

---

## Relatório de TPPE - Entrega 3

Ana Caroline Campos Rocha - 190083930

Gabriel da Silva Rosa - 202023681

Hellen Faria - 202016480

Laís Portela de Aguiar - 190046848

Matheus Raphael Soares de Oliveira - 190058587

# Entrega 3 - Projeto de código

## PERGUNTAS

1. Para cada um dos princípios de bom projeto de código mencionados acima, apresenta sua definição e relacione-o com os maus-cheiros de código apresentados por Fowler em sua obra.
  - a. **Simplicidade:** O código deve ser o mais simples possível para resolver o problema sem introduzir complexidade desnecessária. Dois exemplos dessa situação seriam o **código duplicado** que pode ser eliminado com métodos ou classes reutilizáveis. E outro cenário seriam as **classes longas** que seriam classes muito grandes que podem ser quebradas em classes menores com responsabilidades bem definidas.
  - b. **Elegância:** Código bem estruturado, fácil de entender e que resolve o problema de maneira eficiente. Dois exemplos de maus-cheiros relacionados seriam **nomes ruins** de variáveis, classes e métodos com nomes poucos descritivos que dificultam a leitura do código e o outro exemplo seria um **código morto** que são trechos de código que não são mais usados e poluem o projeto.
  - c. **Modularidade:** Dividir um sistema em módulos bem definidos ou unidades independentes que podem ser desenvolvidos ou testados de maneira isolada. ideia é distribuir as responsabilidades ao longo do sistema, facilitando sua manutenção. Um mau cheiro apontado por Fowler no livro sobre esse conceito é **funções longas (Long Method)**, que se caracteriza por ser um método difícil de manter e testar, devido a grande número de responsabilidade em uma única função. Para resolver isso se deve dividir a função em funções menores. Outra relação de mau cheiro seria a **classe grande**, onde uma classe detém muitas responsabilidades. Nesse caso, a solução seria refatorar a classe em unidades menores e mais coesas.
  - d. **Boas interfaces:** Uma boa interface é aquela que tem um conjunto claro de funcionalidade de um módulo ou componente sem expor detalhes internos. Ela deve ser intuitiva e fácil de usar por outros desenvolvedores do sistema, sem

---

necessidade de entender como ela foi implementada. Um mau cheiro para um interface seria a presença de **métodos com muitos parâmetros** o que indica que a interface foi mal desenhada, isso cria uma complexidade desnecessária e torna a compreensão e o uso da interface mais difíceis. Para solucionar este mau cheiro o ideal é fazer refatorações e introduzir classes de parâmetros. Outro mau cheiro seria **métodos com nomes confusos** que não indicam claramente a responsabilidade de um método, para ter uma interface melhor projetada é necessário que os nomes dos métodos sejam mais descritivos.

- e. **Extensibilidade:** A extensibilidade é a capacidade do código de ser modificado ou ampliado com o mínimo de impacto no sistema existente. Um sistema extensível permite a adição de novas funcionalidades sem a necessidade de modificar significativamente o código já escrito. Isso é alcançado por meio de boas práticas de projeto, como separação de responsabilidades, uso de padrões de projeto e arquitetura modular. Problemas como **Divergent Change** ocorrem quando uma classe precisa ser alterada por várias razões, indicando acúmulo de responsabilidades, enquanto **Shotgun Surgery** acontece quando uma pequena modificação exige alterações em vários pontos do sistema, demonstrando falta de modularidade. **Rigidez** também compromete a extensibilidade, pois significa que qualquer mudança impacta muitas partes do código, dificultando sua evolução.
- f. **Evitar duplicidade:** O conceito "**Don't Repeat Yourself**" (DRY) afirma que cada pedaço de conhecimento dentro de um sistema deve ter uma única representação clara e autoritativa. A duplicação de código pode levar a inconsistências, dificuldades de manutenção e aumento do esforço para corrigir erros. Problemas como **Duplicate Code** aparecem quando trechos idênticos de código são encontrados em vários locais, aumentando a complexidade e o risco de bugs. **Data Clumps** ocorrem quando grupos de variáveis são frequentemente usados juntos, mas sem uma estrutura organizada. **Long Method** indica que um método extenso pode conter código repetitivo que deveria ser extraído para funções menores. **Shotgun Surgery** também está relacionado à duplicação, pois quando uma mesma lógica precisa ser alterada em diversos locais, é um indício de código mal estruturado. Evitar esses maus-cheiros torna o código mais claro, fácil de manter e menos propenso a erros.
- g. **Portabilidade:** A portabilidade é a capacidade de um código ser executado, com pouca ou nenhuma modificação, em diferentes ambientes: podendo ser diferentes sistemas operacionais, arquiteturas de processadores, versões de compiladores, etc. Essa capacidade aumenta a vida útil de um software por reduzir dependências desnecessárias a plataformas específicas, aumenta a escalabilidade, permite migrações mais suaves: garante a continuidade dos serviços e reduz a quantidade de retrabalho, o que a torna fundamental para um

---

bom projeto de software. A principal forma em que a portabilidade é alcançada vem da abstração generalizada entre a lógica do aplicativo e as interfaces do sistema: respondendo ao princípio de separação de responsabilidades (**SRP - Single-Responsibility Principle**) e desacoplagem. Maus-cheiros de código, como **Divergent Change**, **Duplicate Code**, **Large Class** e **Shotgun Surgery**; todos afetam a portabilidade de um código, pois todos são sinais de um código muito acoplado e dependente, ou com responsabilidades muito grandes (no caso da **Large Class**), forçando que cada pequena mudança na lógica vire uma grande mudança no código, requerendo muito mais trabalho dos desenvolvedores/mantenedores.

- h. **Código idiomático e bem documentado:** O código idiomático deve seguir padrões aceitos da linguagem de programação utilizada, além de ser bem estruturado e documentado, a fim de garantir que outros desenvolvedores, ao entrarem no projeto, tenham a seu dispor um código mais compreensível, legível e de fácil manutenibilidade. Considerando este cenário de colaboração dentro de um projeto, a ausência de comentários e/ou documentação de métodos/classes, por exemplo, pode tornar o código difícil de compreender o, que está diretamente ligado ao mau cheiro de **“Code Smell: Insufficient Documentation”**. Outro ponto importante que pode ser afetado é o conceito **“Don't Repeat Yourself”**, citado anteriormente. Ao ser relacionado a um código mal documentado e não simplificado, considerando que isso o faz um código redundante, resulta no aumento da complexidade do mesmo e assim, contribui para a proliferação deste mau-cheiro de duplicação. Ademais, outro ponto central que é afetado pela boa empregabilidade de um código idiomático e bem documentado é a legibilidade do código. Quando os nomes de métodos, variáveis e classes são amplamente representativos, expressivos e indicam claramente o seu objetivo, é possível minimizar alguns maus-cheiros, como por exemplo, o **“Long Method”** e **“Poorly Named Variables”**. Em resumo, um código bem documentado e idiomático, não só facilita a leitura, mas também traz resultados positivos em relação a necessidade de intervenções complexas, tornando o código mais acessível e sustentável a longo prazo.
- 2. Identifique quais são os maus-cheiros que persistem no trabalho prático 2 do grupo, indicando quais os princípios de bom projeto ainda estão sendo violados e indique quais as operações de refatoração são aplicáveis. Atenção: não é necessário aplicar as operações de refatoração, apenas indicar os princípios violados e operações possíveis de serem aplicadas.
  - a. **Classe IRPF**
    - i. A classe IRPF gerencia diferentes responsabilidades (rendimento, dependentes e deduções), podendo até se confundir dependendo do

---

cenário. Uma sugestão de melhoria seria aplicar o princípio de Responsabilidade Única (SRP), uma arquitetura, separando responsabilidades em classes distintas.

**b. Classe Deducao**

- i. Os métodos **adicionarNome** e **adicionarValor** na classe **Deducao** seguem a mesma lógica. Uma boa refatoração seria criar um método genérico para adicionar elementos a um array.
- ii. Métodos **expandArray**, **adicionarNome** e **adicionarValor** fazem cópia manual dos arrays. Uma melhoria possível seria usar `ArrayList` no lugar de arrays fixos.
- iii. **Deducao** tem uma instância de **Dependente** dentro dela, mas também recebe um **Dependente** como parâmetro. Uma melhoria possível seria remover dependências desnecessárias e garantir que uma única instância seja usada. Mas tem que ser analisada, pois dependendo não seria possível.
- iv. O método **getDeducao()** em **Deducao** é redundante com **getDeducao(Dependente dependenteManager)**. Uma melhoria seria fundir métodos redundantes para evitar confusão. Porém essa melhoria deve ser refletida nos testes, para que não acabe quebrando.
- v. Os métodos **getDeducao()** e **getOutrasDeducoes()**, por exemplo, não possuem explicações sobre o que estão realizando, dificultando assim a compreensão do código por outros desenvolvedores.

**c. Classe Dependente**

- i. O código atual mistura várias responsabilidades dentro de métodos e classes, o que dificulta a manutenção e extensibilidade. Por exemplo, o método **cadastrarDependente** não só adiciona um novo dependente, mas também expande arrays manualmente, o que viola o princípio de modularidade. Uma maneira de melhorar isso seria extrair responsabilidades para métodos separados, como uma função para expandir arrays, e uma classe ou método único para o cadastro do dependente. Isso melhoraria a clareza e a manutenção do código.
- ii. O código de expansão dos arrays **nomesDependentes** e **parentescosDependentes** está duplicado nos métodos de cadastro de dependentes. Ambos os arrays são expandidos da mesma forma, o que gera redundância e aumenta a complexidade. Torna o código repetido, mantendo o método difícil de entender.

- 
- iii. O código utiliza arrays fixos para armazenar dados dos dependentes, o que exige o uso manual de lógica para expansão. Isso pode ser um obstáculo para a simplicidade e portabilidade do código, já que arrays têm limitações em termos de flexibilidade. Uma solução seria substituir os arrays por Listas (ArrayList), que permitem manipulações mais dinâmicas e simplificam o código, tornando-o mais eficiente e fácil de manter.
  - iv. Alguns métodos, como **getParentesco**, não têm nomes tão descritivos quanto poderiam ter. O nome do método poderia ser mais claro, como **getParentescoDeDependente**, para refletir de forma mais precisa o que ele faz.

#### d. Classe Rendimento

- i. O código apresenta o mau-cheiro de **Data Clumps**, pois mantém três arrays separados (nomeRendimento, rendimentoTributavel e valorRendimento) que sempre precisam ser manipulados juntos. Isso gera um alto risco de inconsistência e dificulta a manutenção do código, tornando cada operação mais complexa do que deveria. Uma solução mais elegante seria criar uma classe RendimentoItem que encapsulasse essas informações em um único objeto, reduzindo a fragmentação dos dados e melhorando a coesão do código.
- ii. Outro problema presente é o **Primitive Obsession**, que se manifesta no uso direto de arrays ao invés de coleções mais apropriadas como List<>. O código manipula arrays manualmente, exigindo que cada nova entrada seja adicionada através da expansão manual do array, o que não é idiomático em Java e pode levar a problemas de desempenho e complexidade desnecessária. Utilizar uma List<RendimentoItem> eliminaria essa necessidade e tornaria o código mais limpo e extensível.
- iii. A classe também apresenta **Violation of DRY** (Don't Repeat Yourself) devido à repetição da lógica nos métodos expandArray(), que têm implementações quase idênticas para diferentes tipos de dados (T[], float[], boolean[]). Essa duplicação torna o código mais difícil de manter e modificar, pois qualquer alteração na forma como os arrays são manipulados precisaria ser feita em múltiplos lugares. A melhor abordagem seria evitar essa repetição eliminando a necessidade de expansão de arrays, utilizando uma estrutura de dados dinâmica.

- 
- iv. Outro mau-cheiro identificado é o **Lack of Documentation**, pois não há comentários explicando a funcionalidade dos métodos e atributos da classe. A ausência de documentação torna o código menos compreensível para outros desenvolvedores e pode dificultar futuras manutenções. Mesmo que o código seja funcional, ele deve ser legível e explicativo para facilitar sua reutilização e entendimento. Incluir comentários e utilizar nomes de métodos e variáveis mais descritivos melhoraria significativamente a clareza do código.
  - v. Por fim, o método `calcularImpostoFaixa` apresenta o mau-cheiro **Long Method**, pois contém lógica que poderia ser quebrada em funções menores para melhorar a legibilidade. Atualmente, a forma como a ocupação da faixa é calculada torna a compreensão menos intuitiva, exigindo uma análise detalhada para entender seu funcionamento. Refatorá-lo para separar cálculos intermediários e dar nomes mais descritivos às variáveis ajudaria a tornar o código mais simples e fácil de ler.