Entrega 3

Técnicas de Programação em Plataformas Emergentes

Integrantes do grupo:

Nome	Matrícula
Diógenes Dantas Lélis Júnior	19/0105267
Felipe Candido de Moura	20/0030469
Matheus Pimentel Leal	15/0141629
Francisco Mizael Santos da Silva	18/0113321

O objetivo dessa entrega é realizar a exposição e explicação dos maus cheiros de código ainda presentes na entrega 2 realizada pelo grupo, tendo como guia os princípios de um bom projeto de código.

Os princípios que um bom projeto de código possui são: simplicidade, elegância, modularidade, boas interfaces, extensibilidade, evitar duplicações, portabilidade e um código idiomático e bem documentado. A estrutura deste trabalho está dividida em duas etapas: a explicação e conexão de cada um dos princípios de bom projeto de código e por fim a análise do código entregue pelo grupo de trabalho na entrega 2.

Pergunta 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.

Simplicidade

O princípio da simplicidade consiste em evitar complexidade desnecessária, tornando mais fácil a leitura, manutenção e entendimento do código. Soluções diretas e sem sobrecarga facilitam a depuração e evolução do sistema.

Alguns dos mau cheiros de código que indicam que algo pode estar ferindo o princípio da simplicidade são:

- *função longa demais:* Quanto mais longa uma função, mais difícil é de se entender o código escrito, assim ferindo a simplicidade;
- *lista longa de parâmetros:*Como dito por Martin Fowler em seu livro, longas listas de parâmetros são, de modo geral, confusas;
- *aglomerados de dados:* tendo em vista que os dados em aplicações tendem a estar juntos, um grande aglomerado de dados pode fazer com que a simplicidade do código produzido seja comprometida, e, utilizando refatorações como a extração de classe, podemos simplificar chamadas a métodos;

- *obsessão primitiva:* Muitos programadores apresentam resistência à criação de seus próprios tipos fundamentais, fazendo com que sejam utilizados muitos tipos primitivos para variáveis segundo Martin Fowler. Isso pode fazer com que grandes aglomerados de dados surjam e diminuam a simplicidade do código escrito;
- *expressões switch-case repetitivas:* O problema se apresenta na manutenção, fazendo com que em cada vez que seja adicionada uma condicional, temos que encontrar todas as expressões *switch-case* para atualizá-las.
- *generalidade especulativa:* Adicionar trechos de código específicos demais, que geralmente surgem de frases como “Acredito que precisaremos ter essa habilidade algum dia”, fazem com que o código por sua vez se torne difícil de entender e de manter;
- *campos temporários:* Em classes onde um certo campo é usado somente em circunstâncias especiais, a leitura do código fica prejudicada, visto que esperamos que uma classe necessite de todos os seus campos;
- *classe muito grande:* Quando uma classe procura fazer muito, ela geralmente apresenta muitos campos e também trechos de código duplicados, o que pode indicar que a simplicidade está comprometida; e
- código comentado: Códigos muito comentados indicam, certamente, uma dificuldade de compreensão sobre o que está sendo feito, o que fere o princípio da simplicidade.

Elegância

O princípio da elegância consiste em combinar clareza e eficiência, expressando a solução de forma natural e direta. Geralmente, envolve boas abstrações e evita redundâncias.

Alguns dos mau cheiros de código que indicam que algo pode estar ferindo o princípio da elegância são:

- *código duplicado:* A repetição desnecessária de trechos de código reduz a clareza e aumenta o esforço de manutenção. Além de gerar redundâncias no código, por exemplo: a mesma expressão pode estar em dois métodos da mesma classe
- *função longa:* Métodos extensos tornam o código difícil de entender e modificar. Um código elegante deve ser modular, dividindo métodos longos em funções menores e bem nomeadas, que expressam claramente sua responsabilidade.
- *cadeias de mensagem:* Quando um objeto acessa um método e outro objeto, que por sua vez acessa outro método, e assim por diante, o código se torna frágil e de difícil compreensão. A elegância exige a redução dessas cadeias por meio de encapsulamento adequado.
- *cirurgia com rifle:* quando uma pequena mudança exige modificações em múltiplas partes do código, a estrutura perde coesão. Uma solução elegante concentra as mudanças em um único local, utilizando boas abstrações e encapsulamentos.

Modularidade

O princípio da modularidade tem como objetivo separar o código em módulos ou componentes independentes, cada um com uma função clara e bem definida. Desta maneira o sistema se torna mais organizado, limpo e legível. Um bom indicativo de maus cheiros de código que seguem ao desencontro deste princípio são:

- *código duplicado*: A mesma estrutura de código repetida diversas vezes no sistema é um forte indicativo que este trecho de código pode ser abstraído e modularizado, reutilizado devidamente esta estrutura a fim de evitar esta duplicação e permitir futuras modificações de maneira atômica.
- *função longa*: Funções longas indicam que possivelmente ela está realizando mais de uma função atribuída, utilizar abordagens como *Extrair função,* garante uma maior modularidade ao código e reduz complexidade.
- inveja de recurso: Um indicativo de uma má modularização ocorre quando uma função dentro de um módulo se comunica constantemente com outro, isso pode indicar que aquela função se encontra em um módulo possivelmente errado. Pode ser utilizado uma abordagem de *Mover função* para solucionar este problema.
- *agrupamento de dados:* Dados que sempre são tratados juntos, podem ser facilmente abstraídos em um tipo de estrutura modulada que favorece sua manipulação e interação com outras partes do código, se tornando mais claros e legíveis.
- *generalidade especulativa*: Pode ser interpretado como um excesso de funcionalidades desnecessárias ou que cobre casos muito específicos, uma grande quantidade de módulos especulativos no sistema, sugere uma alta complexidade ciclomática e pode levar a problemas de otimização.
- *intermediário*: Uma alta delegação de tarefas a outros módulos em cascata, sugere uma alta modularização necessária, e por consequência um alto acoplamento, uma maneira de resolver este problema e fazendo uso da abordagem Remover intermediário.
- *classe grande*: Assim como em função grande, a classe grande contém o mesmo problema, se a classe esta grande, existe um indício que ela realiza mais do que sua atribuição, sendo possível extrair classes a partir dela, desta maneira modularizando mais o código e tornado mais coeso.

Boas interfaces

A criação de boas interfaces consiste em projetar bem as interfaces para que os módulos do sistema se comuniquem de forma clara, segura e eficiente, reduzindo acoplamento indesejado.

- *Trocas escusas*: Visando reduzir o número de troca de dados entre módulos e consequente reduzindo acoplamento, pode-se utilizar abordagens como mover campo, função ou mesmo classe que não faz sentido no escopo comportamental do módulo atual que se encontra. Desta maneira aprimorando a interface do módulo.
- *classes alternativas com interfaces diferentes*: em interfaces inadequadas onde se precise mudar com frequência a declaração de funções, ocasionalmente pode ocorrer duplicação de código, nestas situações pode ser necessário extrair uma superclasse para generalizar o comportamento de certas entidades.

Extensibilidade

A extensibilidade consiste em ter uma base de código modificável e expansível em que as modificações causem impacto mínimo nas partes existentes, permitindo novas funcionalidades sem grandes retrabalhos.

Uma possível indicação de maus cheiros de código que seguem ao desencontro deste princípio são:

- *switches repetidos:* O uso excessivo de estruturas switch/case pode indicar um código rígido e difícil de estender. Isso ocorre porque, sempre que um novo comportamento precisa ser adicionado, o switch deve ser modificado, impactando diretamente o código existente e aumentando o risco de erros.
- *generalidade especulativa:* Ocorre quando desenvolvedores criam abstrações genéricas sem necessidade real pode dificultar a manutenção e tornar a extensão do código mais complicada do que o necessário.
- *cirurgia com rifle:* Quando uma pequena mudança exige alterações em vários arquivos ou classes, indicando um design rígido e difícil de expandir. Isso torna a evolução do sistema mais complexa e propensa a erros.
- *intermediário:* ocorre quando uma classe serve apenas como ponte para outra sem adicionar valor real, apenas repassando chamadas de métodos sem lógica adicional. Esse tipo de estrutura dificulta a extensibilidade do código, pois adiciona uma camada extra que pode complicar futuras modificações e torná-las mais custosas.

Evitar duplicação

O princípio que orienta a evitar a duplicação de código, pois aumenta a dificuldade de manutenção e a possibilidade de inconsistências no sistema.

Alguns dos maus cheiros de código que sugerem possíveis violações do princípio da elegância são:

- *código duplicado:* Quando um mesmo trecho de código aparece em vários lugares do sistema, exigindo mais trabalho nas alterações e tornando-as mais propensas a erros. O ideal é extrair o código duplicado para um método ou classe reutilizável.
- *classe grande:* Quando uma classes tem campos demais, pode ocorrer uma repetição de lógica dentro de seus métodos, aproximando-se cada vez mais de um código duplicado
- *agrupamento de dados:* Grupos de variáveis frequentemente aparecem juntos em diferentes partes do código, isso pode indicar que há lógica duplicada espelhada pelo sistema. Melhorar a coesão e a modularização ajuda a minimizar o problema

Portabilidade

A portabilidade de código é o princípio de bom projeto de código que consiste em produzir uma base de código portátil que pode ser facilmente executada em diferentes ambientes sem grandes modificações. Isso depende do uso de padrões bem definidos e abstrações adequadas.

Alguns dos mau cheiros de código que indicam que algo pode estar ferindo o princípio da portabilidade são:

- *inveja de recursos:* Quando uma classe ou módulo depende de outra classe de forma excessiva para acessar seus dados ou recursos, isso pode criar uma dependência excessiva entre as partes do código. Isso limita a capacidade do sistema de ser executado de forma independente em diferentes ambientes, já que essas dependências podem não ser facilmente resolvíveis ou compatíveis em todas as plataformas.

- *classe grande:* Classes que acumulam muitas responsabilidades podem ser difíceis de adaptar a diferentes plataformas, especialmente quando suas partes interdependem. Isso também complica a manutenção, pois mudanças em novos ambientes podem exigir ajustes em várias partes da mesma classe.
- *intermediário:* Camadas intermediárias desnecessárias, que não adicionam valor, podem tornar o código mais rígido e dificultar a adaptação a novos ambientes, aumentando o acoplamento.
- *agrupamento de dados:* Quando dados relacionados são dispersos por várias partes do código, sem um agrupamento lógico, o código se torna mais difícil de ajustar em novos contextos. Estruturas de dados não abstraídas dificultam a reutilização do código em diferentes plataformas e aumenta o risco de inconsistências ao transferir dados entre sistemas heterogêneos

Código deve ser idiomático e bem documentado

Por fim, o código deve ser idiomático e bem documentado para que as convenções e boas práticas da linguagem usada sejam seguidas, tornando-o mais legível para outros desenvolvedores, e a documentação adequada complementa essa clareza.

Alguns dos maus cheiros de código que sugerem possíveis violações no conceito de código idiomático e bem documento são:

- *cadeias de mensagens:* Isso ocorre quando se enxerga uma longa cadeia de chamadas ao método getThis, ou como uma sequência de atribuições a variáveis temporárias, podendo indicar um acoplamento excessivo e tornar o código difícil de entender. Seguir práticas idiomáticas da linguagem pode ajudar a evitar esse problema.
- *lista longa de parâmetros:* Métodos que exigem muitos parâmetros dificultam a leitura e o uso correto da função. Utilizar objetos de configuração ou builders pode tornar o código mais idiomático.
- *herança recusada:* Quando uma classe herda métodos e propriedades que não fazem sentido para ela, pode indicar uma modelagem inadequada e desrespeito às boas práticas da linguagem, como o uso de composição ao invés de herança.
- *generalidade especulativa:* Criar abstrações excessivas e genéricas sem necessidade real pode resultar em código mais complexo do que o necessário, contrariando as convenções idiomáticas da linguagem.

Pergunta 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 seguir estão listados os mau cheiros identificados, persistentes do trabalho prático 2:

- Um dos principais pontos de melhoria, seria na melhor modularização da classe IRPF, visto que essa ainda detém muitas responsabilidades consigo, como os métodos referentes a Dependente, que podem ser facilmente extraídos. O princípio violado é o de **Modularidade**, podendo ser resolvido utilizando a técnica de **Extrair Classe.**

- Outra possível refatoração, pode ser feita nos métodos *adicionaNomeDependente* e *adicionaParentescoDependente,* que são chamados pela função *cadastrarDependente* e possuem um código muito semelhante que poderia ser resumido em uma só entidade, uma classe *Dependente*, evitando duplicidade. O princípio violado é o **Evitar Duplicação**, podendo ser utilizadas as técnicas de **Deslocar Instrução** e **Extrair Classe.**

```
public void adicionaNomeDependente(String nome) {
        String[] temp = new String[nomesDependentes.length+1];
        for (int i = 0; i < nomesDependentes.length; i++) {
                temp[i] = nomesDependentes[i];
        }
        temp[nomesDependentes.length] = nome;
        nomesDependentes = temp;
}

/**
 * Método para realizar a adição do grau de parentesco no registro de parentescos de
 * dependentes
 * @param parentesco
 */
public void adicionaParentescoDependente(String parentesco) {
        String[] temp = new String[parentescosDependentes.length + 1];
        for(int i = 0; i < parentescosDependentes.length; i++) {
                temp[i] = parentescosDependentes[i];
        }
        temp[parentescosDependentes.length] = parentesco;
        parentescosDependentes = temp;
}
```

```
 */
public void cadastrarDependente(String nome, String parentesco) {
        adicionaNomeDependente(nome);
        adicionaParentescoDependente(parentesco);

        numDependentes++;
}
```

- O código da forma em que está escrito, por usar muitos *arrays* e funções de *get* e *set* diversas no mesmo contexto, muito por conta da falta de modularidade da classe IRPF, acaba por ser mais complexa de se ler e entender, sem necessidade. Por isso, poderia se utilizar mais classes, *ArrayLists*, entre outros para ajudar numa melhor leitura e entendimento do código. O princípio principal violado é o da **Simplicidade**, podendo ser resolvido utilizando **Extrair Função** e **Extrair Classe**, além de **Substituir primitivo por objeto.**

- O nome dos métodos podem ser melhorados também, já que exemplos como *getOutrasDeducoes, getDeducao* e *getTotalOutrasDeducoes* pode gerar confusão também. O princípio principal violados é o **Código deve ser idiomático e bem documentado**, podendo ser resolvido utilizando as técnicas de **Mudar declaração de função.**

```
public String getOutrasDeducoes(String nome) {
        for (String d : nomesDeducoes) {
                if (d.toLowerCase().contains(nome.toLowerCase()))
                        return d;
        }
        return null;
}

/**
 * Obtem o valor da deducao à partir de seu nome
 * @param nome nome da deducao para a qual se busca seu valor
 * @return valor da deducao
 */
public float getDeducao(String nome) {
        for (int i=0; i<nomesDeducoes.length; i++) {
                if (nomesDeducoes[i].toLowerCase().contains(nome.toLowerCase()))
                        return valoresDeducoes[i];
        }
        return 0;
}

/**
 * Obtem o valor total de todas as deduções que nao sao do tipo
 * contribuicoes previdenciarias ou por dependentes
 * @return valor total das outras deducoes
 */
public float getTotalOutrasDeducoes() {
        float soma = 0;
        for (float f : valoresDeducoes) {
                soma += f;
        }
        return soma;
}
```

- Como o código como um todo utiliza funções de cadastro de dados de tipos semelhantes em estruturas de tipos semelhantes, poderia ser criado um método de *Cadastro* que funcionasse de forma geral, para evitar ainda mais duplicações. O princípio violado é o **Evitar Duplicação**, podendo ser utilizada a técnica de **Extrair Função.**

- O código possui estruturas de cálculo envolvendo as variáveis muito semelhante, uma abordagem para melhorar a organização de código e atribuições, seria utilizar uma interface Calculadora que abstrai o comportamento de estruturas como *getOutrasDeducoes, getDeducao* e *getTotalOutrasDeducoes*. Desta maneira não viola o princípio de **Boas Interfaces**, a técnica utilizada para este processo é a **Extrair Classe** e **classes alternativas com interfaces diferentes**, neste caso caracterizado pela ausência de uma interface.

Referência

[1] FOWLER, M. **Refatoração – 2ª edição**. [s.l.] Novatec Editora, 2020.
[2] GOODLIFFE, P. **Code craft : the practice of writing excellent code**. San Francisco: No Starch Press, 2007.