DESENVOLVIMENTO WEB III

Princípios SOLID

Prof. Esp. Érick Henrique Pereira Nicolau

O que são os Princípios SOLID?

- Conjunto de cinco princípios de design de software que ajudam a criar código mais:
  - Flexível
  - Fácil de manter
  - Fácil de testar
  - Fácil de estender

Por que usar os Princípios SOLID?

- Previne problemas comuns em projetos de software:
  - Alto acoplamento: Mudanças em uma parte do código afetam outras partes de forma inesperada.
  - Baixa coesão: Classes ou módulos com responsabilidades misturadas, dificultando a compreensão e manutenção.
  - Código rígido: Difícil de modificar ou estender sem introduzir novos bugs.

Os Cinco Princípios SOLID

- S - Single Responsibility Principle (SRP)
- O - Open/Closed Principle (OCP)
- L - Liskov Substitution Principle (LSP)
- I - Interface Segregation Principle (ISP)
- D - Dependency Inversion Principle (DIP)

S - Single Responsibility Principle (SRP)

- Uma classe deve ter apenas uma razão para mudar.
- Cada classe deve ter uma única responsabilidade bem definida.
- Isso torna o código mais fácil de entender, testar e modificar.

Exemplo de Violação deste principio SRP

```
class Livro {
  <conteúdo>

  salvarNoBancoDeDados() {
    <conteúdo>
  }
}
```

Qual o erro??????

- A classe Livro tem duas responsabilidades:
  - representar um livro
  - persistir dados no banco de dados.

Aplicação do SRP

```
class Livro {
  <propriedades e métodos relacionados ao livro>
}

class RepositorioDeLivros {
  salvar(livro) {
    <lógica salvar dados>
  }
}
```

A responsabilidade de persistir dados foi movida para a classe RepositorioDeLivros, tornando o código mais coeso.

O - Open/Closed Principle (OCP)

- As entidades de software devem estar abertas para extensão, mas fechadas para modificação.
- Você deve ser capaz de adicionar novas funcionalidades sem modificar o código existente.
- Isso promove a reutilização de código e reduz o risco de introduzir bugs em partes já testadas.

Exemplo de Violação do OCP

```
function calcularArea(forma) {
    if (forma instanceof Retangulo) {
        return forma.largura * forma.altura;
    } else if (forma instanceof Circulo) {
        return Math.PI * forma.raio * forma.raio;
    }
    <continua para n formas>
}
```

Qual o erro?

A cada nova forma geométrica, a função calcularArea precisa ser modificada.

```
class Forma {
        calcularArea() {
                // Método abstrato, deve ser implementado nas subclasses
        throw new Error("Método abstrato não implementado");
        }
}
class Retangulo extends Forma {
        calcularArea() {
                return this.largura * this.altura;
        }
}
class Circulo extends Forma {
        calcularArea() {
                return Math.PI * this.raio * this.raio;
        }
}
function calcularArea(forma) {
        return forma.calcularArea();
}
```

Novas formas podem ser adicionadas criando subclasses de Forma sem modificar a função calcularArea.

L - Liskov Substitution Principle (LSP)

- Subclasses devem se comportar de forma consistente com suas classes pai.
- Isso garante que o código que usa uma classe pai funcione corretamente mesmo se receber uma instância de uma de suas subclasses.

Exemplo de violação

- Classe quadrado tendo como classe pai um retângulo
- Deveria ter uma classe forma

Interface Segregation Principle (ISP)

- Muitos clientes específicos são melhores do que uma interface de propósito geral.
- Interfaces devem ser pequenas e coesas, atendendo às necessidades específicas de seus clientes.
- Isso evita que as classes dependam de métodos que não utilizam.

```
interface Impressora {
  imprimir();
  escanear();
  enviarFax();
}

class ImpressoraSimples implements Impressora {
  imprimir() {
}

  escanear() {
    throw new Error("Esta impressora não suporta escaneamento");
  }

  enviarFax() {
    throw new Error("Esta impressora não suporta fax");
  }
}
```

```
interface Impressora {
  imprimir();
}

interface Scanner {
  escanear();
}

interface Fax {
  enviarFax();
}

class ImpressoraMultifuncional implements Impressora, Scanner, Fax {
}

class ImpressoraSimples implements Impressora {
}
```

Interfaces menores e mais específicas permitem que as classes implementem apenas o que precisam.

D - Dependency Inversion Principle (DIP)

- Dependa de abstrações, não de implementações.
- Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
- Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.
- Isso promove o baixo acoplamento e facilita a troca de implementações.

```
class Motor {
  ligar() {
    // ...
  }
}

class Carro {
  constructor() {
    this.motor = new Motor(); // Depen. direta de uma implementação concreta
  }

  ligar() {
    this.motor.ligar();
  }
}
```

A classe Carro depende diretamente da classe Motor

```
interface IMotor {
  ligar();
}

class MotorEletrico implements IMotor {
}

class MotorCombustao implements IMotor {
}

class Carro {
  constructor(motor) {
    this.motor = motor; // Dependência de uma abstração (interface)
  }

  ligar() {
    this.motor.ligar();
  }
}
```

A classe Carro depende da interface IMotor, permitindo que diferentes tipos de motores sejam usados.