FCTE - Campus UnB Gama
TPPE - Técnicas De Programação Em Plataformas Emergentes
Prof. Dr. André Luiz Peron Martins Lanna
Alunos - Abraão Alves Ribeiro 19/0023376, Bernardo Chaves Pissutti 19/0103302, Jefferson França Santos 18/0102761, Paulo Henrique Almeida da Silva 17/0020291

ENTREGA 3 - PROJETO DE CÓDIGO

1 PRINCÍPIOS DE BOM PROJETO DE CÓDIGO E RELAÇÃO COM MAUS-CHEIROS

1.1 Simplicidade

A simplicidade diz respeito à escrita de um código direto ao ponto e sem complexidade desnecessária. Um design simples é fácil de entender, implementar e manter.

Esse princípio tem relação com os maus-cheiros:

- *Long Function*: Métodos longos dificultam a compreensão e manutenção do código. O livro sugere que funções devem ser pequenas e focadas em uma única responsabilidade;
- *Long Parameter List*: Muitos parâmetros tornam o código confuso e aumentam o risco de erro. Isso fere a simplicidade, pois torna o código mais difícil de entender; e
- *Large Class*: Classes grandes concentram muitas responsabilidades, violando a simplicidade ao invés de dividir o código em componentes menores e coesos.

1.2 Elegância

A elegância está associada a uma solução que, além de funcional, apresenta clareza, concisão e harmonia. Um código elegante resolve o problema de forma precisa, sem redundâncias e com uso adequado dos recursos da linguagem.

Esse princípio tem relação com os maus-cheiros:

- *Shotgun Surgery*: Se uma modificação exige mudanças em vários arquivos ou classes, o código não é elegante, pois não está bem estruturado; e

- *Divergent Change*: Se uma classe precisa ser alterada por múltiplas razões não relacionadas, isso indica que a coesão foi comprometida, prejudicando a elegância do código.

1.3 Modularidade

Modularidade é dividir o sistema em componentes ou módulos bem definidos, onde cada um possui uma responsabilidade única. Essa divisão melhora o encapsulamento, o reuso e a manutenção do código.

Esse princípio tem relação com os maus-cheiros:

- *Feature Envy*: Quando um método acessa mais os dados de outra classe do que os seus próprios, significa que a funcionalidade está no lugar errado, comprometendo a modularidade; e
- *Divergent Change*: Classes que mudam por várias razões quebram a modularidade e indicam baixa coesão.

1.4 Boas Interfaces

Interfaces bem projetadas são claras, simples e coesas e seguem princípios de particionamento, abstração, compressão e substitutibilidade, para oferecer uma boa interação entre os módulos do sistema. Isso implica métodos com nomes expressivos, parâmetros bem definidos e pouca exposição de detalhes internos.

Esse princípio tem relação com os maus-cheiros:

- *Middle Man*: Quando uma classe apenas delega chamadas para outra sem adicionar valor, sua presença não melhora a interface do código e pode ser removida; e
- *Data Clumps*: Dados que frequentemente aparecem juntos deveriam ser encapsulados em uma única entidade para melhorar a coesão e clareza da interface.

1.5 Extensibilidade

Um código extensível permite a inclusão de novas funcionalidades com impacto mínimo no que já foi implementado. O sistema deve ser preparado para evoluir sem a necessidade de grandes alterações estruturais.

Esse princípio tem relação com os maus-cheiros:

- Speculative Generality: Adicionar código que ainda não é necessário compromete a extensibilidade, pois pode dificultar futuras alterações ao invés de prepará-las; e
- Divergent Change: Uma classe que precisa ser alterada frequentemente para acomodar novas funcionalidades pode indicar um design que não é realmente extensível.

1.6 Evitar Duplicação

A duplicação de código é uma fonte frequente de erros e inconsistências, por exemplo, pode ocorrer de, por conta da duplicação de código, existir o mesmo bug em partes diferentes do código, e ao corrigir o bug, este pode só ser corrigido em uma das partes, enquanto a outra continua com o bug.

Esse princípio tem relação com os maus-cheiros:

- *Duplicated Code*: O livro enfatiza que código duplicado deve ser eliminado, pois torna a manutenção mais difícil e aumenta a chance de inconsistências.

1.7 Portabilidade

Portabilidade envolve o desenvolvimento de código que possa ser executado em diferentes ambientes e plataformas com o mínimo de adaptações,de acordo com o que foi definido nos requisitos do projeto. Envolve o desenvolvimento de uma camada de abstração para a plataforma que o código pretende ser executado.

Esse princípio tem relação com os maus-cheiros:

- *Shotgun Surgery*: Quando código específico de uma plataforma está espalhado, a adaptação para outra plataforma exige modificações em vários lugares, prejudicando a portabilidade; e
- *Divergent Change*: A mistura de código genérico com código dependente de plataforma resulta em classes que precisam mudar por múltiplas razões, dificultando a portabilidade.

1.8 Código Idiomático e Bem Documentado

Um código idiomático utiliza as convenções específicas da linguagem que está sendo utilizada no projeto. E, ter uma documentação clara e objetiva ajuda na compreensão do código e dispensa a necessidade do programador novo no projeto inferir a estrutura do projeto.

Esse princípio tem relação com os maus-cheiros:

- *Comments*: O livro menciona que muitos comentários são usados para mascarar código ruim, indicando que o código poderia ser melhor estruturado para se explicar por si mesmo;
- *Long Function*: Funções longas geralmente necessitam de mais comentários, pois são difíceis de entender sem explicações adicionais; e
- *Poor Naming*: Nomes ruins tornam o código confuso e exigem mais documentação para explicar sua intenção.

2 ANÁLISE DE MAUS-CHEIROS NO TRABALHO PRÁTICO 2

2.1 Large Class

A classe IRPF (arquivo *IRPF.java*) gerencia múltiplas responsabilidades: rendimentos, dependentes, contribuições previdenciárias, deduções e pensões. Isso é evidenciado pelos campos rendimentos, dependentes, deducoes e métodos como *criarRendimento()*, *cadastrarDependente()*, e *cadastrarContribuicaoPrevidenciaria()*.

Para refatorar esse código pode-se utilizar a operação de extrair classe. Onde Cria-se classes como *GerenciadorRendimentos* (para métodos *criarRendimento*, *getTotalRendimentosTributaveis*) e *GerenciadorDependentes* (para *cadastrarDependente*, *getParentesco*), segregando responsabilidades.

2.2 Duplicated Code

Os métodos *criarRendimento()*, *cadastrarDependente()*, e *cadastrarDeducaoIntegral()* (em *IRPF.java*) repetem a lógica de redimensionamento de arrays manualmente, usando loops para copiar elementos.

Para refatorar esse código pode-se utilizar a operação de extrair método, extraindo a lógica de redimensionamento em um método utilitário *expandirArray()*.

2.3 Long Function

O método *calcularImpostoPorFaixa()* (em *CalcularImposto.java*) possui uma cadeia extensa de condicionais para calcular impostos por faixa, tornando-o difícil de manter.

Para refatorar esse código pode-se utilizar a operação de extrair método, dividindo o método em funções menores como *calcularFaixaBase()* e *calcularFaixaIntermediaria()*.

2.4 Data Clumps

Os campos *numRendimentos* e *totalRendimentos* (em *IRPF.java*) são sempre usados em conjunto, mas estão dispersos na classe.

Para refatorar esse código pode-se utilizar a operação de introduzir objeto parametro. Onde os campos são agrupados em um objeto *RegistroRendimentos*, com atributos quantidade e *valorTotal*.

2.5 Data Class

Classes como Rendimento, Dependente, e Deducao (arquivos homônimos) são estruturas passivas, com apenas campos e getters, sem comportamentos.

Para refatorar esse código pode-se utilizar a operação de encapsular campo, passando para campos mutáveis (ex: *totalPensaoAlimenticia*) privados e controlar modificações via métodos.

2.6 Mutable Data

Campos como *totalPensaoAlimenticia* (em *IRPF.java*) são modificados diretamente no método *cadastrarPensaoAlimenticia()*, sem encapsulamento.

Para refatorar esse código pode-se utilizar a operação de encapsular passando os campos como *totalPensaoAlimenticia* para atributos privados e controlar atualizações via métodos (ex: *adicionarPensao()*).

2.7 Comments

Comentários como */** Retorna o número de rendimentos... **/* em *getNumRendimentos()* (em *IRPF.java*) são redundantes, pois o nome do método já é claro.

Para refatorar esse código não é necessário uma operação específica, pode ser feito removendo comentários que não agregam informação útil.

2.8 Feature Envy

O método *baseDeCalculoImposto()* (em *IRPF.java*) acessa múltiplos getters (*getTotalRendimentosTributaveis()*, *getDeducao()*) para realizar cálculos que poderiam estar centralizados.

Para refatorar esse código pode-se utilizar a operação de mover método, transferindo o método *baseDeCalculoImposto()* para a classe *CalcularImposto*, onde a lógica de cálculo é centralizada.

2.9 Shotgun Surgery

Refatoração: Esconder Delegação

A lógica de validação de parentesco (ex: *parentesco.toLowerCase().contains("filh")*) está espalhada em *cadastrarPensaoAlimenticia()* e *getParentesco()* (em *IRPF.java*).

Para refatorar esse código pode-se utilizar a operação de consolidar condicional, com a criação de um método *validarParentescoParaPensao(String parentesco)* para centralizar a regra, reduzindo pontos de modificação.

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.