosTributaveis e impostoTotal não sejam negativos.

2.2.3 Responsabilidade Única (SRP Single Responsibility Principle)

A classe tem uma única responsabilidade: calcular a alíquota efetiva.

Ponto Positivo: Isso segue bem o princípio SRP.

A classe IRPF apresenta diversos problemas que impactam sua manutenção e extensibilidade. A refatoração recomendada envolve:

1. Separação de responsabilidades em classes menores.
2. Uso de estruturas de dados apropriadas (List<> em vez de arrays).
3. Evitar repetição de código através da reutilização de métodos.
4. Melhor organização do código para seguir princípios SOLID.

A classe *CalculadoraAliquota* tem um propósito bem definido, mas pode ser aprimorada com boas práticas, como uso de *BigDecimal* e tratamento de exceções

Essas melhorias tornarão o código mais limpo, modular e fácil de manter.

3. Propostas de refatoração

Refatorações Sugeridas para a Classe IRPF

3.1.1 Divisão da Classe em Múltiplas Classes (God Class)

Operação de Refatoração: Extrair Classes

Dividir a classe IRPF em várias classes menores, cada uma com uma responsabilidade específica.

Por exemplo:

Rendimento: Para gerenciar rendimentos.

Dependente: Para gerenciar dependentes.

Deducao: Para gerenciar deduções.

CalculadoraIRPF: Para cálculos relacionados ao IRPF.

Problema Resolvido: Reduz a responsabilidade única da classe IRPF, seguindo o Princípio da Responsabilidade Única (SRP).

3.1.2 Encapsulamento de Dados (Uso Excessivo de Arrays)

Operação de Refatoração: Substituir Arrays por Objetos

Criar classes específicas para representar rendimentos, deduções e dependentes, e utilizar *List*<> ao invés de arrays primitivos.

Problema Resolvido: Melhora o encapsulamento e a coesão, facilitando a manipulação de dados e evitando a manipulação manual de arrays.

3.1.3. Eliminação de Código Duplicado (Repetição de Código)

Operação de Refatoração: Extrair Método

Criar métodos reutilizáveis para operações repetitivas, como adicionar rendimentos, dependentes e deduções.

Problema Resolvido: Reduz a duplicação de código, seguindo o princípio DRY (Don't Repeat Yourself).

3.1.4 Substituição de Números Mágicos (Uso de Constantes Numéricas Explícitas)

Operação de Refatoração: Introduzir Constantes

Definir constantes nomeadas para representar valores fixos, como faixas de imposto.

Problema Resolvido: Melhora a clareza e a manutenção do código, facilitando a compreensão dos valores utilizados.

3.1.5 Refatoração de Métodos Longos (Métodos Longos e de Difícil Compreensão)

Operação de Refatoração: Extrair Método

Dividir métodos longos em funções menores e mais específicas, cada uma realizando uma única operação.

Problema Resolvido: Aumenta a legibilidade e simplicidade do código, facilitando a compreensão e manutenção.

3.1.6. Desacoplamento de Dependências (Dependência Desnecessária na Classe CalculadoraAliquota)

Operação de Refatoração: Introduzir Injeção de Dependência

Utilizar injeção de dependência para instanciar a classe CalculadoraAliquota, permitindo a substituição por outras implementações se necessário.

Problema Resolvido: Reduz o acoplamento forte entre as classes, seguindo o Princípio da Inversão de Dependência (DIP).

Refatorações Sugeridas para a Classe CalculadoraAliquota

3.2.1. Substituição de Tipos Primitivos (Uso de Tipos de Dados Primitivos)

Operação de Refatoração: Substituir Tipos Primitivos por Objetos

Utilizar BigDecimal para representar valores financeiros, garantindo precisão nos cálculos.

Problema Resolvido: Evita problemas de precisão devido à aritmética de ponto flutuante, melhorando a confiabilidade dos cálculos.

3.2.2 Validação de Dados (Falta de Tratamento de Exceções)

Operação de Refatoração: Introduzir Validação de Parâmetros

Adicionar validações no construtor para garantir que *totalRendimentosTributaveis* e *impostoTotal* não sejam negativos.

Problema Resolvido: Garante a integridade dos dados, evitando valores inválidos e melhorando a robustez do código.

SonarQube

Ademais, nós executamos o **SonarQube** para analisar o código e ele nos deu issues de manutenibilidade para refatorar, como podem ser vistas nestas duas imagens:

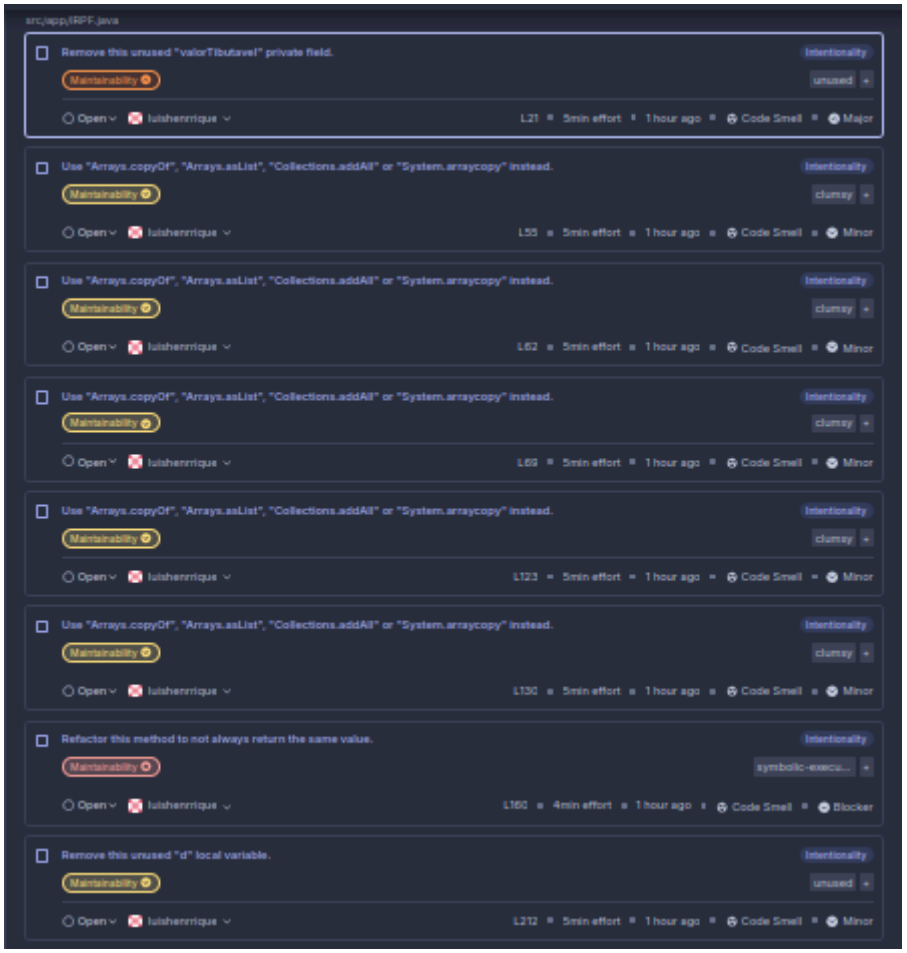

Figura 1 - Refatorações Issues de manutenibilidade

Autoria: Própria

Figura 2 - Refatorações Issues de manutenibilidade

Autoria: Própria

As operações de refatoração sugeridas visam melhorar a estrutura e a qualidade do código, tornando-o mais modular, legível e fácil de manter. A aplicação dessas refatorações

ajudará a seguir os princípios SOLID e outras boas práticas de desenvolvimento de software.

Referência:

Livros:

1. FOWLER, Martin. Refactoring: Improving the Design of Existing Code. 2. ed. Boston: Addison-Wesley, 2018.

2. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman, 2000.

3. BLOCH, Joshua. Effective Java. 2. ed. Boston: Addison-Wesley, 2008.

4. MARTIN, Robert C. Código Limpo: Habilidades Práticas do Agile Software. Rio de Janeiro: Alta Books, 2009.

5. MARTIN, Robert C. Agile Software Development: Principles, Patterns, and Practices. Upper Saddle River: Prentice Hall, 2002.

6. SOMMERVILLE, Ian. Engenharia de Software. 10. ed. São Paulo: Pearson, 2019.

Artigos e Sites:

1. ENGENHARIA de Software Moderna. Cap. 9: Refactoring. Disponível em: https://engsoftmoderna.info/cap9.html. Acesso em: 11 fev. 2025.

2. SAMMAN Coaching. Divergent Change. Disponível em: https://sammancoaching.org/code_smells/divergent_change.html. Acesso em: 11 fev. 2025.

3. SAMMAN Coaching. Message Chains. Disponível em: https://sammancoaching.org/code_smells/message_chains.html. Acesso em: 11 fev. 2025.

4. LUZKAN. Insider Trading. Disponível em: https://luzkan.github.io/smells/insider-trading. Acesso em: 11 fev. 2025.

5. C2 WIKI. Switch Statements Smell. Disponível em: https://wiki.c2.com/?SwitchStatementsSmell. Acesso em: 11 fev. 2025.

6. BITO.AI. Eliminating Shotgun Surgery: Examples and Refactoring Guide. Disponível em: https://bito.ai/blog/eliminating-shotgun-surgery/. Acesso em: 11 fev. 2025.

7. CELESTINO, André. Feature Envy. Disponível em: https://www.andrecelestino.com/feature-envy/. Acesso em: 11 fev. 2025.