TPPE

Trabalho 03

Projeto de software

Alunos:

Thiago Borges
Bruno Seiji Kishibe
Jackes da Fonseca
Henrique Pucci

Unb - FGA

Questão 01 - 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:

Resposta:

Análise da Classe `ClientEspecial`

1. **Simplicidade**
    - **Definição:** O código deve ser o mais simples possível para realizar a tarefa sem adicionar complexidade desnecessária.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** O método `calcularDesconto` e o método `calcularDescontoFrete` têm lógicas distintas, mas o código para calcular descontos em cada método é específico e poderia ser consolidado em um único método genérico ou em uma classe de utilitário, dependendo do contexto.
        - **Método Longo:** Não há métodos excessivamente longos nesta classe, mas a simplicidade pode ser melhorada separando preocupações e criando métodos mais focados.
2. **Elegância**
    - **Definição:** O código deve ser escrito de uma forma que seja clara e bem estruturada.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** A lógica de desconto é repetida com diferentes cálculos para diferentes tipos (compra e frete). Isso pode ser feito de maneira mais elegante usando polimorfismo ou um padrão de estratégia.
        - **Condicional Complexo:** A condição para verificar o início do cartão poderia ser simplificada ou movida para uma classe de utilitário.
3. **Modularidade**
    - **Definição:** O código deve ser dividido em partes pequenas e independentes que realizam tarefas específicas.
    - **Relação com os maus-cheiros:**
        - **Método Longo e Função com Muitos Argumentos:** A classe `ClientEspecial` possui métodos que realizam diferentes cálculos de desconto, o que poderia ser modularizado em diferentes classes ou métodos especializados para facilitar a manutenção e a extensão.
        - **Alta Complexidade:** Cada método faz uma tarefa específica, mas a classe poderia ser mais modular se a lógica de cálculo de desconto fosse isolada em outra classe ou serviço.
4. **Boas Interfaces**
    - **Definição:** Interfaces devem ser claras e fáceis de usar.
    - **Relação com os maus-cheiros:**

        - **Interface Confusa:** A classe `ClientEspecial` expõe métodos específicos para desconto sem uma abstração clara de quais descontos podem ser aplicados. Ter uma interface mais clara e bem definida para tipos de desconto poderia melhorar a clareza.
        - **Nome de Método Inadequado:** `calcularDescontoFrete` e `calcularDesconto` são específicos e poderiam ser mais descritivos ou genéricos para indicar melhor o que está sendo calculado.

5. **Extensibilidade**
    - **Definição:** O código deve permitir a fácil adição de novas funcionalidades com o mínimo de modificação no código existente.
    - **Relação com os maus-cheiros:**
        - **Acoplamento:** A classe `ClientEspecial` está acoplada às regras específicas de cálculo de desconto. Para extensibilidade, seria melhor usar um padrão de estratégia onde diferentes tipos de desconto podem ser adicionados sem modificar a classe existente.
        - **Condicional Complexo:** Se novas regras de desconto precisarem ser adicionadas, a classe pode ficar complexa e difícil de manter.
6. **Evitar Duplicação**
    - **Definição:** O código não deve ter duplicação de lógica ou dados.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** Os cálculos de desconto para `calcularDesconto` e `calcularDescontoFrete` poderiam ser unificados ou melhor organizados para evitar duplicação de lógica.
        - **Duplicação de Lógica:** Se houver outras partes do código que calculam descontos de forma similar, isso deveria ser abstraído em uma única classe ou método.
7. **Portabilidade**
    - **Definição:** O código deve ser escrito de maneira que seja fácil de portar para diferentes plataformas ou ambientes.
    - **Relação com os maus-cheiros:**
        - **Dependências Específicas:** A classe `ClientEspecial` não mostra sinais claros de falta de portabilidade, mas usar strings específicas (como o início do cartão) pode ser uma má prática se isso mudar entre plataformas ou regiões.
8. **Código Idiomático e Bem Documentado**
    - **Definição:** O código deve seguir as práticas e estilos recomendados da linguagem e ser bem documentado.
    - **Relação com os maus-cheiros:**
        - **Código Não Idiomático:** A lógica de comparação do cartão é específica e poderia ser melhor documentada ou encapsulada.
        - **Falta de Documentação:** A classe não possui comentários ou documentação que expliquem o propósito e a lógica dos métodos. A falta de documentação pode tornar o código mais difícil de entender e manter.

Análise da Classe `Cliente`

1. **Simplicidade**
    - **Definição:** O código deve ser o mais simples possível para realizar a tarefa sem adicionar complexidade desnecessária.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** A classe `Cliente` é bastante simples e não possui duplicação de código. No entanto, se houver outras classes que também mantêm informações semelhantes, poderia haver oportunidades para refatoração e reutilização.
        - **Método Longo:** Os métodos são curtos e focados em obter ou definir valores, o que está em linha com a simplicidade.
2. **Elegância**
    - **Definição:** O código deve ser escrito de uma forma clara e bem estruturada.
    - **Relação com os maus-cheiros:**
        - **Código Redundante:** A classe é direta e clara, sem redundância evidente. A adição de um campo `tipo` pode ser vista como redundante se não houver uso específico para ele ou se não for extensível.
3. **Modularidade**
    - **Definição:** O código deve ser dividido em partes pequenas e independentes que realizam tarefas específicas.
    - **Relação com os maus-cheiros:**
        - **Alta Complexidade:** A classe `Cliente` é bem modularizada e não possui alta complexidade. No entanto, se o campo `tipo` é usado apenas para um caso específico, ele pode ser considerado uma modularidade adicional desnecessária se não houver mais funcionalidade relacionada.
4. **Boas Interfaces**
    - **Definição:** Interfaces devem ser claras e fáceis de usar.
    - **Relação com os maus-cheiros:**
        - **Interface Confusa:** A interface da classe `Cliente` é simples e clara. Contudo, se a classe `Cliente` for estendida com funcionalidades adicionais, pode haver necessidade de revisar a clareza da interface.
5. **Extensibilidade**
    - **Definição:** O código deve permitir a fácil adição de novas funcionalidades com o mínimo de modificação no código existente.
    - **Relação com os maus-cheiros:**

        - **Acoplamento:** A classe `Cliente` é relativamente fácil de estender. No entanto, se o campo `tipo` não for bem definido para diferentes tipos de cliente, pode ser difícil adicionar novos tipos sem modificar a classe existente.
6. **Evitar Duplicação**
    - **Definição:** O código não deve ter duplicação de lógica ou dados.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** Não há duplicação na classe `Cliente` propriamente dita. No entanto, é importante garantir que a lógica e os dados relacionados a `Cliente` não se repitam em outras partes do código.
7. **Portabilidade**
    - **Definição:** O código deve ser escrito de maneira que seja fácil de portar para diferentes plataformas ou ambientes.
    - **Relação com os maus-cheiros:**
        - **Dependências Específicas:** A classe `Cliente` não tem dependências específicas e é independente do ambiente. Portanto, é portável.
8. **Código Idiomático e Bem Documentado**
    - **Definição:** O código deve seguir as práticas e estilos recomendados da linguagem e ser bem documentado.
    - **Relação com os maus-cheiros:**
        - **Falta de Documentação:** A classe `Cliente` não possui comentários ou documentação. Adicionar documentação ajudaria a explicar a finalidade dos atributos e métodos, especialmente se `tipo` tiver um propósito específico.

Análise da Classe `ClientePrime`

1. **Simplicidade**
    - **Definição:** O código deve ser o mais simples possível para realizar a tarefa sem adicionar complexidade desnecessária.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** A classe `ClientePrime` não possui duplicação direta. No entanto, a lógica de crédito de cashback e a lógica de cálculo e crédito de cashback poderiam ser mais moduladas para evitar complexidade desnecessária.
        - **Método Longo:** O método `calcularECreditarCashback` faz duas coisas: calcula e credita o cashback. Pode ser mais claro e simples se esses dois passos forem separados em métodos distintos.
2. **Elegância**
    - **Definição:** O código deve ser escrito de uma forma clara e bem estruturada.

- **Relação com os maus-cheiros:**
  - **Código Redundante:** A lógica de cálculo de cashback é específica e está embutida em um método. Poderia ser mais elegante se essa lógica fosse encapsulada em uma classe separada ou um serviço, permitindo uma fácil alteração e reutilização.

3. **Modularidade**
   - **Definição:** O código deve ser dividido em partes pequenas e independentes que realizam tarefas específicas.
   - **Relação com os maus-cheiros:**
     - **Método Longo e Função com Muitos Argumentos:** O método `calcularECreditarCashback` realiza mais de uma tarefa (cálculo e crédito). Separar essas responsabilidades pode melhorar a modularidade.
     - **Alta Complexidade:** A classe pode ser mais modular se a lógica de cashback for delegada a uma classe ou serviço especializado.
4. **Boas Interfaces**
   - **Definição:** Interfaces devem ser claras e fáceis de usar.
   - **Relação com os maus-cheiros:**
     - **Interface Confusa:** A interface da classe `ClientePrime` pode ser mais clara se as responsabilidades estiverem melhor divididas. Por exemplo, separar o cálculo de cashback da ação de creditar cashback pode tornar a interface mais intuitiva.
     - **Nome de Método Inadequado:** `calcularECreditarCashback` é um nome longo e pode ser dividido em métodos mais específicos.
5. **Extensibilidade**
   - **Definição:** O código deve permitir a fácil adição de novas funcionalidades com o mínimo de modificação no código existente.
   - **Relação com os maus-cheiros:**
     - **Acoplamento:** A lógica de cashback está acoplada diretamente à classe `ClientePrime`. Seria mais extensível se essa lógica fosse abstraída em uma classe separada, permitindo a modificação e extensão sem alterar a classe `ClientePrime`.
6. **Evitar Duplicação**
   - **Definição:** O código não deve ter duplicação de lógica ou dados.
   - **Relação com os maus-cheiros:**
     - **Código Duplicado:** Não há duplicação clara na classe `ClientePrime`, mas a lógica de crédito de cashback poderia ser abstraída se for utilizada em outros contextos.
7. **Portabilidade**
   - **Definição:** O código deve ser escrito de maneira que seja fácil de portar para diferentes plataformas ou ambientes.
   - **Relação com os maus-cheiros:**

        - **Dependências Específicas:** A classe não tem dependências específicas que afetam sua portabilidade. A lógica de cashback é genérica e deve funcionar em diferentes ambientes.
8. **Código Idiomático e Bem Documentado**
    - **Definição:** O código deve seguir as práticas e estilos recomendados da linguagem e ser bem documentado.
    - **Relação com os maus-cheiros:**
        - **Falta de Documentação:** A classe `ClientePrime` não possui documentação. Comentários explicando o propósito dos métodos e a lógica por trás do cálculo de cashback melhorariam a compreensão e a manutenção.

Análise da Classe `NotaFiscal`

1. **Simplicidade**
    - **Definição:** O código deve ser o mais simples possível para realizar a tarefa sem adicionar complexidade desnecessária.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** A lógica para clientes do tipo `ClientePrime` é duplicada (verificação e uso de cashback). A verificação do tipo de cliente é feita duas vezes.
        - **Método Longo:** O método `calcularValorFinal` realiza várias operações e decisões, o que o torna um pouco longo e complexo. A lógica poderia ser simplificada.
2. **Elegância**
    - **Definição:** O código deve ser escrito de uma forma clara e bem estruturada.
    - **Relação com os maus-cheiros:**
        - **Código Redundante:** O código possui redundância ao verificar duas vezes se o cliente é uma instância de `ClientePrime`. Isso pode ser simplificado.
        - **Condicional Complexo:** A lógica de cálculo do valor final é complexa e pode ser dividida em métodos auxiliares para melhorar a clareza e a elegância.
3. **Modularidade**
    - **Definição:** O código deve ser dividido em partes pequenas e independentes que realizam tarefas específicas.
    - **Relação com os maus-cheiros:**
        - **Método Longo e Função com Muitos Argumentos:** O método `calcularValorFinal` faz muitos cálculos e contém lógica complexa. Pode ser dividido em métodos menores e mais específicos.

        - **Alta Complexidade:** A classe pode ser mais modular se a lógica para cálculo de impostos, valor final e cashback forem separadas em métodos ou classes distintas.
4. **Boas Interfaces**
    - **Definição:** Interfaces devem ser claras e fáceis de usar.
    - **Relação com os maus-cheiros:**
        - **Interface Confusa:** A interface da classe `NotaFiscal` pode ser mais clara se a lógica de cálculo for melhor dividida. Por exemplo, a verificação do tipo de cliente e o cálculo do cashback poderiam ser abstraídos.
        - **Nome de Método Inadequado:** O nome `calcularValorFinal` pode ser mais descritivo, e a lógica envolvida pode ser dividida em métodos com nomes mais claros.
5. **Extensibilidade**
    - **Definição:** O código deve permitir a fácil adição de novas funcionalidades com o mínimo de modificação no código existente.
    - **Relação com os maus-cheiros:**
        - **Acoplamento:** A classe `NotaFiscal` está acoplada à lógica específica de `ClientePrime`. Para permitir extensibilidade, seria melhor usar uma abordagem mais flexível para o cálculo de cashback, talvez usando um padrão de estratégia ou serviço.
6. **Evitar Duplicação**
    - **Definição:** O código não deve ter duplicação de lógica ou dados.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** Há duplicação na verificação e uso de `ClientePrime`. Esse código pode ser simplificado para evitar redundância.
7. **Portabilidade**
    - **Definição:** O código deve ser escrito de maneira que seja fácil de portar para diferentes plataformas ou ambientes.
    - **Relação com os maus-cheiros:**
        - **Dependências Específicas:** A classe `NotaFiscal` não possui dependências específicas que afetem sua portabilidade. No entanto, a lógica de impostos e cashback pode precisar de ajustes se a aplicação for adaptada para diferentes contextos.
8. **Código Idiomático e Bem Documentado**
    - **Definição:** O código deve seguir as práticas e estilos recomendados da linguagem e ser bem documentado.
    - **Relação com os maus-cheiros:**
        - **Falta de Documentação:** A classe `NotaFiscal` não possui documentação. Adicionar comentários explicativos e descritivos sobre a lógica dos cálculos e o propósito dos métodos ajudaria na manutenção e compreensão do código.

Análise da Classe `Produto`

1. **Simplicidade**
    - **Definição:** O código deve ser o mais simples possível para realizar a tarefa sem adicionar complexidade desnecessária.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** A classe `Produto` não tem duplicação de código em si. No entanto, a classe pode ser simplificada se alguns atributos não forem necessários ou se houver atributos desnecessários.
        - **Método Longo:** Não há métodos longos, mas a classe possui muitos atributos. Se a classe crescer, métodos complexos podem surgir.
2. **Elegância**
    - **Definição:** O código deve ser escrito de uma forma clara e bem estruturada.
    - **Relação com os maus-cheiros:**
        - **Código Redundante:** A presença de atributos como `valorVenda` e `preco` pode ser redundante se ambos representarem o mesmo conceito. É importante revisar se todos os atributos são necessários e se há redundância entre eles.
        - **Nome de Atributos:** Os nomes dos atributos são geralmente claros, mas a distinção entre `valorVenda` e `preco` não é evidente. Se ambos forem diferentes, isso deve ser bem documentado.
3. **Modularidade**
    - **Definição:** O código deve ser dividido em partes pequenas e independentes que realizam tarefas específicas.
    - **Relação com os maus-cheiros:**
        - **Alta Complexidade:** A classe `Produto` é relativamente simples, mas se a classe se expandir, pode se tornar complexa. Considerar a separação de responsabilidades, se necessário.
4. **Boas Interfaces**
    - **Definição:** Interfaces devem ser claras e fáceis de usar.
    - **Relação com os maus-cheiros:**
        - **Interface Confusa:** A interface da classe `Produto` é bastante clara. No entanto, se houver confusão sobre os atributos `valorVenda` e `preco`, é importante clarificar seu uso.
        - **Nome de Método Inadequado:** Não há métodos além dos getters, então não há problemas de nomenclatura em métodos.
5. **Extensibilidade**
    - **Definição:** O código deve permitir a fácil adição de novas funcionalidades com o mínimo de modificação no código existente.
    - **Relação com os maus-cheiros:**
        - **Acoplamento:** A classe `Produto` está bem encapsulada, mas a adição de novas funcionalidades (como descontos, promoções, etc.) pode exigir alterações na classe existente. Pode ser útil introduzir novas classes ou métodos para essas funcionalidades.
6. **Evitar Duplicação**
    - **Definição:** O código não deve ter duplicação de lógica ou dados.

    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** Não há duplicação direta na classe. No entanto, se `valorVenda` e `preco` são redundantes, isso deve ser resolvido para evitar duplicação de dados.
7. **Portabilidade**
    - **Definição:** O código deve ser escrito de maneira que seja fácil de portar para diferentes plataformas ou ambientes.
    - **Relação com os maus-cheiros:**
        - **Dependências Específicas:** A classe `Produto` não possui dependências específicas, o que a torna portável.
8. **Código Idiomático e Bem Documentado**
    - **Definição:** O código deve seguir as práticas e estilos recomendados da linguagem e ser bem documentado.
    - **Relação com os maus-cheiros:**
        - **Falta de Documentação:** A classe `Produto` não possui documentação. Comentários sobre a finalidade de cada atributo e as diferenças entre `valorVenda` e `preco` ajudariam na compreensão.

Análise da Classe `Venda`

1. **Simplicidade**
    - **Definição:** O código deve ser o mais simples possível para realizar a tarefa sem adicionar complexidade desnecessária.
    - **Relação com os maus-cheiros:**
        - **Método Longo:** O método `calcularValorTotal` é simples, mas o cálculo de valores e a criação de `NotaFiscal` no construtor podem adicionar complexidade. Idealmente, o construtor deve ser simples e delegar cálculos para métodos auxiliares.
2. **Elegância**
    - **Definição:** O código deve ser escrito de uma forma clara e bem estruturada.
    - **Relação com os maus-cheiros:**
        - **Código Redundante:** Há uma redundância ao chamar `notaFiscal.getValorTotal()` várias vezes para obter valores como frete, desconto e valor final, quando esses valores são diretamente calculados a partir de `NotaFiscal`.
        - **Nome de Métodos:** Os nomes dos métodos são claros, mas os métodos de `NotaFiscal` como `getValorTotal` são chamados de forma redundante para obter o valor do frete e do desconto.
3. **Modularidade**
    - **Definição:** O código deve ser dividido em partes pequenas e independentes que realizam tarefas específicas.

- **Relação com os maus-cheiros:**
  - **Alta Complexidade:** A classe `Venda` é relativamente simples, mas a criação da `NotaFiscal` e o cálculo do valor total poderiam ser mais bem modularizados para evitar lógica complexa no construtor.

4. **Boas Interfaces**
   - **Definição:** Interfaces devem ser claras e fáceis de usar.
   - **Relação com os maus-cheiros:**
     - **Interface Confusa:** O uso dos métodos `getValorFrete`, `getDesconto`, `getImpostoICMS`, e `getImpostoMunicipal` retorna `notaFiscal.getValorTotal()` em vez dos valores corretos. Esses métodos não são claros em seu propósito atual.
5. **Extensibilidade**
   - **Definição:** O código deve permitir a fácil adição de novas funcionalidades com o mínimo de modificação no código existente.
   - **Relação com os maus-cheiros:**
     - **Acoplamento:** A classe `Venda` está acoplada à classe `NotaFiscal`. Para permitir a extensibilidade, a lógica de criação e cálculos de nota fiscal pode ser separada.
6. **Evitar Duplicação**
   - **Definição:** O código não deve ter duplicação de lógica ou dados.
   - **Relação com os maus-cheiros:**
     - **Código Duplicado:** Há duplicação de chamadas a `notaFiscal.getValorTotal()`. Cada método deveria retornar seu valor específico sem depender da mesma chamada repetidamente.
7. **Portabilidade**
   - **Definição:** O código deve ser escrito de maneira que seja fácil de portar para diferentes plataformas ou ambientes.
   - **Relação com os maus-cheiros:**
     - **Dependências Específicas:** A classe `Venda` depende diretamente de `NotaFiscal` e de seus cálculos. A modularização e a separação de responsabilidades podem melhorar a portabilidade.
8. **Código Idiomático e Bem Documentado**
   - **Definição:** O código deve seguir as práticas e estilos recomendados da linguagem e ser bem documentado.
   - **Relação com os maus-cheiros:**
     - **Falta de Documentação:** A classe `Venda` não possui documentação. Comentários sobre o propósito dos métodos e as responsabilidades da classe ajudariam a tornar o código mais compreensível.

Análise da Classe `VendasService`

1. **Simplicidade**
    - **Definição:** O código deve ser o mais simples possível para realizar a tarefa sem adicionar complexidade desnecessária.
    - **Relação com os maus-cheiros:**
        - **Método Longo:** Ambos os métodos (`calcularTotalVendasUltimoMes` e `atualizarStatusCliente`) são relativamente simples, mas o método `calcularTotalVendasUltimoMes` poderia ser mais modular para melhorar a clareza.
2. **Elegância**
    - **Definição:** O código deve ser escrito de uma forma clara e bem estruturada.
    - **Relação com os maus-cheiros:**
        - **Código Redundante:** O cálculo do total de vendas é repetido dentro do método `atualizarStatusCliente`. Isso pode ser otimizado.
3. **Modularidade**
    - **Definição:** O código deve ser dividido em partes pequenas e independentes que realizam tarefas específicas.
    - **Relação com os maus-cheiros:**
        - **Alta Complexidade:** A classe `VendasService` não é muito complexa, mas poderia ser melhor modularizada. Por exemplo, o cálculo do total de vendas poderia ser isolado em um método separado que também lida com a lógica de data.
4. **Boas Interfaces**
    - **Definição:** Interfaces devem ser claras e fáceis de usar.
    - **Relação com os maus-cheiros:**
        - **Interface Confusa:** Os métodos têm interfaces claras, mas a lógica de comparação de datas e a atualização de status podem ser mais intuitivas se separadas.
5. **Extensibilidade**
    - **Definição:** O código deve permitir a fácil adição de novas funcionalidades com o mínimo de modificação no código existente.
    - **Relação com os maus-cheiros:**

        - **Acoplamento:** O código está fortemente acoplado ao modelo `Cliente` e `Venda`. A modularização e abstração adicional podem facilitar a extensão.
6. **Evitar Duplicação**
    - **Definição:** O código não deve ter duplicação de lógica ou dados.
    - **Relação com os maus-cheiros:**
        - **Código Duplicado:** O cálculo do total de vendas é repetido nos métodos. Pode ser reutilizado em métodos auxiliares.
7. **Portabilidade**
    - **Definição:** O código deve ser escrito de maneira que seja fácil de portar para diferentes plataformas ou ambientes.
    - **Relação com os maus-cheiros:**
        - **Dependências Específicas:** O código é independente de plataforma específica e deve ser portátil.
8. **Código Idiomático e Bem Documentado**
    - **Definição:** O código deve seguir as práticas e estilos recomendados da linguagem e ser bem documentado.
    - **Relação com os maus-cheiros:**
        - **Falta de Documentação:** A classe `VendasService` carece de documentação. Comentários sobre o propósito e a lógica dos métodos ajudariam na manutenção e compreensão.

Questão 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.

Resposta:

- Na classe ClientEspecial, os métodos calcularDesconto e calcularDescontoFrete estão duplicados. Para solucionar, deveriam ser extraídos em um mesmo mesmo método, adaptando a lógica do desconto para que pudesse ser reutilizado.

- Na classe ClientePrime, o método calcularECreditarCashback o método chama creditarCashback apenas para realizar a adição do cashback. Sendo recomendado o uso do *inline method*, uma vez que o método é adiciona uma complexidade maior do que se propõe a resolver. Por tanto seria feita a fusão dos métodos em um único.

- Na classe Produto, o atributo valorVenda e o método getValorVenda representam uma redundância com o atributo valorTotal e o método getValorFinal da classe NotaFiscal. Devendo ser utilizada a operação de substituição do algoritmo, sendo optado por manter o código presente na classe NotaFiscal.

- Na classe NotaFiscal, o método calcularValorFinal é muito longo e realiza muitas operações. Para sua refatorização seria utilizada a operação de extração do método, modularizando melhor o código em métodos menores e fazendo o chamado por meio deste método.