Princípios do Código Limpo

Exercício 1 – Importância dos princípios abaixo

1 – Usar nomenclatura adequada

É imprescindível importância de definir padrões e declarar variáveis, funções, métodos, entre outros, com nomes adequados que referenciam sua causa, para garantir a legibilidade e entendimento do código. Sem dúvida uma das meais simples, importantes e óbvias regras para manter um código limpo.

Por exemplo em 1999 a NASA perdeu $125 milhões de dólares no programa de exploração de Marte chamado: Mars Climate Orbiter.

Onde o problema no software de navegação que calculava as correções de trajetória do satélite surgiu devido à confusão entre duas equipes que trabalhavam na missão.

Uma delas calculava **usando unidades métricas** (Sistema Internacional de Unidades, SI) para medir forças e empuxo, ou seja, newtons. Enquanto a outra **usava unidades imperiais** (Sistema Inglês), medindo forças em libras-força (pound-force).

A expectativa era que o satélite entrasse em órbita a aproximadamente 226 km de altitude. No entanto, devido ao erro, ele passou a apenas cerca de **57 km** da superfície de Marte.

A essa altitude, a sonda foi exposta a forças muito maiores de fricção atmosférica e queimou ao entrar na atmosfera. A missão foi perdida, resultando em uma perda enorme financeira.

Se as variáveis tivessem nomes mais descritivos, como ***'thrust_lbf'*** (libras-força) e '***thrust_newtons'***, poderia ter sido mais fácil identificar que havia uma discrepância de unidades entre as equipes.

No relatório de investigação da NASA, um dos pontos destacados foi justamente a falta de controle e verificação das variáveis e unidades usadas nas comunicações entre os diferentes sistemas e equipes.

2 – Resolver os problemas na causa raiz

Se temos na nossa casa uma goteira proveniente de um buraco no telhado, colocar um balde apenas para que a água não se espalhe não resolve o problema, em outras palavras é como tapar o sol com a peneira. Com o desenvolvimento de software acontece a mesma coisa, o problema pode tardar a voltar, podendo inclusive se tornar uma bola de neve, desencadeando uma série de novos problemas.

Por exemplo, em um sistema simples de cadastro de usuários, cada usuário precisa fornecer um nome de usuário e um email, e o sistema deve garantir que o **email seja único** para cada cadastro.

Entretanto o sistema permite que mais de um usuário se cadastre com o mesmo email, porque não há uma verificação no banco de dados para garantir a **unicidade** do email.

A solução superficial seria o desenvolvedor adicionar uma verificação no código para verificar se o email já existe antes de cadastrar um novo usuário. Se o email já estiver em uso, o sistema exibe uma mensagem de erro.

Mas se por um acaso dois usuários tentam se cadastrar **ao mesmo tempo** com o mesmo email, o sistema pode permitir ambos os cadastros antes de verificar o banco de dados.

Portanto para tratar a causa raiz, o desenvolvedor deve implementar uma **restrição de unicidade diretamente no banco de dados**, garantindo que apenas um usuário possa cadastrar um email único, mesmo em condições de alta concorrência.

Esse é um problema que pode parecer simples à primeira vista, mas, se não for tratado, pode causar sérios problemas. Se, por exemplo, o Google tivesse ignorado questões aparentemente simples como essa, eles nunca teriam alcançado o nível de sucesso que têm como empresa de software.

3 – Seguir a política do escoteiro

O desenvolvimento de software pode ser comparado ao cuidado com algo vivo, e são os pequenos detalhes que fazem toda a diferença, destacando a qualidade de um serviço prestado, seja ele qual for. Seguir a política do escoteiro – "deixe o código melhor do que encontrou" – garante que cada melhoria contribua para a evolução, robustez e longevidade do sistema ao longo do tempo.
Sistemas legados, por exemplo, são o terror de qualquer desenvolvedor. Se tivessem recebido o devido cuidado, atenção e manutenção desde seus primeiros anos, poderiam hoje se tornar potências no mercado da tecnologia, atendendo cada vez mais às necessidades da categoria, em vez de serem fantasmas que assombram o mercado de trabalho, com seus códigos cheios de comentários de trechos antigos e

avisos assustadores de que, se você mexer em determinado ponto, o sistema inteiro vai degringolar! (risos)

Exercício 2 – Quais princípios os exemplos abaixo estão “ferindo”?

1 - Private void somaNumeros(int a, int b, int c, int d, int e, int f)

Dependendo do contexto em que esse código é inserido ele pode ferir mais de um princípio, como:

1. Single Responsibility Principle (Princípio da Responsabilidade Única) onde parece estar realizando mais de uma tarefa ao lidar com muitos parâmetros, o que pode indicar que ele está fazendo mais do que o indicado.
2. **Encapsulamento**: Ter muitos parâmetros expõe a implementação do método e pode tornar o código mais difícil de entender e manter, o que também implica no conceito citado acima.
3. **Princípio DRY (Don't Repeat Yourself)**: Se a lógica fosse realizada em vários lugares do código, o método pode ser um indicativo de que a lógica poderia ser encapsulada de maneira mais eficaz, evitando duplicação.

2 – Private void oPaiTaOn();

Uso de nomenclatura adequada mandou lembrança!

Mesmo vendo essa atrocidade com um contexto, em um sistema, é quase impossível definir do que se trata.

3 – privateDoubleChecaSaldoEAtualiza(long userId, double value)

1. **Encapsulamento**
2. Princípio da Responsabilidade Única
3. **Princípio DRY (Don't Repeat Yourself)**

4. **Princípio da Coesão:** O método pode ser considerado de baixa coesão, pois está tentando fazer duas coisas distintas (checar e atualizar). Um método com alta coesão deve ter uma única tarefa bem definida.

5. Separation of Concerns (Princípio da Separação de Preocupações): Misturar a lógica de verificação e atualização em um único método pode dificultar a manutenção do código, já que cada parte poderia ser tratada de maneira

independente. Isso vai contra o conceito de separar diferentes funcionalidades em suas respectivas camadas.

6. Princípio da Inversão de Dependência Se esse método acessa diretamente os dados do usuário (por exemplo, consultando um banco de dados), ele pode estar violando o princípio de que módulos de alto nível não devem depender de módulos de baixo nível. Uma melhor abordagem seria usar uma interface ou serviço que abstraia o acesso aos dados.

7. Teste Unitário: Um método que realiza múltiplas operações é mais difícil de testar de forma isolada. Isso pode dificultar a criação de testes unitários eficazes, pois você precisa simular tanto a verificação quanto a atualização.

Um erro que pode se tornar gravíssimo dependendo do contexto, podendo se tornar uma dor de cabeça para qualquer dev que pegar um código nesse estado, onde se utiliza duas funções em uma.