TÉCNICAS DE PROGRAMAÇÃO EM PLATAFORMAS EMERGENTES

Aluno: Júlio César Martins França

Matrícula: 190015721

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

1. **Simplicidade**

   Um projeto simples evita complexidades desnecessárias e foca em resolver o problema atual de forma direta e eficiente, sem se antecipar a requisitos futuros que possam nunca ocorrer. Um projeto simples é mais fácil de entender, implementar e modificar, o que facilita a manutenção e a evolução do software. Embora a simplicidade possa parecer óbvia, alcançar uma solução realmente simples muitas vezes exige um processo rigoroso de análise, experimentação e refatoração.

   Maus-cheiros relacionados: Função longa, Classe grande, Código Duplicado, Generalidade especulativa e Switches repetidos.

2. **Elegância**

   A elegância em um projeto de software está ligada à inteligência e à sofisticação de uma solução, alcançada sem recorrer à complexidade excessiva quando não for necessário. Um design elegante é eficiente e claro, frequentemente andando junto à simplicidade, pois consegue resolver problemas de forma harmoniosa e direta. A elegância reflete a habilidade de criar código que é não apenas funcional, mas também esteticamente agradável e fácil de manter.

   Maus-cheiros relacionados: Nome misterioso, Código Duplicado, Métodos Longos, Classe Grande e Campo temporário.

3. **Modularidade**

   Um projeto bem modularizado divide um problema maior e complexo em subproblemas menores, organizados em módulos ou componentes. Cada módulo é projetado para ter alta coesão, ou seja, suas funcionalidades estão

logicamente relacionadas e se concentram em um único propósito. Ao mesmo tempo, o projeto busca manter baixo acoplamento, garantindo que os módulos sejam independentes entre si, de modo que mudanças em um módulo não impactem significativamente os outros. Essa abordagem facilita a separação de responsabilidades, a divisão de tarefas dentro da equipe e a manutenção do código, permitindo que futuras modificações sejam feitas de maneira mais segura.

Maus-cheiros relacionados: Dados globais, Dados mutáveis, Classe de dados, Intermediário.

4. **Boas interfaces**

Interfaces são contratos que definem como módulos ou componentes se comunicam entre si, abstraindo os detalhes de implementação e expondo apenas o necessário para que o cliente possa consumir seus serviços ou funcionalidades. Elas servem como o ponto de encontro entre quem fornece e quem consome, garantindo que as partes possam interagir de forma consistente e previsível. Além disso, uma interface permite que a implementação interna seja substituída ou modificada, desde que mantenha os mesmos comportamentos e funcionalidades previamente definidos. Segundo Goodliffe (2006, p. 248-249, tradução nossa), os passos necessários para criar uma boa interface são: “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; Determinar a natureza da operação.”

Maus-cheiros relacionados: Lista longa de parâmetros e Classe alternativa com interfaces diferentes.

5. **Extensibilidade**

Um bom projeto de código deve considerar sua extensibilidade, isto é, a capacidade de permitir a adição de novas funcionalidades de forma fácil e sem necessidade de grandes alterações na base de código existente. No entanto, a extensibilidade deve ser cuidadosamente planejada, pois o excesso de previsões para futuras extensões pode comprometer a simplicidade e clareza do código.

Maus-cheiros relacionados: Alteração divergente e Inveja de

recursos.

6. **Evitar duplicação**

A duplicação em um projeto de código deve ser evitada ao máximo, pois compromete os princípios de simplicidade e elegância do design. Códigos duplicados não só aumentam a complexidade do projeto, mas também dificultam a manutenção e o processo de depuração. Quando uma modificação é necessária, ela deve ser aplicada em múltiplos locais, aumentando o risco de inconsistências e erros caso algum ponto seja negligenciado.

Maus-cheiros relacionados: Código duplicado e Switches repetidos.

7. **Portabilidade**

A portabilidade refere-se a capacidade do código funcionar em diferentes ambientes. Um bom design não necessariamente se preocupa com a portabilidade, portanto é importante pensar nessa decisão o mais cedo possível. Uma abordagem comum é criar uma camada de abstração para o ambiente em que o código vai rodar, permitindo assim uma maior portabilidade entre sistemas (Goodliffe, 2006, 252).

Maus-cheiros relacionados: Classe grande e Classe de dados.

8. **Código deve ser idiomático e bem documentado**

Um projeto de código idiomático significa usar as convenções e boas práticas tanto do design como de uma linguagem de programação. Isso padroniza o código e o torna mais fácil de entender para outros programadores.

E também, um bom projeto de código deve ser bem documentado, pois a documentação desempenha um papel importante na preservação de ideias, requisitos e decisões de design ao longo do desenvolvimento. Ela serve como uma referência, tanto para os desenvolvedores atuais quanto para futuros colaboradores, facilitando o entendimento do código, a continuidade do trabalho e a manutenção do sistema. Além disso, uma boa documentação contribui para a comunicação eficaz entre os membros da equipe, reduzindo ambiguidades e evitando mal-entendidos.

Maus-cheiros relacionados: Comentários, Generalidade especulativa.

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. **Classe ProcessarCashback**

   Mal-cheiro: Obsessão por primitivos

   O uso de strings para verificar o tipo de cliente. (cliente.getTipo().equals("prime")) é um exemplo de obsessão por primitivos. Isso pode ser problemático, pois a comparação de strings é propensa a erros e dificulta a evolução do código.

   Refatoração: Substituir o tipo de cliente por uma enumeração (TipoCliente) ou criar subclasses específicas para diferentes tipos de clientes

2. **Classe Venda**

   Mal-cheiro: Classe Grande

   A classe Venda concentra muitas responsabilidades diferentes, incluindo o cálculo de impostos, descontos, frete, cashback e o processamento de produtos. Isso vai contra o Princípio da Responsabilidade Única (SRP).

   Refatoração: Dividir a classe Venda em classes menores e mais coesas, como CalculadoraDeImpostos, CalculadoraDeFrete, CalculadoraDeDescontos, e ProcessadorDeCashback. Cada uma dessas classes ficaria responsável por uma parte específica da lógica, promovendo a coesão.

   Mal-cheiro: Obsessão por Primitivos

   A classe Venda utiliza strings e outros tipos primitivos para representar informações que poderiam ser encapsuladas em objetos mais significativos. Por exemplo, String formaPagamento poderia ser uma classe específica.

   Refatoração: Criar classes para encapsular os tipos primitivos, como

FormaPagamento, melhorando a clareza e a validade dos dados.

Mal-cheiro: Campo Temporário
Variáveis como totalProdutos, totalImpostos e totalDesconto, são usadas apenas para cálculos temporários dentro dos métodos. Isso indica que a lógica está excessivamente complexa e deveria ser simplificada ou separada.
Refatoração: Separar a lógica em métodos menores e reutilizáveis, eliminando a necessidade de variáveis temporárias. Por exemplo, métodos como calcularTotalProdutos e calcularTotalImpostos poderiam ser movidos para classes específicas, onde a lógica é mais clara e coesa.

Referências Bibliográficas

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

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