Arthur Taylor de Jesus Popov - 190084642
Pedro Lucas Siqueira Fernandes - 190115564
Thiago Oliveira Cunha - 190096071

TP03 - TPPE - Grupo 09

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.

De acordo com a obra de Pete Goodliffe, podemos definir alguns pontos chave para um código com um bom design, sendo eles expostos e explicados a seguir, além disso estão relacionados com os Bad Smells da obra de Fowler.

- Simplicidade: A característica mais importante de um código com um bom design, evitando partes desnecessárias sendo fácil de entender e implementar, além de ser consistente e coerente. Código simples é o menor código possível que implementa a lógica necessária.

  Maus-cheiros relacionados:

  Complexidade Acidental: Código desnecessariamente complicado, sem motivo claro.

  Código Duplicado: A existência de código similar em diferentes partes do sistema que poderia ser simplificado.

- Elegância: Esta característica diz respeito ao código, a forma como é escrita evitando uma complexidade exacerbada, além de não procurar soluções mirabolantes para problemas que podem ser solucionados de forma mais simplista

  Maus-cheiros relacionados:

Código Morto: Código que não é mais usado, contribuindo para a falta de elegância.

Código Confuso: Quando o código é complicado e difícil de seguir.

- Modularidade:A modularidade diz respeito à divisão de funções complexas em módulos que se complementam para realizar funções mais complexas. Para que seja garantido uma modularidade de forma coesa é necessário que o exista uma alta coesão (modo como as funções funcionam juntamente para que o código funcione corretamente) e um baixo acoplamento (diz respeito a dependência entre os módulos, um código não pode ser totalmente desacoplado, mas um alto acoplamento é prejudicial para o funcionamento), se feito de forma correta, cada módulo pode ser dividido, desenvolvido e testado separadamente.

  Maus-cheiros relacionados:

  Grande Classe: Uma classe que faz mais do que deveria, violando a modularidade.

  Grande Método: Métodos que fazem mais do que deveriam, tornando a modularidade difícil.

- Boas interfaces: A modularização permite a separação dos problemas, cada um destes módulo define uma interface (a forma como o código é apresentado para fora do módulo) usualmente chamada de API é a forma como a funcionalidade é acessada, e a qualidade desta interface define a qualidade do módulo. Para criar boas interfaces podem ser seguidos os seguintes passos: *identificar o cliente e o que ele deseja fazer, identificar o fornecedor e o que ele é capaz de fazer, inferir o tipo de interface necessária e determinar a natureza da operação*. Todos estes passos devem estar alinhados para a garantir a qualidade da interface.

Particionamento: As interfaces possuem pontos de contato entre o cliente e o que foi implementado, códigos que possuem um bom design possuem papeis e responsabilidades bem definidas.

Abstração: Permite quem vê a API ignorar certos pontos da implementação, e focar em pontos importantes, basicamente realizando toda a implementação por trás da API e garantindo para o consumidor que estará certo, fazendo com que seja gasto tempo na implementação do consumidor.

Compressão: A agilidade como que uma interface representa operações complexas de forma simples.

Substituição: A possibilidade de substituir qualquer parte da interface por outra que possui um comportamento semelhante, e isso não irá afetar o funcionamento da interface.

Maus-cheiros relacionados:

Interface Preguiçosa: Interfaces que não oferecem o que deveriam, ou oferecem mais do que é necessário.

Excesso de Parâmetros: Métodos que aceitam muitos parâmetros, tornando a interface confusa.

- Extensibilidade: Este tópico diz respeito à possibilidade de expansão do código, de forma com que isso não acarrete uma engenharia exagerada que impactaria no funcionamento e no entendimento. Deixar pontos para extensibilidade em todo o código também pode ser prejudicial para o código, uma vez que que pode afetar a sua qualidade, deve ocorrer um balanço de até quando o código deve ser estendido.

  Maus-cheiros relacionados:

  Rigidez: Código que é difícil de modificar ou estender.

Fragilidade: Código que, quando alterado, causa problemas em áreas aparentemente não relacionadas.

- Evitar duplicação: A duplicação é extremamente abominada no design simples e elegante, basicamente código duplicado ocorre quando um programador copia e cola fragmentos de código sem entender completamente o propósito, se um código deve ser utilizado com pequenas mudanças ele provavelmente pode ser transformado em uma função e isso faz com que caso ocorra um erro seja de manutenção mais fácil por ter de alterar apenas um local.

  Mau-cheiro relacionado:

  Código Duplicado: Repetição do código em várias partes do sistema, que poderia ser refatorado para evitar duplicidade.

- Portabilidade: Códigos portáteis podem ser de grande valia em alguns casos, mas deve ser considerado se o código deve ser portável nos estágios iniciais do desenvolvimento, isso faz com que o escopo seja definido de forma correta e o projeto construído dentro do escopo proposto. Adicionar complexidade para portabilidade de um código que nunca será portado é algo desnecessário, mas por outro lado, se o código pode ser portado e reutilizado para economizar trabalho de desenvolver do zero, pode ser muito vantajoso, por isso a importância da definição dessa possibilidade nos estágios iniciais do projeto.

  Mau-cheiro relacionado:

  Dependências: Código fortemente acoplado a uma tecnologia ou ambiente específico, tornando a portabilidade difícil.

- Código deve ser idiomático e bem documentado: Um código idiomático diz respeito a seguir as normas definidas pela linguagem e por programadores que a utilizam, a fim de facilitar o entendimento por outros programadores que não escreveram o código em si, mas necessitam compreendê-lo para desenvolver ou realizar manutenções, isso juntamente com uma boa documentação de escolhas e de design do produto é essencial para facilitar o desenvolvimento e a manutenção dos códigos desenvolvidos.

  Maus-cheiros relacionados:

  Comentários Ruins: Comentários que são inúteis ou enganosos.

  Nomes Ruins: Variáveis, métodos ou classes com nomes que não seguem convenções ou não são descritivos.

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.

Após análise foram identificados os seguintes maus-cheiros no projeto:

- Code Duplication (Duplicação de Código): As classes ClienteEspecial e ClientePadrao contêm código duplicado no método getFrete. Isso viola o princípio DRY (Don't Repeat Yourself), o que pode dificultar a manutenção do código.
  Método de Refatoração: *Extract Method* (Extrair Método) ou *Template Method Pattern* (Padrão Template Method).

- Long Method (Método Longo): O método getFrete usa um switch com múltiplos case que tornam o código extenso e difícil de ler. Isso pode ser refatorado usando uma abordagem mais limpa, como um Map.

Método de Refatoração: *Replace Conditional with Polymorphism* (Substituir Condicional por Polimorfismo) ou *Replace Conditional with Map* (Substituir Condicional por Mapa).

- Unused Setters (Setters Não Utilizados): A classe Cliente possui métodos setNome, setEstado e setInterior que não são usados nas subclasses, sugerindo uma potencial violação do princípio YAGNI (You Aren't Gonna Need It).
  Método de Refatoração: *Remove Dead Code* (Remover Código Morto).

- Encapsulation Issues (Problemas de Encapsulamento): Na classe ClientePrime, o saldo de cashback é manipulado diretamente, o que pode comprometer a integridade dos dados. Seria melhor encapsular essa lógica para proteger contra valores negativos.
  Método de Refatoração: *Encapsulate Field* (Encapsular Campo) ou *Introduce Assertion* (Introduzir Asserção).

- Primitive Obsession (Obsessão por Primitivos): As classes CodigoItem, DescricaoItem, e PrecoItem encapsulam valores primitivos simples (int, String, float). Esses tipos de classes podem ser vistos como uma obsessão por encapsular primitivos sem fornecer comportamentos adicionais significativos.
  Método de Refatoração: *Replace Primitive with Object* (Substituir Primitivo por Objeto).

- Data Clumps (Agrupamento de Dados): As classes Produto e Venda agregam múltiplos objetos simples (CodigoItem, DescricaoItem, etc.), que frequentemente aparecem juntos. Isso sugere um possível data clump, onde esses dados poderiam ser agrupados de forma mais coesa.
  Método de Refatoração: *Introduce Parameter Object* (Introduzir Objeto de Parâmetro) ou *Introduce Class* (Introduzir Classe).

- Inappropriate Intimacy (Intimidade Inapropriada): A classe Venda depende diretamente da classe ProcessaVenda, o que pode indicar uma forte dependência entre as classes, dificultando a manutenção e o teste independente das classes.
  Método de Refatoração: *Move Method* (Mover Método) ou *Decouple Classes* (Desacoplar Classes).

- Feature Envy (Inveja de Recursos): A classe Venda parece ter comportamentos relacionados ao processamento de vendas, mas delega esses comportamentos para ProcessaVenda. Isso pode ser um indício de feature envy, onde o comportamento deveria residir diretamente na classe que contém os dados.
  Método de Refatoração: *Move Method* (Mover Método) ou *Extract Class* (Extrair Classe).

- Encapsulation Violation (Violação de Encapsulamento): A classe Produto expõe diretamente suas dependências internas (CodigoItem, DescricaoItem, PrecoItem) através de getters e setters, o que pode levar a uma violação de encapsulamento. Isso permite que outras partes do código modifiquem diretamente o estado interno do Produto, o que pode causar problemas de consistência.
  Método de Refatoração: *Encapsulate Collection* (Encapsular Coleção) ou *Hide Delegate* (Esconder Delegado).