Universidade de Brasília
Faculdade de Ciências e Tecnologias em Engenharias - FCTE

TÉCNICAS DE PROGRAMAÇÃO EM PLATAFORMAS EMERGENTES
Docente: André Luiz Peron Martins Lanna

Trabalho Prático

Entrega 3 - Projeto de código/software

Gabrielly Assuncao Rodrigues - 200018442
Laís Ramos Barbosa - 170107574
Maria Eduarda Barbosa Santos - 200023934
Renann de Oliveira Gomes - 200043030
Vitor Borges dos Santos - 200028626

Brasília, DF
2025

1. Introdução

O projeto de software é frequentemente entendido como uma atividade inicial do desenvolvimento, na qual são definidas as estruturas e interações do sistema. No entanto, a prática demonstra que a codificação também é uma atividade de projeto. Como apontado por Martin Fowler (1999), o aprimoramento do código ocorre continuamente, sem alterar seu comportamento externo, evidenciando que programar é uma atividade de refinamento do projeto de software.

A cada nova funcionalidade implementada, os programadores devem ajustar o projeto, garantindo a coerência, sustentabilidade e evolução do mesmo. Para que isso ocorra de maneira eficiente, é necessário seguir princípios fundamentais de um bom projeto de código, como simplicidade, modularidade, elegância, boas interfaces, extensibilidade, evitar duplicação, portabilidade e escrita idiomática e bem documentada.

Contudo, desvios desses princípios podem resultar em "maus-cheiros" de código, que comprometem a qualidade do software. Fowler (1999) identifica esses problemas e sugere técnicas de refatoração para mitigá-los, permitindo que um código mal estruturado seja melhorado sem comprometer sua funcionalidade. Este trabalho apresenta a relação entre os princípios de bom projeto e os maus-cheiros de código, com base nas entregas anteriores e todas as refatorações realizadas.

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

2.1. Simplicidade:

2.1.1. **Definição:** O código deve ser claro, direto e sem complexidade desnecessária. Soluções simples são mais fáceis de entender e de manter.

2.1.2. **Maus-cheiros relacionados:** Alguns maus-cheiros de código, como o **Método Longo (Long Method)**, indicam a falta de simplicidade. Quando um método é muito grande, ele geralmente reflete a tentativa de fazer muitas coisas ao mesmo tempo, tornando o código difícil de entender e de manter. Além disso, o uso excessivo de **Comentários Excessivos (Comments)** pode ser um sinal de que o código não é simples o suficiente para ser entendido sem muitas explicações. Em vez disso, o código deveria ser claro o suficiente para não precisar de tantos comentários, conforme Fowler. Outro ponto relacionado à simplicidade é a **Classe Grande (Large Class)**, que ocorre quando uma classe assume muitas responsabilidades. Isso contraria a ideia de simplicidade, pois classes com muitas

responsabilidades podem se tornar complexas e difíceis de entender. A modularidade do código, como a separação de responsabilidades em diferentes classes, pode ajudar a evitar esse mau cheiro, promovendo um código claro e organizado.

2.2. Modularidade:

2.2.1. **Definição:** O código deve ser dividido em partes menores e bem definidas, que podem ser desenvolvidas e testadas de forma independente.

2.2.2. **Maus-cheiros relacionados:** Um exemplo de code smell que afeta a modularidade é a **Classe Grande (Large Class)**. Pois como já vimos no tópico anterior, quando uma classe assume muitas responsabilidades, ela se torna difícil de entender, testar e modificar. Conforme disse Fowler em sua obra, classes sobrecarregadas fogem do princípio da separação de responsabilidades, tornando o código menos modular e mais propenso a erros. Juntamente a isso tem os **Métodos Longos (Long Method)**, que dificulta a modularização do código. O ideal é dividi-los em partes menores, garantindo que cada método tenha uma única responsabilidade. Além disso, a **Dependência Excessiva (Feature Envy)** também prejudica a modularidade. Esse code smell ocorre quando um método de uma classe acessa com frequência os atributos ou métodos de outra classe, indicando uma separação inadequada de responsabilidades. Segundo Fowler, essa dependência excessiva pode ser um sinal de que a lógica do método deveria estar localizada na própria classe que contém os dados relevantes.

2.3. Elegância:

2.3.1. **Definição:** Um código elegante deve ser estruturado de forma clara e eficiente, evitando redundâncias e soluções muito complicadas.

2.3.2. **Maus-cheiros relacionados:** Um dos problemas que afetam a elegância é a **Duplicação de Código (Duplicated Code)** quando um trecho de código aparece várias vezes e em várias partes do sistema, qualquer alteração necessário exige modificações em múltiplos locais, aumentando o risco de inconsistências e dificultando a manutenção. Conforme disse Fowler em sua obra, a duplicação de código leva a soluções menos elegantes, tornando o código mais propenso a erros e a manutenção mais trabalhosa.

Outro fator que compromete a elegância do código é a **Complexidade Desnecessária (Complexity)** quando há uma lógica muito complexa que poderia ser resolvida de maneira mais simples, isso torna o código mais difícil de entender e depurar.

Isso contraria um dos princípios fundamentais de um bom design de software, que é buscar soluções diretas e objetivas. Como Fowler aponta, a simplicidade e a elegância devem guiar o desenvolvimento para evitar sistemas desnecessariamente complicados.

2.4. Boas interfaces:

2.4.1. **Definição:** Interfaces bem definidas e intuitivas que permitem diferentes partes do sistema interagirem de forma clara, sem se preocupar com detalhes internos.

2.4.2. **Maus-cheiros relacionados:** A **Interface Confusa (Primitive Obsession)**, ocorre quando há um uso excessivo de tipos primitivos, como inteiros ou strings, em vez de tipos mais abstratos e representativos. Fowler afirma em sua obra que essa prática pode dificultar a compreensão do código, pois os dados não possuem significado explícito, tornando as interfaces menos intuitivas e propensas a erros.

Outro problema comum é o **Parâmetro de Função Excessivo (Long Parameter List)** que são funções que exigem muitos parâmetros que podem ser difíceis de entender e utilizar corretamente. Fowler sugere que, em vez de passar uma longa lista de parâmetros, é melhor agrupá-los em objetos ou estruturas que representem um conceito único. Isso melhora a legibilidade e torna as interfaces mais organizadas.

2.5. Extensibilidade:

2.5.1. **Definição:** Extensibilidade é a facilidade com que podemos adicionar novas funcionalidades ou modificar o sistema sem bagunçar o que já está funcionando. Ou seja, se o sistema é extensível, fica fácil acrescentar coisas novas sem precisar mudar o código todo.

2.5.2. **Maus-cheiros relacionados:** A falta de extensibilidade é frequentemente resultado do **Objeto Deus** (**God Object)**. Segundo o Fowler, esse mau cheiro é quando uma única classe ou objeto assume muitas responsabilidades. O problema é que, quando uma classe faz de tudo, qualquer modificação ou adição nova exige mexer em várias partes do código, o que torna as mudanças mais arriscadas. O **Inveja de Funcionalidade** (**Feature Envy)**, outro mau cheiro mencionado por Fowler, ocorre quando uma classe está sempre "olhando" demais para outra, acessando seus dados e métodos. Isso indica que o código não está bem organizado, dificultando alterações e tornando a manutenção mais difícil, pois os componentes ficam interligados demais.

2.6. Evitar Duplicação:

2.6.1. **Definição:** Evitar duplicação tem como definição não escrever o mesmo código mais de uma vez. Quando o código se repete, fica mais difícil de manter, porque se precisar mudar algo, temos que alterar em vários lugares.

2.6.2. **Maus-cheiros relacionados:** O **Código Duplicado** (**Duplicate Code)**, como Fowler destaca, é um dos maus cheiros mais comuns. Quando o código é copiado e colado em várias partes do sistema, qualquer modificação precisa ser feita em todos os lugares, o que aumenta a complexidade e o risco de erros. Outro problema que se relaciona com a duplicação é o **Código Morto (Dead Code)**, que é o código que está lá, mas nunca é executado. Segundo Fowler, isso só aumenta a confusão no sistema, pois deixa ele mais difícil de entender e de manter, mesmo sem trazer valor real ao projeto.

2.7. Portabilidade:

2.7.1. **Definição:** É a capacidade do código ser facilmente mudado para plataformas ou ambientes sem que mudanças significativas sejam necessárias.

2.7.2. **Maus-cheiros relacionados:** Um dos maus cheiros, como um obstáculo para a portabilidade é o **Código Específico da Plataforma (Platform-specific code)** que acontece quando o código depende fortemente de uma plataforma ou sistema operacional específico. O problema é que se você tentar rodar o código em outro ambiente ele pode não funcionar. Fowler explica que isso torna o código altamente dependente de um único contexto, o que restringe sua flexibilidade. Além disso, **Dependência de Ambiente (Environmental Dependency)** é outro mau cheiro que impede a portabilidade. Quando o código depende de configurações específicas do ambiente, como versões de bibliotecas ou configurações de hardware, ele perde a capacidade de ser facilmente ajustado para outros contextos.

2.8. Código deve ser idiomático e bem documentado:

2.8.1. **Definição:** Código idiomático quer dizer que um código que segue as convenções e práticas comuns da linguagem ou plataforma utilizada, ou seja, deve parecer “natural” para desenvolvedores experientes naquela tecnologia. Já um código bem documentado significa que ele deve ser autoexplicativo sempre que possível e com uma documentação robusta e atualizada.

2.8.2. **Maus-cheiros relacionados:** dentre os maus-cheiros relacionados à falta de um código idiomático, pode-se destacar o uso inapropriado de recursos da linguagem e a inconsistência de estilo definido por Fowler como **Código Não Idiomático (Lazy Class / Speculative Generality)**. Tal como **Nome Mal Escolhido (Inconsistent Naming)** conforme Fowler destaca, se o código não segue convenções de nomenclatura adequadas ou utiliza nomes inconsistentes, ele se torna difícil de entender, contrariando o princípio de código idiomático. Já os maus-cheiros relacionados à documentação fraca ou problemática pode-se destacar a documentação do óbvio, documentação excessiva e comentários obsoletos ou mentirosos **Comentários Excessivos (Comments)**. Fowler também cita os **Códigos Mortos (Dead Code)** que são trechos de código que não são mais utilizados, mas permanecem na base, podendo confundir desenvolvedores e comprometer a clareza do sistema. Um código bem documentado deve ser mantido atualizado

3. 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.

3.1. **Mau-cheiro identificado: <u>Comentários Excessivos (Comments):</u>** As classes contêm comentários que servem para gerar documentação explicando o que cada parte do código faz, mas há trechos com comentários desnecessários. Por exemplo, no arquivo IRPF.java, nas linhas 287 a 313, há um comentário referente a um código antigo que não é mais utilizado. Da mesma forma, na linha 271 do mesmo arquivo, há um comentário que também já não é necessário.

```
287    /**
288     * Obtem o valor do imposto de cada faixa
289     * @return Lista de valores referente a cada faixa
290     */
291    // public float[] getImpostoPorFaixa() {
292    //  float baseCalculo = getBaseCalculo();
293
294    //      float[] valoresTributados = new float[aliquotas.length];
295
296    //      float remanescente = baseCalculo - limites[0];
297    //      float limiteAnterior = limites[0];
298
299    //      for (int i = 1; i < limites.length; i++) {
300    //          if (remanescente > 0) {
301    //              float faixa = Math.min(remanescente, limites[i] - limiteAnterior);
302    //              valoresTributados[i] = faixa * aliquotas[i];
303    //              remanescente -= faixa;
304    //              limiteAnterior = limites[i];
305    //          }
306    //      }
307
308    //      if (remanescente > 0) {
309    //          valoresTributados[valoresTributados.length - 1] += remanescente * aliquotas[aliquotas.length - 1];
310    //      }
311
312    //      return valoresTributados;
313      // }
314
315
```

Imagem 1: Classe IRPF.java linha 287 a 313

3.1.1. **Os princípios sendo violados são:** Simplicidade, se é necessário comentários extensos para que o código seja entendido, então ele não cumpre com o princípio de simplicidade que exige um código escrito de forma clara e direta, além de poluir visualmente o script.

3.1.2. **As operações de refatorações identificadas pelo grupo:** Remover os comentários no arquivo IRPF.java, nas linhas 287 a 313, e na linha 271. Pois não são úteis, ou seja, não servem para a documentação do código.

3.2. Mau-cheiro identificado: <u>Lista diferentes para armazenar atributos de objetos (Long Parameter List):</u> Algumas listas simples poderiam ser substituídas por listas de objetos o que facilitaria a busca, atualização e criação de novos objetos. Com listas separadas é necessário percorrer cada lista individualmente para adicionar, atualizar, remover.

```
201
202     /**
203      * Metodo para cadastrar deduções integrais para o contribuinte. Para cada
204      * dedução é informado seu nome e valor.
205      * @param nome nome da deducao
206      * @param valorDeducao valor da deducao
207      */
208     public void cadastrarDeducaoIntegral(String nome, float valorDeducao) {
209         nomesDeducoes = adicionarDeducaoNome(nomesDeducoes, nome);
210         valoresDeducoes = adicionarDeducaoValor(valoresDeducoes, valorDeducao);
211     }
212
213     private String[] adicionarDeducaoNome(String[] nomesDeducoes, String nome) {
214         String[] temp = new String[nomesDeducoes.length + 1];
215         for (int i = 0; i < nomesDeducoes.length; i++) {
216             temp[i] = nomesDeducoes[i];
217         }
218         temp[nomesDeducoes.length] = nome;
219         return temp;
220     }
221
222     private float[] adicionarDeducaoValor(float[] valoresDeducoes, float valorDeducao) {
223             float[] temp = new float[valoresDeducoes.length + 1];
224             for (int i = 0; i < valoresDeducoes.length; i++) {
225                     temp[i] = valoresDeducoes[i];
226             }
227             temp[valoresDeducoes.length] = valorDeducao;
228             return temp;
229     }
230
```

Imagem 2: Métodos semelhantes que poderiam ser refatorados se implementado List<Obj>

3.2.1. **Os princípio sendo violados são:** Elegância e Evitar Duplicação, uma vez que para qualquer alteração é necessário, replicá-la para as demais listas que são relacionadas, isso gera métodos com o corpo semelhante e aumenta a dificuldade para futura manutenção

3.2.2. **As operações de refatorações identificadas pelo grupo:** Poderia ser possível utilizar uma List<Obj> em que Obj teria como atributo cada valor. Para buscar, atualizar, remover e demais ações, seria apenas 1 método de busca com base em um atributo, para as outras ações (atualizar, remover, adicionar), os demais métodos poderiam chamar o mesmo método de busca e trabalhar com o item.

3.3. **Mau-cheiro identificado: <u>Arquitetura má definida (Ambiguous Layering ou Camadas Ambíguas):</u>** Todas as classes estão em um mesmo nível sem diferenciação de classes principais e utilitárias, muito menos separação entre interfaces e classes.

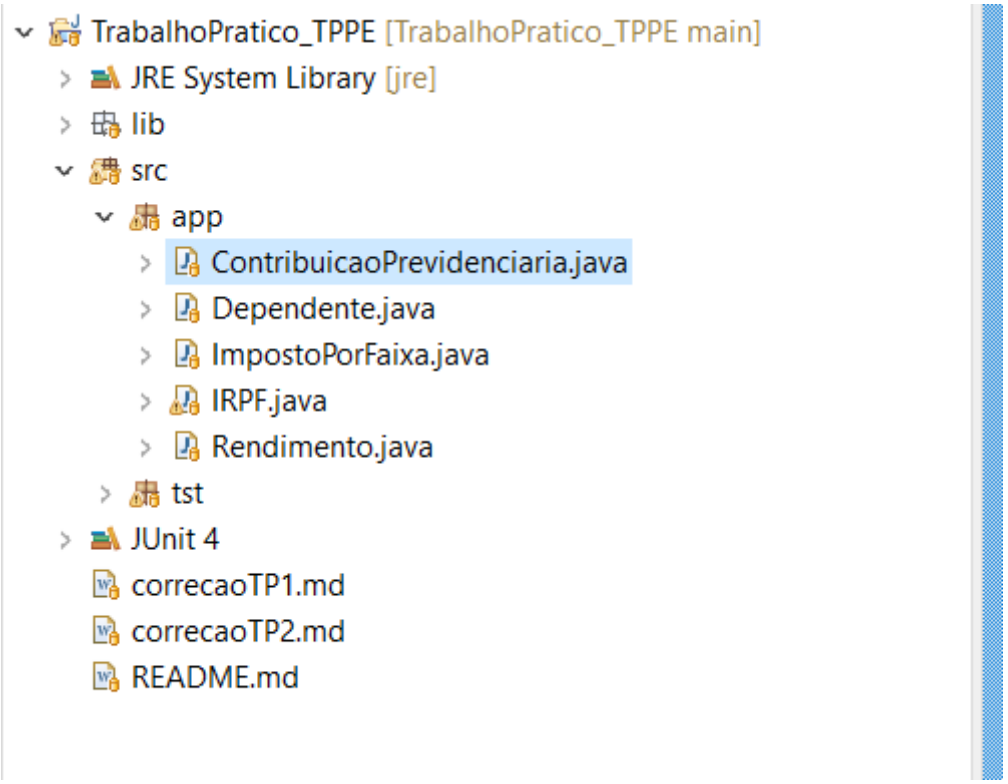
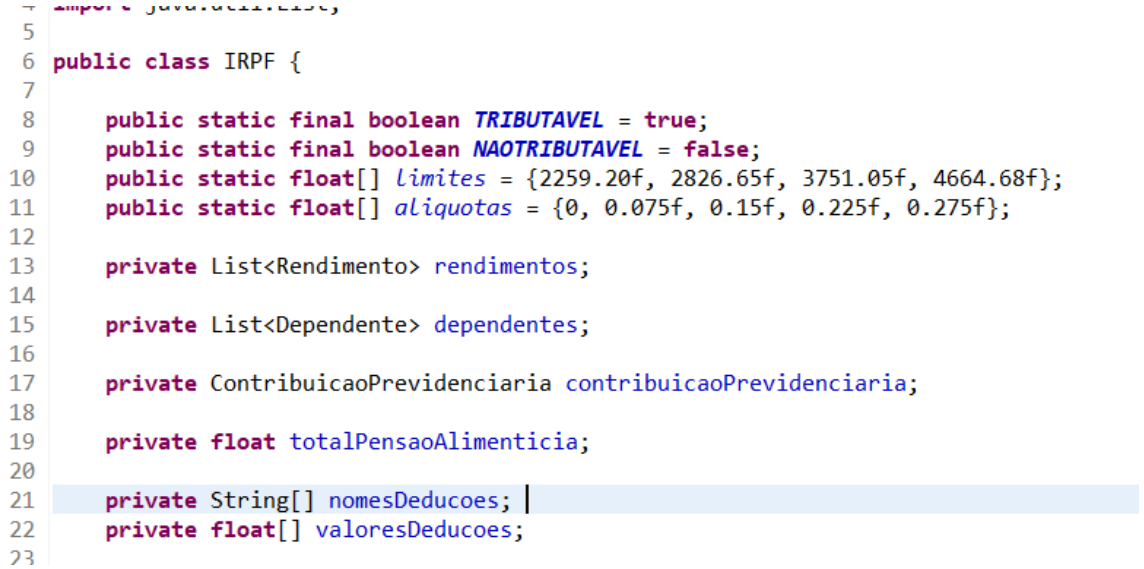


Imagem 3: Arquitetura inicial do projeto e hierarquia de pastas

3.3.1. **Os princípios sendo violados são:** Extensibilidade, sem uma arquitetura definida o projeto fica muito propenso a surgir classes com várias funcionalidades concentradas ou classes que alteram constantemente o atributo de outras classes.

3.3.2. **As operações de refatorações identificadas pelo grupo:** Seria possível dividir as classes em módulos mais específicos como “service”, “utils”, “data”,”infra” e entre outros

3.4. **<u>Uso extensivo de tipos primitivos:</u>** Muitos atributos como float[] e String[]



```
5
6  public class IRPF {
7
8      public static final boolean TRIBUTAVEL = true;
9      public static final boolean NAOTRIBUTAVEL = false;
10     public static float[] limites = {2259.20f, 2826.65f, 3751.05f, 4664.68f};
11     public static float[] aliquotas = {0, 0.075f, 0.15f, 0.225f, 0.275f};
12
13     private List<Rendimento> rendimentos;
14
15     private List<Dependente> dependentes;
16
17     private ContribuicaoPrevidenciaria contribuicaoPrevidenciaria;
18
19     private float totalPensaoAlimenticia;
20
21     private String[] nomesDeducoes;
22     private float[] valoresDeducoes;
23
```

Imagem 4: Classe com uso excessivo de tipos primitivos

3.4.1. **Os princípios sendo violados são:** Boa Interface, deve-se evitar o uso excessivo de tipos primitivos, pois eles são poucos representativos e menos descritivos.

3.4.2. **As operações de refatorações identificadas pelo grupo:** Poderia substituir os tipos float[] e String[] por uma List<Obj> onde Obj seria um objeto mais representativo e abstrato para cada caso específico.

4. Conclusão

Garantir um bom projeto de código é um processo contínuo que exige atenção à simplicidade, modularidade, elegância e outros princípios fundamentais (Goodliffe, 2006). A codificação, longe de ser apenas uma etapa de implementação, é uma atividade essencial de projeto, na qual o software é constantemente refinado para se manter coerente e sustentável.

Os maus-cheiros de código identificados por Fowler demonstram como desvios desses princípios podem comprometer a qualidade do software, tornando-o mais difícil de entender, modificar e estender. No caso analisado, a presença de comentários excessivos evidencia a violação da clareza e simplicidade do código, tornando-o menos legível e potencialmente confuso.

Logo, compreender e aplicar princípios de bom projeto, aliando-os a técnicas de refatoração, é fundamental para garantir a evolução saudável de um software. A adoção dessas práticas não apenas melhora a manutenibilidade do código, mas também reduz sua complexidade, tornando o sistema mais robusto e preparado para mudanças futuras.

5. Referência Bibliográfica

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.