Universidade de Brasília
UnB/FGA – Engenharia de Software
Técnicas de Programação para Plataformas Emergentes - 2024/2

Trabalho Prático

Membros - Grupo 17:
Guilherme Nishimura da Silva - 200030264
Karla Chaiane da Silva Feliciano - 200021541
Matheus Costa Gomes - 190093331
Pedro Henrique Nogueira Bragança - 190094478

Professor: André Lanna

1 - Para cada um dos princípios de bom projeto de código mencionados acima, apresente sua definição e relacione-o com os maus-cheiros de código apresentados por Fowler em sua obra.

Princípios de um Código Limpo e Bem Estruturado

---

1. Simplicidade

Definição: O código deve ser simples, evitando complexidade desnecessária. A simplicidade facilita a leitura, a manutenção e a evolução do software.

Relação com maus-cheiros de código:

- **Código duplicado (Duplicated Code):** Repetições aumentam a complexidade e dificultam a manutenção. A simplicidade sugere a eliminação dessas duplicatas por meio da extração de métodos ou classes reutilizáveis.
- **Método longo (Long Method):** Métodos muito extensos são difíceis de compreender e modificar. A simplicidade recomenda dividir métodos longos em métodos menores e mais focados.
- **Classe grande (Large Class):** Classes com muitas responsabilidades são complexas. A simplicidade sugere dividir classes grandes em classes menores e coesas.

2. Elegância

Definição: O código deve ser bem estruturado, claro e expressivo, refletindo uma solução bem pensada para o problema.

Relação com maus-cheiros de código:

- **Código morto (Dead Code):** Código que não é mais utilizado polui o projeto e prejudica a elegância. Deve ser removido para manter o código limpo.
- **Nomes obscuros (Mysterious Name):** Variáveis, métodos ou classes com nomes pouco descritivos dificultam a compreensão do código. A elegância sugere nomes claros e significativos.

3. Modularidade

Definição: O código deve ser organizado em módulos coesos e com responsabilidades bem definidas, promovendo a separação de conceitos.

Relação com maus-cheiros de código:

- **Classe grande (Large Class):** Classes que fazem muitas coisas violam a modularidade. A solução é dividir a classe em módulos menores.
- **Intimidade inapropriada (Inappropriate Intimacy):** Quando classes ou módulos conhecem detalhes internos uns dos outros, a modularidade é comprometida. A solução é reduzir o acoplamento.

4. Boas Interfaces

Definição: As interfaces devem ser claras, bem definidas e estáveis, facilitando a interação entre módulos e componentes.

Relação com maus-cheiros de código:

- **Interface grande (Large Interface):** Interfaces com muitos métodos tornam o código difícil de entender e manter. Boas interfaces devem ser minimalistas e focadas.
- **Mensagens encadeadas (Message Chains):** Quando um objeto depende de uma cadeia de chamadas para acessar outro objeto, a interface fica frágil. Boas interfaces devem ser diretas e independentes.

5. Extensibilidade

Definição: O código deve ser projetado para permitir mudanças futuras sem grandes refatorações.

Relação com maus-cheiros de código:

- **Código rígido (Rigidity):** Um código que não pode ser facilmente alterado prejudica a extensibilidade. O uso de padrões de projeto como Strategy e Decorator facilita a evolução do sistema.
- **Código frágil (Fragility):** Código que quebra facilmente ao ser modificado não é extensível. A solução é reduzir o acoplamento e aumentar a coesão.

6. Evitar Duplicação

Definição: O código deve evitar repetições desnecessárias, promovendo a reutilização e a manutenção consistente.

Relação com maus-cheiros de código:

- **Código duplicado (Duplicated Code):** A duplicação é um dos principais maus-cheiros, pois dificulta a manutenção e introduz inconsistências. A solução é extrair métodos ou classes comuns.

7. Portabilidade

Definição: O código deve ser projetado para funcionar em diferentes ambientes ou plataformas com o mínimo de modificações.

Relação com maus-cheiros de código:

- **Código acoplado a plataforma (Platform Dependency):** Código que depende fortemente de detalhes específicos de uma plataforma reduz a portabilidade. A solução é usar abstrações ou camadas de compatibilidade.

8. Código Idiomático e Bem Documentado

Definição: O código deve seguir as convenções e padrões da linguagem ou framework utilizado, além de ser bem documentado para facilitar o entendimento.

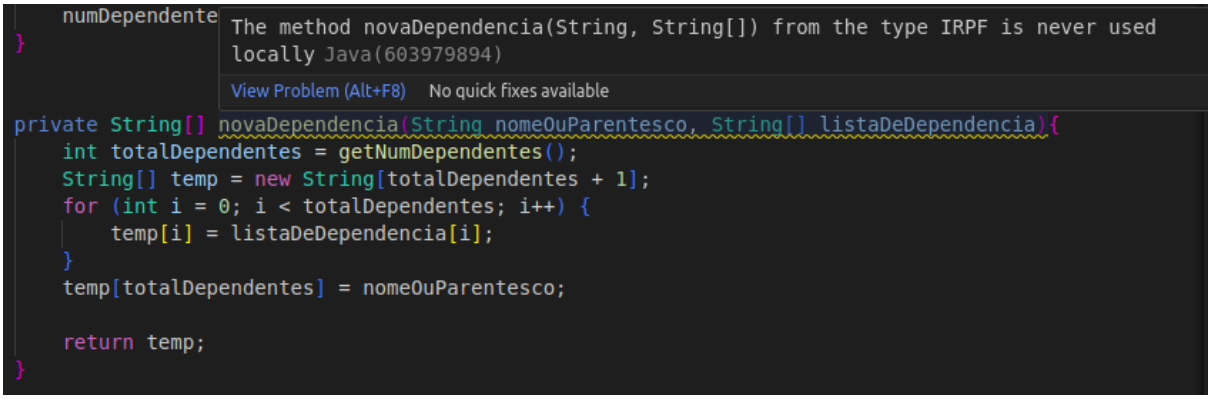
Relação com maus-cheiros de código:

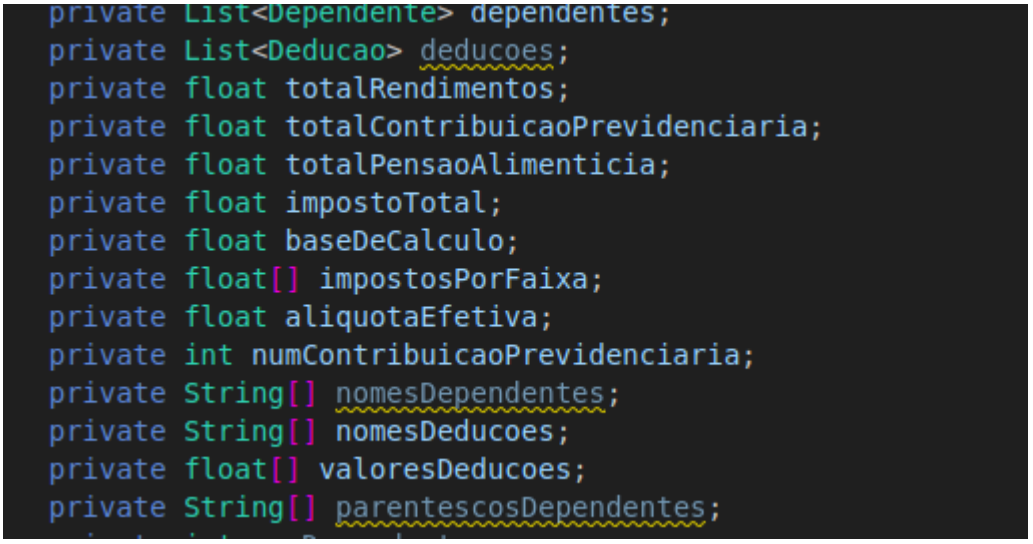
- **Comentários inadequados (Comments):** Comentários excessivos ou desnecessários podem indicar código pouco claro. A solução é melhorar a clareza do código em vez de depender de comentários.
- **Nomes obscuros (Mysterious Name):** Nomes pouco descritivos tornam o código difícil de entender. Seguir padrões de nomenclatura adequados melhora a legibilidade.

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.

1. Dead Code

Na classe IRPF foram encontrados maus cheiros do tipo **código morto** (*dead code*): a função *novaDependencia* passou a não ser mais utilizada após a aplicação da técnica de Extrair Classe para gerar a classe Dependente, bem como alguns atributos da classe que deixaram de ser utilizados mas que permaneceram no código (sublinhados em amarelo):





Uma solução simples para este caso é a remoção dos trechos não utilizados

2. Data Clumps

Existem listas separadas para “deduções”, como *nomesDeducoes* e *valoresDeducoes* que aparecem frequentemente juntos, como é possível ver na imagem abaixo. Porém, sendo utilizados como de arrays separados dificulta a manutenção e coerência dos dados.
A solução ideal é agrupar esses valores em um objeto “Dedução” tendo nome e valor como atributos, e recuperar apenas uma lista de deduções.

```java
// Deduções
public void cadastrarDeducaoIntegral(String nome, float valorDeducao) {
    String temp[] = new String[nomesDeducoes.length + 1];
    for (int i = 0; i < nomesDeducoes.length; i++) {
        temp[i] = nomesDeducoes[i];
    }
    temp[nomesDeducoes.length] = nome;
    nomesDeducoes = temp;

    float temp2[] = new float[valoresDeducoes.length + 1];
    for (int i = 0; i < valoresDeducoes.length; i++) {
        temp2[i] = valoresDeducoes[i];
    }
    temp2[valoresDeducoes.length] = valorDeducao;
    valoresDeducoes = temp2;
}
```

3. Long Class

A classe IRPF está fazendo muitas coisas ao mesmo tempo, lidando com cadastros e recuperação de rendimentos, deduções, dependentes e vários cálculos fiscais, exemplificado nas imagens abaixo, violando o princípio da responsabilidade única (SRP - Single Responsibility Principle).

```java
public void cadastrarDependente(String nome, String parentesco) {
    Dependente dependente = new Dependente(nome, parentesco);
    dependentes.add(dependente);
    numDependentes++;
}
```

```java
public void calcularAliquotaEfetiva(float rendimentosTributaveis, float impostoDevido) {
    if (rendimentosTributaveis == 0) {
        aliquotaEfetiva = 0; // Evitar divisão por zero
    } else {
        aliquotaEfetiva = (impostoDevido / rendimentosTributaveis) * 100;
    }
}
```

Podemos refatorar extraindo classes menores como GestorDeRendimentos, GestorDeDeducoes, GestorDeDependentes, e CalculadoraIRPF para encapsular esses comportamentos específicos. Cada classe deve se responsabilizar por sua parte do código, reduzindo o acoplamento e facilitando testes.

Referências

- Martin Fowler. Refactoring: Improving the design of Existing Code. Addison-Wesley Professional, 1999.
- Pete Goodliffe. Code Craft: The practice of Writing Excellent Code. No Starch Press, 2006.