TP3 - Boas Práticas de programação
Grupo 18
Ana Beatriz Norberto da Silva - 211041080
Bruno Valério Martins Bomfim - 211039297
Kauã Vinícius Ponte Aguiar - 211029399

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.

a) **Simplicidade:** Código simples e fácil de se entender e por consequência, manter, evitando complexidades desnecessárias e seguindo a regra do KISS(Keep it simple, Stupid). Podemos correlacionar com os Code Smells: Método longo, Classe grande, Instruções switch, lista de parâmetros longa,homem do meio,mudança divergente, cadeia de mensagens, código duplicado,dentre outros, podemos relacionar com muitos, pois, um código grande ou "prolixo" demais, pode levar o programador a não entender o que está na tela.
b) **Elegância:** Código esteticamente agradável e que resolve os problemas de maneira clara e eficiente, podemos correlacionar com o código morto, código duplicado, instruções switch, lista de parâmetros longa, legado recusado, comentários, generalidade especulativa.A elegância se relaciona os esses code smells pois normalmente eles tornam o código sujo e não agradável aos olhos de quem ler, além disso, é mais informação desnecessária na tela, oque pode gerar confusão.
c) **Modularidade:** Refere-se a divisão do código em partes pequenas, independentes e componentizadas (mudam em conjunto e realizam tarefas semelhantes). Podemos correlacionar os code Smells: Cirurgia com espingarda, agrupamento de dados, Obsessão primitiva, legado recusado,data clumps, mudança divergente. Todos esses code smells se relacionam com modularidade pois revelam que o código não está bem componentizado ou organizado.
d) **Boas interfaces:** Criação de interfaces intuitivas e fáceis de se utilizar, abstraindo complexidades internas e expondo apenas o necessário. Podemos relacionar com herança negada, Classes alternativas com diferentes interfaces, data clumps, hierarquias de herança paralelas. Todos os code smells se relacionam com interfaces e podem confundir o usuário na hora de herdá-la, causando, a longo prazo, outros code smells.
e) **Extensibilidade:** Capacidade do código ser facilmente expandido ou adaptado a novas features sem causar muita fricção no código. Deve seguir o princípio do aberto e fechado, no qual fala que as classes

estão fechadas para modificação e abertas para extensão. Podemos observar os code smells como: Cirurgia com sniper, mudança divergente, hierarquia de herança paralela, data clumps, long method, long class, long parameter list, entre outros. Podemos perceber que um código que não possui extensividade, a longo prazo vira uma bomba relógio pois como o código está em constante mudança, é necessário manter uma estrutura adaptável.

f) **Evitar duplicação:** Eliminar código redundante, permitindo a reutilização através das abstrações, segue o princípio Don't Repeat Yourself (DRY). Podemos evidenciar os Code Smells: cirurgia com sniper, código duplicado, mudança divergente. Esses code smells estão relacionados ao princípio pois ao mudar algo em um local do código, essa mudança não pode ser reaproveitada em outros locais.

g) **Portabilidade:** Capacidade do código ser executado em diferentes ambientes ou sistemas sem necessidade de alterações significativas. Um código portável deve estar livre de dependências específicas de um sistema ou ambiente e deve ser flexível o suficiente para funcionar em diferentes plataformas. Relaciona-se com os maus-cheiros como dependências inapropriadas e acoplamento excessivo. Por exemplo, um código que depende de bibliotecas ou funcionalidades específicas de uma plataforma pode ser considerado um code smell, pois compromete a portabilidade. Isso pode evoluir para problemas maiores à medida que o código precisa ser adaptado a novos ambientes.

h) **Código deve ser idiomático e bem documentado:** Escrever código que siga as convenções e práticas recomendadas para a linguagem usada, tornando-o claro e compreensível para outros desenvolvedores. A documentação adequada ajuda a manter a legibilidade e a compreensão do código. Relaciona-se com code smells como "Comentários Necessários", onde a necessidade excessiva de comentários pode indicar que o código não é claro ou está mal estruturado. Além disso, "Nomes Ruins" pode dificultar a compreensão, exigindo mais esforço para entender o código. Quando o código não segue padrões idiomáticos ou não é bem documentado, aumenta a probabilidade de erros e a dificuldade de manutenção, evoluindo para outros code smells como "Código Duplicado" ou "Método Longo".

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.

- **Taxes.java**

  Na classe Taxes, foi identificado um mau-cheiro relacionado à Large Class e Long Method, violando o princípio da Responsabilidade Única (SRP) e o princípio de Encapsulamento. A lógica de cálculo de impostos estava toda centralizada nessa classe, o que resultava em alto acoplamento. A refatoração recomendada seria a Extração de Classe, onde a lógica de cálculo de impostos seria movida para uma nova classe, como TaxCalculator, melhorando a coesão e alinhando o código ao SRP.

- **Sale::calculateFreight()**

  Na classe Sale, o método calculateFreight() apresentava mau-cheiro de Large Class, Long Method e Duplicated Code, violando os princípios de Responsabilidade Única (SRP), Encapsulamento, e o princípio Don't Repeat Yourself (DRY). A solução recomendada seria também a Extração de Classe, movendo a lógica de cálculo de frete para uma nova classe. Isso reduziria a duplicação de código e seguiria o SRP, melhorando a modularidade e a manutenibilidade do sistema.

- **Sale::finish()**
  O método finish() da classe Sale foi refatorado aplicando a técnica de Substituir Método por Objeto Método. Originalmente, esse método continha lógica complexa para calcular impostos e descontos, o que tornava o código difícil de entender e manter. Ao encapsular essa lógica em uma nova classe, como TaxCalculator, transformamos o método em um objeto que pode ser instanciado e reutilizado. Isso isola a lógica de cálculo, facilitando a manutenção e a extensão do código, além de melhorar a coesão e aderir ao princípio de Responsabilidade Única (SRP).

Levando em consideração os princípios que mais foram violados podemos identificar outras classes que precisariam de refatoração:

1. Single Responsibility Principle (SRP)

- **<u>Violações:</u>** As classes **Client.java, Sale.java, Cart.java, ProductInfo.java, e Database.java** estão assumindo múltiplas responsabilidades, o que indica

que elas estão sobrecarregadas com funcionalidades que deveriam estar separadas.

- **Consequências:** Isso torna o código mais difícil de manter, testar e estender, pois uma mudança em uma responsabilidade pode afetar outras funcionalidades da classe, gerando efeitos colaterais inesperados.

- **Refatoração:** Extração de Classe e Divisão de Responsabilidades para garantir que cada classe tenha uma única responsabilidade bem definida.

2. Don't Repeat Yourself (DRY)

- **Violações:** A classe **PrimeClient.java** possui código duplicado em relação a outras classes de clientes, como SpecialClient.

- **Consequências:** A duplicação de código leva a problemas de manutenção, pois qualquer alteração no código duplicado precisa ser replicada em vários locais, aumentando a chance de erros e inconsistências.

- **Refatoração Recomendada:** Extração de Método ou Extração de Classe para consolidar o código repetido em um único lugar.

Referências

1 - Martin Fowler. Refactoring: Improving the design of Existing Code. Addison-Wesley Professional, 1999.

2 - Pete Goodliffe. Code Craft: The practice of Writing Excellent Code. No Starch Press, 2006.