SOLID: Princípios para um Código Mais Limpo e Flexível

Introdução

Este documento aborda os princípios SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation e Dependency Inversion) e como eles contribuem para a criação de software de alta qualidade, juntamente com exemplos práticos em TypeScript.

O que é SOLID?

SOLID é um acrônimo que representa cinco princípios de design de software que visam tornar o código mais fácil de entender, manter e estender ao longo do tempo.

Single Responsibility Principle (SRP)

Cada classe ou módulo deve ter apenas uma responsabilidade. Isso significa que uma classe deve ter apenas um motivo para mudar.

```typescript
// Violação do SRP
class Employee {
    constructor(public name: string, public position: string) {}

    getDetails() {
        return `${this.name} works as ${this.position}`;
    }

    save() {
        // código para salvar empregado no banco de dados
    }

    generateReport() {
        // código para gerar relatório do empregado
```

```
    }
}

// Aplicando o SRP
class Employee {
    constructor(public name: string, public position: string) {}

    getDetails() {
        return `${this.name} works as ${this.position}`;
    }
}

class EmployeeRepository {
    save(employee: Employee) {
        // código para salvar empregado no banco de dados
    }
}

class EmployeeReportGenerator {
    generate(employee: Employee) {
        // código para gerar relatório do empregado
    }
}
```

Open/Closed Principle (OCP)

As entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão,

mas fechadas para modificação. Isso significa que você deve ser capaz de adicionar novas funcionalidades sem modificar o código existente.

```typescript
// Violação do OCP
class AreaCalculator {
    calculateRectangleArea(width: number, height: number): number {
        return width * height;
    }

    calculateCircleArea(radius: number): number {
        return Math.PI * radius * radius;
    }
}

// Aplicando o OCP
interface Shape {
    area(): number;
}

class Rectangle implements Shape {
    constructor(private width: number, private height: number) {}

    area(): number {
        return this.width * this.height;
    }
}
```

```
class Circle implements Shape {

    constructor(private radius: number) {}


    area(): number {

        return Math.PI * this.radius * this.radius;

    }

}


class AreaCalculator {

    calculate(shapes: Shape[]): number {

        return shapes.reduce((sum, shape) => sum + shape.area(), 0);

    }

}
```

Liskov Substitution Principle (LSP)

Os objetos de uma classe derivada devem ser substituíveis por objetos da classe base sem afetar a correção do programa. Isso significa que as subclasses devem ser capazes de substituir suas classes base sem quebrar o código que as utiliza.

```typescript
// Violação do LSP

class Bird {

    fly(): void {

        console.log("Flying");

    }

}
```

```
class Penguin extends Bird {
    fly(): void {
        throw new Error("Penguins can't fly");
    }
}

// Aplicando o LSP
interface Bird {
    makeSound(): void;
}

interface FlyingBird extends Bird {
    fly(): void;
}

class Sparrow implements FlyingBird {
    fly(): void {
        console.log("Flying");
    }

    makeSound(): void {
        console.log("Chirp");
    }
}
```

```
class Penguin implements Bird {
    makeSound(): void {
        console.log("Quack");
    }
}
```

Interface Segregation Principle (ISP)

Uma classe não deve ser forçada a implementar interfaces que ela não usa. Isso significa que você deve criar interfaces menores e mais específicas para que as classes possam implementar apenas os métodos que realmente precisam.

```typescript
// Violação do ISP
interface Worker {
    work(): void;
    eat(): void;
    sleep(): void;
}

// Aplicando o ISP
interface Workable {
    work(): void;
}

interface Eatable {
    eat(): void;
}
```

```
interface Sleepable {
    sleep(): void;
}

class Robot implements Workable {
    work(): void {
        console.log("Robot working");
    }
}

class Human implements Workable, Eatable, Sleepable {
    work(): void {
        console.log("Human working");
    }

    eat(): void {
        console.log("Human eating");
    }

    sleep(): void {
        console.log("Human sleeping");
    }
}
```

Dependency Inversion Principle (DIP)

As classes de alto nível não devem depender de classes de baixo nível. Ambas devem depender de abstrações. As abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações. Isso significa que você deve usar interfaces ou classes abstratas para desacoplar as dependências entre as classes.

```typescript
// Violação do DIP
class BackendDeveloper {
    develop() {
        console.log("Writing backend code");
    }
}

class FrontendDeveloper {
    develop() {
        console.log("Writing frontend code");
    }
}

class Project {
    private backendDeveloper = new BackendDeveloper();
    private frontendDeveloper = new FrontendDeveloper();

    deliver() {
        this.backendDeveloper.develop();
        this.frontendDeveloper.develop();
    }
```

```
}

// Aplicando o DIP
interface Developer {
    develop(): void;
}

class BackendDeveloper implements Developer {
    develop() {
        console.log("Writing backend code");
    }
}

class FrontendDeveloper implements Developer {
    develop() {
        console.log("Writing frontend code");
    }
}

class Project {
    private developers: Developer[];

    constructor(developers: Developer[]) {
        this.developers = developers;
    }
```

```
    deliver() {
        this.developers.forEach(developer => developer.develop());
    }
}
```

Conclusão

Ao aplicar os princípios SOLID em conjunto com a Arquitetura Limpa, que promove a separação de responsabilidades em camadas e a inversão de dependências, você estará no caminho para construir um software mais robusto, flexível, testável e fácil de manter a longo prazo.