Princípios SOLID

1. Princípio de Responsabilidade Única:

- Definição: ele define que uma classe deve ter apenas uma única responsabilidade bem definida, ou seja, ela devera realizar apenas uma única tarefa ou conceito dentro de um sistema, ou seja todas as suas funções devem estar diretamente ligadas a esta responsabilidade.
- Importância: na POO ela tem uma tremenda importância pois ela facilita na manutenção e na evolução do sistema, que ela foi aplicada. Ela também ajuda na leitura e compreensão do seu código, promovendo um melhor design final de seu código.

2. Princípio Aberto/Fechado:

- Definição: Os componentes de softwares abertos, eles são liberados para a extensão, devem permitir a adição de novos comportamentos, mas fechados no quesito de modificação, ele não deve ser alterado diretamente.
- Importância: Eles auxiliam o sistema a ser flexível e adaptável a novas necessidades, não comprometendo necessidades existentes. Isso ajuda na redução de bugs e na evolução do software, muito importante de ser utilizado, principalmente em sistemas grandes e complexos.

3. Principio da Substituição de Liskov:

- Definição: Se uma subclasse (“Classe filho”) e usada no lugar de sua superclasse (“Classe pai”), o comportamento do programa deve permanecer correto e previsível. A subclasse deve preservar as expectativas da superclasse.
- Importância: Garante que o polimorfismo (e a capacidade de um objeto assumir diferentes formas**,** de uma mesma interface ou método, se comportar de maneiras diferentes dependendo do objeto que a implementa). Evita erros sutis causados por subclasses que violam contratos definidos pela superclasse.

4. Princípio da Segregação de Interfaces:

- Definição: Este princípio visa não forçar seus clientes a depender de métodos que eles não são acostumados a utilizar. Em vez disso, as interfaces devem ser especificas, apresentando apenas os métodos relacionados.
- Importância: Evita acoplamentos desnecessários e torna o sistema mais organizado. Isso ajuda na implementação, testes e manutenção, permite que diferentes partes do sistema evoluam de forma independente.

5. Princípio da Inversão de Dependência:

- Definição: este princípio define que a classe de alto nível, não deve depender diretamente da classe de baixo nível. Ambas devem de abstrações. Além disso, as abstrações não devem depender de detalhes, mas os detalhes dependem das abstrações.
- Importâncias: Gera um sistema desacoplado, onde os componentes se tornam mais dependentes e flexíveis, facilitando na injeção de dependência (Uma técnica onde as dependências de uma classe são fornecidas por um componente externo), ajuda também na testabilidade, tornando o código mais adaptável a possíveis mudanças.

Código limpo

O código limpo e aquele que transmite de forma clara a sua intenção. Quando você lê, ou configura um trecho que possui um código limpo, você percebe que não terá esforço nenhum para identificar o que cada coisa faz. Isto e essencial por que, na pratica, desenvolvedores passam muito mais tempo lendo código e corrigindo erros do que escrevendo, ate por que como exemplo o vscode tem assistência na hora de você digitar seu código, o que te auxilia bastante durante o desenvolvimento. Por isso buscar clareza em seu código e muito importante, facilitando ate a vida de quem vai manter o seu sistema depois.

Um do pilar do código limpo e a nomenclatura. Dar nomes bons para variáveis, funções e classes, tentando no máximo dar clareza no que você esta se referindo exemplo:(Calculo_de_desconto_final) e muito melhor do que (calculo), bons nomes reduz a necessidade de ficar explicando o código toda hora.

Outro ponto importante e o tamanho das funções. Funções curtas são bem mais fáceis de se entender, testar e se reutilizar. Já a função longa torna o código difícil de se entender e cansativo para o programador, e consequentemente ela se torna mais propensa a erros. O ideal e que cada função tenha sua finalidade.

Por fim, mas não menos importantes são os comentários uteis, eles têm um papel de complementar seu código. Comentários bem explicados ajudam a entender o porque e o que aquela parte de seu código faz, principalmente em partes complexas.

O Que e DRY e Padrões de Design:

O que é?

Busca eliminar duplicação de código e garante uma logica e funcionalidade que só exista

apenas em um lugar.

Para que serve?

1. Evitar repetição de códigos e menos bugs.

2. Facilitar manutenção mudanças em um único lugar.

3. Código mais organizado.

4. Garantir consistência regras únicas em todo o sistema

Exemplo pratico:

Sem DRY: a mesma validação repetida em muitos arquivos.

Com DRY: validação centralizada em uma função reutilizável

Por que é importante?

1.Economiza mais tempo para não reescrever o códigos

2.Reduz erros sem conflitantes da mesma logica

3.Código mais limpo e profissional

Como utiliza: Identifique as linhas repetidas, após extraia para função, classes e módulos, depois e só centralize a logica

Impacto: Código mais limpo e profissional gerando um sistema mais fácil de se manter, tornando um desenvolvimento mais eficiente.

Conclusão:

O DRY é essencial para reduzir e centralizar em um único lugar para garantir um código mais limpo e fácil, para manutenções e facilita na compreensão. Sua aplicação reduz bugs e economiza tempo e melhora a estabilidade, sem eliminar repetições justificáveis. Ao extrair padrões repetidos em funções ou módulos reutilizáveis, criando sistemas mais profissionais e eficientes.

Teste Unitário:

O que e Teste Unitário na (OOP):

O teste unitário e um tipo de teste de software ou algoritimo, que tem o objetivo de testar ou verificar a partes de uma aplicação, que pode chamar de “unidades”. No contexto da OOP, essa unidade costuma ser um método de uma classe.

O principal objetivo de um teste unitário é focar em uma parte específica do programa e verificar se ela funciona. Ou seja, que são criados métodos de entrada e saída para ter certeza que o objetivo esta certo.

Características do Teste Unitário em POO:

- **Foco em objetos:** o testes são feitos para verificar o desempenho de cada objeto especifico e seus métodos.
- **Isolamento:** para fazer um teste de unidade separado e normal usar o duble de teste. Eles simulam outros itens fora do programa como bancos der dados.
- **Automação:** Os testes unitários são automatizados e podem ser executados a qualquer momento, proporcionando uma resposta rápida sobre a saúde do código.
- **Repetibilidade:** Um teste unitário sempre produz o mesmo resultado se não alterar o código.

O Que É Test-Driven Development (TDD)?

O Desenvolvimento Guiado por Testes (TDD) é uma forma de desenvolvimento de software que e ao contrario da lógica tradicional. Em vez de escrever o código primeiro e os testes depois, o TDD faz com que os testes sejam escritos antes do código.

O TDD se baseia em um ciclo repetitivo de três fases, conhecido como "Red-Green-Refactor":

1. **Red (Vermelho):** O desenvolvedor escreve um teste para uma função que ainda não existe. Como o código não foi feito, o teste quando ser executado, irá falhar (ficar "vermelho").
2. **Green (Verde):** O desenvolvedor escreve o mínimo de código necessário para fazer o teste ficar "verde". O foco é deixar mais simples e fazer o teste funcionar, sem se preocupar com a ordem do código.
3. **Refactor (Refatorar):** Com o teste funcionando, o desenvolvedor pode agora refatorar(Refazer, modificar) o código, ou seja, melhorá-lo, remover duplicações, aumentar a clareza e a eficiência, sem alterar o comportamento. O TDD garante que a refatoração não fez nenhum bug ou erro.

Este ciclo é repetido para cada nova função e código, garante os testes mais limpos, com melhor design, e melhor forma de ser feito.

Na minha opinião escrever os testes e depois usar eles e mais difícil pois você precisa de um pouco mais de tempo de pratica e técnica de codificação mas por outro lado pode ser mais fácil mas eu continuo com o pensamento de fazer o código na hora e corrigir a o mesmo tempo e melhor para deixá-lo mais coerente e você ainda sabe os erros por causa que você fez por partes mas a técnica TDD pode ser mais rápida e as vezes melhor.

Refatoração

Refatoração o que é: Imagine que você escreveu um texto na pressa e cheio de repetições. Refatorar seria revisar esse texto, cortando excessos, organizando melhor as ideias e deixando tudo mais fluido sem mudar o sentido original, isso é refatorar.

Melhorias que refatorar as linhas de código traz:

• Eliminar código duplicado:

A duplicação de código leva a inconsistências e aumentando a dificuldade de manutenção.

• Simplificar lógicas complexas:

Lógicas muito difíceis podem ser complicadas de entender podendo causar erros. Refatorar para simplificar essas lógicas torna o código mais simples e menos suscetível a bugs.

• Melhorar nomes de variáveis, funções e classes:

Nomes claros tornam o código mais legível e fácil de entender para os desenvolvedores.

Qual e a necessidade real de refatorar

• Código Legível = Menos Bugs: Código claro facilita a identificação de erros e a implementação de novas funcionalidades.

• Melhora Performance: Pode otimizar algoritmos e estruturas de dados.

• Prepara o Código para Testes: Código bem estruturado é mais fácil de testar.

Resumo do Princípio da Responsabilidade Única

O princípio da responsabilidade Única estabelece que uma classe dever ter apenas uma responsabilidade bem definida, ela deve realizar uma única tarefa ou representar um único conceito dentro do sistema. Essa pratica e muito importante na POO, pois promove um design mais limpo, facilitando a leitura de seu código. Após aplicar o SRP, evitamos que alterações em certa parte de classes afetem outras funcionalidades do código que não tem relações com o que foi alterado, promovendo a redução de erros em seu código.

Pergunta instigante:

Se uma classe tiver múltiplas responsabilidades, como isso pode impactar a manutenção e a escalabilidade de um sistema em longo prazo?