Enunciado:

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

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.

1 - Princípios de bom projeto de código conforme descrito por Martin Fowler

1. Simplicidade

- **Definição**: Um código simples é aquele que realiza sua função de maneira direta e clara, sem complexidade desnecessária. A simplicidade facilita a compreensão, modificação e manutenção do código.
- **Maus-cheiros Relacionados**:
    - **Código Duplicado**: A duplicação de código é um dos principais inimigos da simplicidade, tornando o código mais complexo e difícil de manter.
    - **Método Longo**: Métodos que fazem muitas coisas tornam o código complicado. Dividir métodos longos em métodos menores e mais simples ajuda a manter a simplicidade.
    - **Classe Grande**: Classes que fazem muitas coisas (além de sua responsabilidade principal) tornam o sistema complexo e difícil de entender. Refatorar essas classes em várias classes menores pode restaurar a simplicidade.

2. Elegância

- **Definição**: Elegância no código se refere à implementação de soluções que são não apenas corretas, mas também claras e eficientes. Um código elegante é geralmente conciso, sem ser confuso, e resolve o problema de maneira natural.
- **Maus-cheiros Relacionados**:
    - **Código Rebuscado**: Quando a solução para um problema é mais complicada do que o necessário, isso pode comprometer a elegância do código. Refatorar para simplificar pode restaurar a elegância.
    - **Nomes Ambíguos**: Usar nomes de variáveis, métodos ou classes que não comunicam claramente sua função ou propósito torna o código menos elegante. Escolher nomes significativos ajuda a melhorar a clareza e a elegância.

3. Modularidade

- **Definição**: Modularidade refere-se à divisão do sistema em partes independentes (módulos), onde cada módulo é responsável por uma parte específica da funcionalidade do sistema. Isso facilita a manutenção, reutilização e evolução do código.
- **Maus-cheiros Relacionados**:
    - **Classes com Muitas Responsabilidades**: Quando uma classe assume muitas responsabilidades, ela compromete a modularidade do sistema. O princípio da responsabilidade única sugere que cada classe deve ter uma única responsabilidade claramente definida.
    - **Dependência Circular**: Quando módulos ou classes dependem uns dos outros de maneira circular, isso dificulta a modularidade e torna o sistema mais difícil de modificar.

4. Boas Interfaces

- **Definição**: Boas interfaces definem claramente como os módulos ou classes do sistema interagem entre si. Elas devem ser intuitivas, consistentes e minimizar o acoplamento entre diferentes partes do sistema.
- **Maus-cheiros Relacionados**:
    - **Interface Preguiçosa**: Quando uma interface ou classe não fornece todos os métodos necessários para que seja usada de forma eficiente, ou quando ela expõe métodos desnecessários, isso pode indicar um problema de projeto.
    - **Intermediário Necessário**: Quando um objeto é usado principalmente para delegar operações para outro objeto, a interface pode estar mal definida ou desnecessária. Refatorar para eliminar o intermediário melhora a clareza e a eficiência.

5. Extensibilidade

- **Definição**: Extensibilidade refere-se à capacidade de um sistema de ser estendido ou modificado com novas funcionalidades sem necessidade de grandes mudanças no código existente.
- **Maus-cheiros Relacionados**:
    - **Mudança Divergente**: Quando diferentes partes de um código precisam ser modificadas para implementar uma única mudança, o código não é extensível. Um bom design permite a extensão do código sem afetar outras partes do sistema.
    - **Inveja de Característica**: Quando uma classe usa mais métodos de outra classe do que seus próprios métodos, isso sugere que a funcionalidade deveria estar organizada de maneira diferente para facilitar a extensão.

6. Evitar Duplicação

- **Definição**: Evitar duplicação significa reduzir a repetição de código. Cada pedaço de conhecimento ou lógica deve existir em apenas um lugar no sistema. Isso facilita a manutenção e a atualização do sistema.
- **Maus-cheiros Relacionados**:
    - **Código Duplicado**: Este é o mau-cheiro mais diretamente relacionado à duplicação. Quando o mesmo código aparece em múltiplos lugares, qualquer alteração requer modificações em todos esses locais, aumentando a chance de erros.
    - **Falsos Acoplamentos**: Quando duas partes do código parecem estar relacionadas apenas porque repetem lógica semelhante, isso pode indicar uma oportunidade para abstrair a lógica comum em uma única classe ou método.

7. Portabilidade

- **Definição**: Portabilidade refere-se à capacidade do código de ser executado em diferentes ambientes ou plataformas com o mínimo de esforço.
- **Maus-cheiros Relacionados**:
    - **Dependência de Plataforma Específica**: Quando o código depende fortemente de uma plataforma específica, ele se torna menos portátil. Abstrair essas dependências em interfaces ou classes específicas para a plataforma pode melhorar a portabilidade.
    - **Hardcoded Environment Variables**: Usar variáveis ou caminhos codificados diretamente no código em vez de usar variáveis de ambiente ou configurações externas prejudica a portabilidade.

8. Código Idiomático e Bem Documentado

- **Definição**: Um código idiomático segue as convenções e padrões estabelecidos pela linguagem ou framework usado, tornando-o familiar e previsível para outros desenvolvedores. Além disso, deve ser bem documentado, para que qualquer desenvolvedor possa compreendê-lo rapidamente.
- **Maus-cheiros Relacionados**:
    - **Nomes Ambíguos**: Nomes que não seguem as convenções idiomáticas ou que são confusos comprometem a legibilidade e a manutenção.
    - **Comentário Desnecessário**: Comentários que explicam o óbvio ou que não são mantidos em sincronia com o código são prejudiciais. A documentação deve complementar o código e não explicar coisas que deveriam ser claras a partir do próprio código.

Obs: Essas definições e relações ajudam a entender como os princípios de bom projeto de código se alinham com os maus-cheiros de código descritos por Martin Fowler. Manter esses princípios em mente durante o desenvolvimento ajuda a garantir que o software seja bem projetado, fácil de manter, e resistente a mudanças.

2 - Identificação dos maus-cheiros no trabalho prático 2

As identificações dos maus-cheiros vão ser feitas e sinalizadas por arquivo começando por:

Arquivo -> Cliente.java

1. Comentários Excessivos e Inline

- Há um comentário inline (`// valor em centavos`) que explica o que o campo `saldoCashback` representa. Embora seja útil, se o nome da variável for mais descritivo, o comentário pode ser desnecessário.
- **Refatoração**: Renomear `saldoCashback` para algo mais descritivo como `saldoCashbackEmCentavos`, eliminando a necessidade de um comentário.

2. Múltiplos Campos Booleanos

- A classe `Cliente` possui múltiplos campos booleanos (`ehPrime`, `ehCapital`, `ehEspecial`). Isso pode indicar a presença do mau-cheiro **"Múltiplas Responsabilidades"**, onde a classe pode estar assumindo responsabilidades demais ou ter uma complexidade maior do que o necessário.
- **Refatoração**: Utilizar o **"Substituir Tipo Primitivo por Objeto"** para encapsular esses campos booleanos em uma classe que faça mais sentido no domínio da aplicação.

3. Número Mágico

- O campo `regiao` usa números mágicos para representar diferentes regiões (`0 - DF`, `1 - Centro-Oeste`, etc.). Isso é um mau-cheiro de código, pois torna o código menos legível e mais propenso a erros.
- **Refatoração**: Usar um **Enum** para representar as diferentes regiões, tornando o código mais legível e seguro.

Arquivo -> Impostos.java

1. Dependência com `ProdutoVenda`

- O método `calculaImpostos` depende de `ProdutoVenda` para calcular os impostos. Isso pode ser um indicativo de **"Acoplamento Excessivo"**, onde a classe `Impostos` está muito acoplada à classe `ProdutoVenda`.
- **Refatoração**: Uma refatoração possível seria usar o **"Introduzir Objeto Parâmetro"** para criar uma abstração entre `ProdutoVenda` e `Impostos`, reduzindo o acoplamento entre as duas classes.

2. Nomeação de Métodos e Parâmetros

- Os métodos e parâmetros na classe têm nomes genéricos (setIcms, setMunicipal, etc.). Embora funcionais, esses nomes poderiam ser mais descritivos para refletir melhor o que esses métodos fazem no contexto do domínio.
- **Refatoração**: Renomear os métodos para algo mais descritivo, como calcularIcmsParaRegiao ou definirTaxaMunicipal, para que o código seja mais autoexplicativo.

Arquivo -> Loja.java

1. Lista de Clientes, Produtos e Vendas

- A classe Loja mantém listas de clientes, produtos e vendas (ArrayList<Cliente>, ArrayList<Produto>, ArrayList<Venda>). Isso pode levar a **"Inchaço de Classe"**, onde a classe está lidando com muitos detalhes diretamente, o que pode resultar em código difícil de manter e entender.
- **Refatoração**: Aplicação do princípio de **"Segregação de Responsabilidade"**, criando classes específicas para gerenciar clientes, produtos e vendas, ou mesmo aplicando o padrão **"Repository"** para encapsular a lógica de armazenamento e recuperação de dados.

2. Método ehEspecial

- O método ehEspecial está realizando múltiplas operações em um único método, como filtragem de vendas, cálculo de valores e atualização do estado do cliente. Isso pode ser um indicativo de **"Método Longo"** ou **"Múltiplas Responsabilidades"**.
- **Refatoração**: Dividir este método em métodos menores e mais específicos, como filtrarVendasDoUltimoMes e calcularTotalCompras.

Arquivo -> Produto.java

1. Comentário Inline

- Há um comentário inline (// em centavos) para explicar que o valor é armazenado em centavos. Isso sugere que o nome da variável valor não é suficientemente descritivo.
- **Refatoração**: Renomear valor para algo mais específico, como valorEmCentavos, eliminando a necessidade de um comentário.

2. Falta de Validação

- Não há nenhuma validação para os valores passados para os atributos de Produto, como valor ou descricao. Isso pode levar a inconsistências ou erros inesperados.
- **Refatoração**: Adicionar validações no construtor e nos setters para garantir que os valores atribuídos a Produto sejam válidos.

Arquivo -> ProdutoVenda.java

1. Dependência Excessiva

- A classe ProdutoVenda tem uma forte dependência da classe Impostos, especialmente no método calculaValorParcial. Isso pode ser um sinal de **"Acoplamento Excessivo"**.
- **Refatoração**: Realizar a aplicação de **"Inversão de Dependência"** para reduzir o acoplamento entre ProdutoVenda e Impostos.

2. Método calculaValorParcial

- O método calculaValorParcial é responsável por calcular o valor parcial somando o valor do produto e os impostos. Isso pode indicar **"Múltiplas Responsabilidades"** dentro de um único método.
- **Refatoração**: Dividir este método em métodos menores e mais especializados.

Arquivo -> Start.java

1. Classe Start Como Ponto de Entrada com Muitas Responsabilidades

- A classe Start está sendo usada como ponto de entrada para a aplicação e contém métodos que lidam com a inicialização de clientes, produtos e a exibição de menus. Isso pode ser um indicativo de **"Muitas Responsabilidades"**.
- **Refatoração**: Dividir essa classe em várias classes menores que sejam responsáveis por diferentes aspectos da inicialização e controle da aplicação.

2. Scanner como Campo Estático

- O uso de Scanner como um campo estático pode ser um indicativo de **"Dependência Global"**, o que pode dificultar o teste e a manutenção do código.
- **Refatoração**: Passar o objeto Scanner como parâmetro para os métodos que necessitam dele.

Arquivo -> ValorFinalCalculator.java

1. Classe com Muitas Dependências

- A classe ValorFinalCalculator depende diretamente de várias propriedades de Venda, Cliente, e outras variáveis relacionadas. Isso pode ser um indicativo de **"Inchaço de Classe"** e **"Acoplamento Excessivo"**.
- **Refatoração**: Reduzir o acoplamento passando apenas os dados necessários para o cálculo como parâmetros em vez de depender diretamente das propriedades de Venda e Cliente.

2. Uso Extensivo de Condicionais

- O método compute faz uso extensivo de condicionais (if) para determinar o valor final da venda. Isso pode ser um sinal de **"Código de Ramificação Complexo"**.

- **Refatoração**: Aplicar o padrão **"Chain of Responsibility"** ou **"Strategy"** para encapsular as diferentes regras de cálculo em classes separadas, tornando o código mais modular e fácil de manter.

Arquivo -> Venda.java

1. Comentário Inline

- O campo metodoPagamento tem um comentário (// "PIX", "DEBITO", "CREDITO", "BOLETO") explicando os valores possíveis. Isso sugere que o nome da variável ou a implementação poderia ser mais descritiva.
- **Refatoração**: Usar um Enum para MetodoPagamento, o que eliminaria a necessidade de comentários e garante que apenas valores válidos fossem usados.

2. Acoplamento Excessivo

- A classe Venda está fortemente acoplada a ProdutoVenda e Cliente, realizando operações diretamente sobre esses objetos. Isso pode dificultar a manutenção e a evolução do código.
- **Refatoração**: Introduzir interfaces ou abstrações para reduzir o acoplamento e melhorar a modularidade.

3. Métodos Longos e Complexos

- Métodos como calculaValorFinal, calculaValorTotal, e calculaValorCashback podem se tornar longos e complexos, o que pode indicar **"Método Longo"**.
- **Refatoração**: Dividir esses métodos em métodos menores e mais focados, que lidam com partes específicas da lógica de cálculo.