Renato Portilho Costa RA: 24101301

5° Resumo

Os princípios SOLID são um conjunto de boas práticas pra deixar o código mais organizado, flexível e fácil de mexer.

1- Single Responsibility Principle

Cada classe tem que ter só um motivo pra mudar, ou seja, uma única responsabilidade. Se um código faz várias coisas ao mesmo tempo, ele fica confuso e difícil de manter. Exemplo errado:

```
Class Relatorio {
    void gerarRelatorio() {}
    void SalvarNoBanco() {}
    void EnviarPorEmail() {}
}
```

Aqui, a classe relatorio está fazendo tudo ao mesmo tempo, o certo é separar as funções Certo:

```
Class GeradorRelatorio {}
Class BancoDeDados {}
Class ServicoEmail {}
```

2- OPen/Closed Principle

Aqui, o código deve ser fácil de expandir, mas sem precisar alterar o que já existe Exemplo errado:

```
Class CalculadoraSalario {
    double calcularSalario(String TipoFuncionario) {
        if(TipoFuncionario.equals("CLT")) {
            return 3000;
        } else if(TipoFuncionario.equals("PS")) {
            return 4000;
        }
    }
}
```

Se surgir um novo tipo de funcionario, temos que modificar essa classe, e isso pode causar erros Certo:

```
Interface Funcionario {
    double calcularSalario();
}
Class CLT implements Funcionario {
    public double calcularSalario() { return 3000; }
}
Class PS implements Funcionario {
    Public double calcularSalario() { return 4000; }
}
```

3- Liskov Substitution Principle

Se uma classe filha herda de uma mãe, ela tem que poder substituir a mãe sem quebrar nada. Basicamente: a herança tem que fazer sentido. Exemplo errado:

```
Class Passaro {
    Void Voar () {}
}

Class Pinguim extends Passaro {}
```

Pinguim não voa, então a herança esta errada. Certo:

```
Interface Passaro {
    Void mover();
}

Class Passaro que voa implements Passaro {
    public void mover () { System.out.println ("Voando!");}
}

class Pingum implements Passaro {
    public void mover() { System.out.println("Andando!");}
}
```

4- Interface Segregation Principle (ISP)

Aqui a regra é: melhor ter varias interfaces pequenas do que uma interface gigante cheia de métodos que nem todo mundo usa. Exemplo errado:

```
Interface Funcionario {
    Void trabalha ();
    void baterPonto();
    void tiraFerias();
}
```

Se tivermos um freelancer, ele não bate ponto, logo, essa interface não faz sentido. Certo:

```
Interface trabalhavel {
    void trabalhar ();
}

Interface Assalariado extends trabalhavel {
    Void BaterPonto ();
    Void tiraFerias();
}
```

5- Dependency inversion Principle

A ideia aqui é que módulos de Alto Nível não devem depender diretamente de módulos de baixo Nível, em vez disso, ambos devem depender de abstrações. Exemplo errado:

```
Class BancoDeDados {
    Void Salvar()
}

Class ServicoUsuario {
    BancoDeDados banco = New BancoDeDados();
    Void processar() { banco.Salvar();}
}
```

Aqui, ServicoUsuario tá amarrado ao BancoDeDados, dificultando mudanças

Certo:

```
Interface Repositorio {
    void Salvar();
}

class BancoDeDados implements Repositorio {
    public void Salvar() {}
}

class ServicoUsuario {
    Repositorio repositorio;
    ServicoUsuario(Repositorio r) { this.repositorio = r; }
    void processa() { repositorio.Salvar(); }
}
```