Clean Code no Node

⚠️ “Maneira que o SOLID é aplicado na prática, pois os princípios se sobrepõem, estando totalmente conectados” - Diego Fernandes

Princípios do SOLID

▼ SRP - Single Responsiblity Principle (Princípio da responsabilidade única)

Cada classe (metódo, função ou módulo) deve ser responsável por uma única coisa

Como identificar se este princípio está sendo seguido:

Olhe para a classe e descreva o que essa classe faz.

Exemplo: essa classe cria um usuário E envia um email de boas vindas

- A utilização do conectivo E define que tem 2 tarefas diferentes sendo reutilizados na classe.

▼ OCP - Open-Closed Principle (Princípio Aberto-Fechado)

Deve ser possível adicionar comportamentos na classe, mas nunca modificar um comportamento já existente

Como identificar se este princípio está sendo seguido:

Quando há condicionais em partes do código onde vai ser necessário adicionar mais estruturas de decisão futuramente.

Exemplo: classe que lida com calculo de frete

- Problema: Adição de transportadoras com cálculos diferentes.

1. Foi adicionado a transportadora correios e é necessário uma estrutura de decisão para saber se é a transportadora selecionada e então o calculo do frete de um jeito diferente.
2. Foi adicionado a transportadora testeExpress e é necessário mais uma estrutura de decisão para saber se foi selecionada e para seu calculo.

Nesse caso temos o princípio sendo violado pois o comportamento da classe está sendo modificado.

▼ LSP - Liskov Substitution Principle (Princípio da substituição de Liskov)

Deve ser possível substituir uma dependência de uma classe por outra desde que a nova dependência seja do mesmo formato.

Como identificar se este princípio está sendo seguido:

Se houver erros de funcionamento da classe por causa da troca de dependências de uma classe

Exemplo: Camada Controller conectada a camada de Repository que se comunica com um banco de dados Postgress

- Caso seja necessário adicionar um bando de dados Mysql

1. Controller deve passar para a camada Repository esse novo banco de dados e está camada deve funcionar corretamente.

Se passar a ter erros, o princípio foi violado.

▼ ISP - Interface Segregation Principle (Princípio da Segregação da Interface)

Instrui que as interfaces (Traz regras que uma classe deve seguir) sejam modularizadas. Ou seja, criação de algumas interfaces menores invés de

uma geral para representar a máxima variedade de comportamento da classe

Exemplo de interfaces para impressoras:

- **Apenas uma interface**

```
//
interface PrinterInterface(){
    print: () ⇒ void
    scan: () ⇒ void
}

class Printer implements PrinterInterface {}
```

No caso acima, caso seja necessário criar uma instância de uma impressora que apenas imprima, não seria possível porque a interface `PrinterInterface` está impondo que **todas as implementações** dela precisam ter os dois métodos, `print` e `scan`.

- **Dividindo a interface**

```


interface PrinterWithScanInterface(){
    scan: () ⇒ void
}
interface BasicPrinterInterface(){
    print: () ⇒ void
}

class Printer implements BasicPrinterInterface, PrinterWithScanInterface {
    scan: () ⇒ void
    print: () ⇒ void
}

class Printer implements BasicPrinterInterface {
    print: () ⇒ void
}
```

Agora é possível ter impressora que apenas imprime e impressora que imprime e escaneia.

▼ DIP - Dependency Inversion Principle (Princípio da inversão da dependência)

As formas de lidar com as dependências (arquivos ou funcionalidades externas) das classes deve ser inversa ao modelo tradicional.

Modelo tradicional:

```
import {createUserOnDatabase} from 'db'


function createUser(){
   creteUserOnDatabase()
}
```

Problema: A função `createUser` depende diretamente de `createUserOnDatabase`. Caso queira trocar a implementação de `createUserOnDatabase` terá que fazer alterações na função `createUser` . O código não está seguindo o **DIP.**

Aplicando o DIP

Em vez de `createUser` depender diretamente de `createUserOnDatabase`, `createUserOnDatabase` vai ser passado como argumento para a função `createUser`

```
//arquivo dentro de repository
function createUser(creteUserOnDatabase: () => void){
   creteUserOnDatabase()
}
```

Em outro arquivo onde a função `createUser` é instanciada, passa a ser importado a função `createUserOnDatabase` e passada como parâmetro para `createUser`

```
//arquivo dentro de controller
import {createUserOnDatabase} from 'db'
```

```
import {createUser} from 'repository'

createUser(createUserOnDatabase)
```

- **Flexibilidade**: Pode ser passado diferentes implementações para `createUser` sem alterar seu código.
- **Desacoplamento**: A função `createUser` não está mais acoplada à implementação de banco de dados específica (`createUserOnDatabase`), o que a torna mais reutilizável e testável.
