1.Testes TDD

1.1Estrutura do Projeto

O projeto consiste em diversas classes que modelam o comportamento de usuários, compras, produtos e endereços. As principais classes e suas responsabilidades são:

- **Address:** Representa um endereço e contém métodos para identificar a região e se a cidade é capital.
- **Card**: Representa um cartão de crédito.
- **Buy:** Representa uma compra realizada por um usuário.
- **Product:** Representa um produto da loja.
- **User:** Representa um usuário da loja, podendo ser do tipo standard, prime ou special.
- **Store:** Gerencia as compras realizadas na loja.
- **TaxCalculator:** Calcula os impostos ICMS e municipais.
- **FreightCalculator:** Calcula o valor do frete com base na localização e tipo de usuário.

1.2 Testes Desenvolvidos

- **TestAdress:** A classe Address possui métodos para obter a região do endereço e verificar se a cidade é uma capital.
- **TestBuy:** A classe Buy representa uma compra realizada por um usuário. Os testes para esta classe foram desenvolvidos para validar o cálculo do total da compra, incluindo descontos, impostos e frete.
- **TestCard:** A classe Card representa um cartão de crédito. Os testes para esta classe verificam se o cartão foi criado corretamente com o número de cartão fornecido.
- **TestProductTax:** A classe Product possui métodos para calcular impostos (ICMS e municipais) sobre o produto. Os testes para esta classe verificam se esses cálculos estão corretos.
- **TestStore:** A classe Store gerencia as compras realizadas na loja. Os testes para esta classe verificam se as compras são adicionadas e removidas corretamente da lista de compras.
- **TestUser:** A classe User representa um usuário da loja. Os testes para esta classe verificam se os diferentes tipos de usuários (standard, prime e special) são criados corretamente e se suas propriedades são configuradas corretamente.

2. Princípios de Bom Projeto de Código e Maus-Cheiros de Código

2.1 Princípio de Simplicidade

- **Definição**: Um código simples é fácil de entender, testar e manter. Ele evita complexidades desnecessárias e foca em resolver o problema da maneira mais direta possível.

- **Maus-cheiros Relacionados**:
    - *Large Class (Classe Grande)*: Quando uma classe tem muitas responsabilidades, violando o princípio da simplicidade.
    - *Long Method (Método Longo)*: Um método que faz muitas coisas, o que pode indicar uma falta de simplicidade.
    - Código duplicado, também é mau-cheiro que viola a simplicidade. Ele aumenta a complexidade e dificulta a compreensão.

2.2 Princípio de Elegância

- **Definição**: Código elegante é aquele que resolve problemas de maneira clara e concisa, com um design estético e eficiente. Deste modo, um bom projeto de código busca um baixo acoplamento para garantir que mudanças em uma classe/módulo tenham um impacto mínimo em outras partes do sistema.
- **Maus-cheiros Relacionados**:
    - *Feature Envy (Inveja de Função)*: Quando um método em uma classe está muito interessado nos detalhes de outra classe, resultando em um acoplamento excessivo.
    - *Inappropriate Intimacy (Intimidade Inapropriada)*: Quando duas classes estão excessivamente acopladas, conhecendo detalhes internos uma da outra.
    - Classes ou métodos que fazem mais do que deveriam (ex: "God Object", "Feature Envy") são maus-cheiros que comprometem a elegância do código.

2.3 Princípio de Modularidade

- **Definição**: Código modular é dividido em partes menores e independentes que podem ser desenvolvidas, testadas e mantidas separadamente. Com o encapsulamento é possível esconder os detalhes internos de uma classe e expor apenas o que é necessário para o uso externo. Isso ajuda a proteger a integridade dos dados, facilita a manutenção e torna o código ainda mais modular.
- **Maus-cheiros Relacionados**:
    - *Data Clumps (Agrupamento de Dados)*: Quando certos grupos de variáveis aparecem repetidamente juntos, violando o princípio de modularidade.
    - *Data Class (Classe de Dados)*: Uma classe que só contém campos e métodos de acesso sem comportamentos próprios.

2.4 Boas Interfaces

- **Definição**: Interfaces claras e concisas permitem que os diferentes componentes do sistema interajam de maneira consistente e previsível. Uma boa interface deve expor apenas o necessário, mantendo detalhes internos ocultos, o que promove encapsulamento e coesão.
- **Maus-cheiros Relacionados**:
    - *Inappropriate Intimacy*: Este mau-cheiro ocorre quando classes ou módulos têm conhecimento excessivo dos detalhes internos de outros, violando o encapsulamento. Isso pode ser causado por interfaces mal projetadas que expõem detalhes desnecessários.
    - *Message Chains*: Ocorre quando uma classe chama uma série de métodos em sequência para obter o que precisa de outra classe. Isso indica que a

interface de um componente não está fornecendo um serviço claro e direto, forçando o chamador a navegar pelas dependências internas.

2.5 Extensibilidade

- **Definição**: Um sistema extensível é projetado para permitir que novas funcionalidades sejam adicionadas com o mínimo de impacto no código existente. Ele segue o princípio do “Open/Closed” (Aberto para extensão, fechado para modificação), permitindo a adição de novas características sem a necessidade de alterar o código base.
- **Maus-cheiros Relacionados**:
    - Shotgun Surgery: Quando uma pequena mudança requer modificações em muitas partes do código, o sistema se torna difícil de estender. Isso é um sintoma de falta de encapsulamento e de baixa coesão.
    - Divergent Change: Quando várias alterações diferentes precisam ser feitas em uma única classe, isso indica que a classe não está bem estruturada para suportar futuras extensões.
    - Parallel Inheritance Hierarchies: Este mau-cheiro ocorre quando, ao adicionar uma nova classe em uma hierarquia, é necessário criar uma classe correspondente em outra hierarquia. Isso dificulta a extensão do sistema, pois uma alteração simples exige mudanças em múltiplos lugares, indicando uma falta de independência entre as hierarquias.

2.6 Evitar Duplicação

- **Definição**: O princípio DRY (Don’t Repeat Yourself) prega que não devemos repetir lógica ou informações em mais de um lugar. Código duplicado gera inconsistências e aumenta o esforço de manutenção, pois qualquer mudança precisa ser replicada em vários locais.
- **Maus-cheiros Relacionados**:
    - Duplicated Code: A duplicação é o mau-cheiro mais direto associado a este princípio. Ela indica que há lógica redundante em diferentes partes do código, o que pode levar a inconsistências e bugs quando uma mudança é feita em um lugar, mas não em outros.

2.7 Portabilidade

- **Definição**: Portabilidade refere-se à capacidade do código de ser executado em diferentes ambientes (sistemas operacionais, plataformas, etc.) com pouca ou nenhuma modificação. Um código portátil é independente de características específicas do ambiente e é fácil de transferir e manter em diferentes contextos.
- **Maus-cheiros Relacionados**:

    - Inappropriate Intimacy: Dependências excessivas entre classes podem dificultar a portabilidade do código, especialmente se as classes estão fortemente acopladas a características específicas de um ambiente.
    - Data Clumps: Dados agrupados que sempre aparecem juntos em diferentes partes do código podem indicar falta de portabilidade, especialmente se esses grupos de dados forem específicos a um ambiente particular.

2.8 Código Idiomático e Bem Documentado

- **Definição**: Código idiomático segue as convenções e práticas recomendadas da linguagem de programação em uso. É fácil de entender para outros desenvolvedores familiarizados com a linguagem, o que minimiza a necessidade de explicações extensas. Código bem documentado complementa o entendimento, proporcionando clareza sem depender excessivamente de comentários.

- **Maus-cheiros Relacionados**:
    - Comments: Fowler menciona que o excesso de comentários pode indicar que o código não é claro ou idiomático. Se você precisa de muitos comentários para explicar o que está fazendo, provavelmente o código pode ser melhorado.
    - Long Method: Métodos longos podem ser difíceis de entender e documentar, resultando em código não idiomático e difícil de manter.

3. Identificação de Maus-Cheiros no Trabalho Prático 2

3.1 Maus-Cheiros Persistentes

- *Large Class*: Se o código que implementa as diferentes regras para clientes (padrão, especiais e prime) está concentrado em uma única classe, isso sugere uma classe com muitas responsabilidades, violando o princípio de coesão.
- *Feature Envy*: Se há métodos que dependem excessivamente dos detalhes internos de outras classes, como o cálculo de descontos e impostos em métodos que deveriam ser independentes, isso indica um acoplamento elevado.

3.2 Princípios Violados e Operações de Refatoração

- **Princípio Violado**: *Coesão*
    - **Operações de Refatoração**: Aplicar *Extract Class* para separar responsabilidades distintas em classes menores e mais coesas. Por exemplo, uma classe para calcular descontos, outra para impostos, e outra para frete.
- **Princípio Violado**: *Acoplamento*
    - **Operações de Refatoração**: Usar *Move Method* para mover métodos que estão muito acoplados a outras classes para as classes apropriadas. Por

exemplo, o cálculo do cashback pode ser movido para uma classe específica de cashback.

- **Princípio Violado**: *Encapsulamento*
    - **Operações de Refatoração**: Aplicar *Encapsulate Field* para proteger os dados e *Extract Method* para simplificar métodos longos que estão expondo muitos detalhes internos.

Com essas análises, o grupo pode focar em melhorar a estrutura do código, seguindo os princípios de bom design e considerando as operações de refatoração para remover os maus-cheiros identificados.