Desvendando

SOLID Principles

Isaac Gomes

o que é SOLID ?

S O L I D

SOLID é um acrônimo que consolida 5 itens que são considerados como boas práticas no mundo do desenvolvimento orientado a objetos.

Isaac Gomes

o que é cada acrônimo ?

S => Single Responsibility Principle

O => Open/Closed Principle

L => Liskov Substitution Principle

I => Interface Segregation Principle

D => Dependency Inversion Principle

Isaac Gomes

SOLID

Single Responsibility Principle

Cada classe deve ter uma única responsabilidade ou razão para mudar. Isso significa que uma classe deve ter apenas uma responsabilidade ou uma função específica para manter a coesão.

Isaac Gomes

SOLID

Single Responsibility Principle

```ts
class UserService {
  createUser(name: string, email: string): void {
    this.saveUserToDatabase(name, email);
  }

  private saveUserToDatabase(name: string, email: string){}

  sendWelcomeEmail(email: string){}
}
```

note quantas responsabilidades temos em um unico local (criar, persistir e enviar email)

concorda que persistir os dados e enviar email não deviria influenciar na criação?

Isaac Gomes

TS

SOLID

Single Responsibility Principle

```
class UserService {
  private userRepository: UserRepository;
  private emailService: EmailService;

  constructor(
    userRepository: UserRepository,
    emailService: EmailService
  ){
    this.userRepository = userRepository;
    this.emailService = emailService;
  }

  createUser(name: string, email: string): void {
    this.userRepository.save(name, email);
    this.emailService.sendWelcomeEmail(email);
  }
}
```

note que agora quebramos essas **responsabilidades** em outros **services** e agora esse service de user tem somente um motivo para mudar

Isaac Gomes

TS

SOLID

Open/Closed Principle

Entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação. Isso implica que o comportamento de uma entidade pode ser estendido sem alterar seu código-fonte original.

Isaac Gomes

SOLID
Open/Closed Principle

```ts
class DiscountCalculator {
  calculateDiscount(product: { type: string, price: number }): number {
    if (product.type === 'electronics') {
      return product.price * 0.1;
    }
    if (product.type === 'clothing') {
      return product.price * 0.2;
    }
    return 0;
  }
}
```

note que qualquer adição resulta em uma modificação na class

Isaac Gomes

TS

SOLID
Open/Closed Principle

```ts
class ClothingDiscount implements DiscountStrategy {
  calculate(price: number): number {
    return price * 0.2;
  }
}
```

```ts
class DiscountCalculator {
  private discountStrategies: { [key: string]: DiscountStrategy } = {};
  constructor() {
    this.discountStrategies['electronics'] = new ElectronicsDiscount();
    this.discountStrategies['clothing'] = new ClothingDiscount();
  }

  calculateDiscount(product: { type: string, price: number }): number {
    const strategy = this.discountStrategies[product.type];
    if (!strategy) return 0;
    return strategy.calculate(product.price);
  }
}
```

agora note criamos um strategy e basicamente se quisermos adicionar um novo basta implementar e injetar

Isaac Gomes

TS

SOLID

Liskov Substitution Principle

Objetos de uma classe derivada devem poder substituir objetos de uma classe base sem alterar a funcionalidade do programa. Isso significa que as subclasses devem ser substituíveis por suas classes base.

Isaac Gomes

SOLID

Liskov Substitution Principle

```ts
class Bird {
  fly(): void {
    console.log('Flying...');
  }
}

class Eagle extends Bird {}

class Penguin extends Bird {
  fly(): void {
    throw new Error('Penguins cannot fly!');
  }
}

function makeBirdFly(bird: Bird) {
  bird.fly();
}
```

Vamos considerar um exemplo de uma classe base Bird e duas **subclasses** Eagle e Penguin. Ambas as **subclasses herdam** de Bird, mas um pinguim não pode voar, quebrando assim o **LSP**.

Isaac Gomes

TS

SOLID

Liskov Substitution Principle

```ts
interface Flyable {
  fly(): void;
}

class Bird {}

class Eagle extends Bird implements Flyable {
  fly(): void {
    console.log('Flying...');
  }
}

class Penguin extends Bird {}

function makeBirdFly(bird: Flyable) {
  bird.fly();
}
```

Para resolver este problema, podemos usar a composição em vez da herança, ou podemos introduzir uma interface para classes de aves voadoras.

Isaac Gomes

TS

SOLID

Interface Segregation Principle

Uma interface deve ter apenas os métodos que são relevantes para a classe que a implementa. Classes não devem ser forçadas a implementar métodos que não utilizam.

Isaac Gomes

SOLID

Interface Segregation Principle

```ts
interface Worker {
  work(): void;
  attendMeeting(): void;
  fileReport(): void;
}

class Developer implements Worker {
  work(): void {}

  attendMeeting(): void {}

  fileReport(): void {
    // Developer não precisa implementar isso
    throw new Error('Method not implemented.');
  }
}

class Manager implements Worker {
  work(): void {}

  attendMeeting(): void {}

  fileReport(): void {
    // Manager não precisa implementar isso
    throw new Error('Method not implemented.');}
}
```

veja como temos uma **interface** muito generica acabamos com **metodos** que não dizem respeito a quele **contexto**

Isaac Gomes

TS

SOLID

Interface Segregation Principle

```ts
interface Workable {
  work(): void;
}

interface MeetingAttendee {
  attendMeeting(): void;
}

interface ReportFiler {
  fileReport(): void;
}

class Developer implements Workable, MeetingAttendee {
  work(): void {}
  attendMeeting(): void {}
}

class Manager implements MeetingAttendee, ReportFiler {
  attendMeeting(): void {}
  fileReport(): void {}
}
```

agora quebramos as **interfaces** e conseguimos fazer uma **implementação** que faz sentido no contexto

Isaac Gomes

TS

SOLID

Dependency Inversion Principle

Dependa de abstrações, não de concretizações. Isso significa que os módulos de alto nível não devem depender de módulos de baixo nível, mas ambos devem depender de abstrações. Além disso, as abstrações não devem depender de detalhes, mas os detalhes devem depender de abstrações.

Isaac Gomes

SOLID
Dependency Inversion Principle

```ts
class EmailService {
  sendEmail(to: string, subject: string, body: string): void {
    console.log(`Sending email to ${to}: ${subject} - ${body}`);
  }
}

class OrderProcessor {
  private emailService: EmailService;

  constructor() {
    this.emailService = new EmailService();
  }

  processOrder(orderId: string): void {
    this.emailService.sendEmail(
      'customer@example.com',
      'Order Confirmation',
      `Your order ${orderId} has been processed.`
    );
  }
}
```

veja com instanciamos diretamente acabamos tendo esse forte acoplamento

Isaac Gomes

TS

SOLID

Dependency Inversion Principle

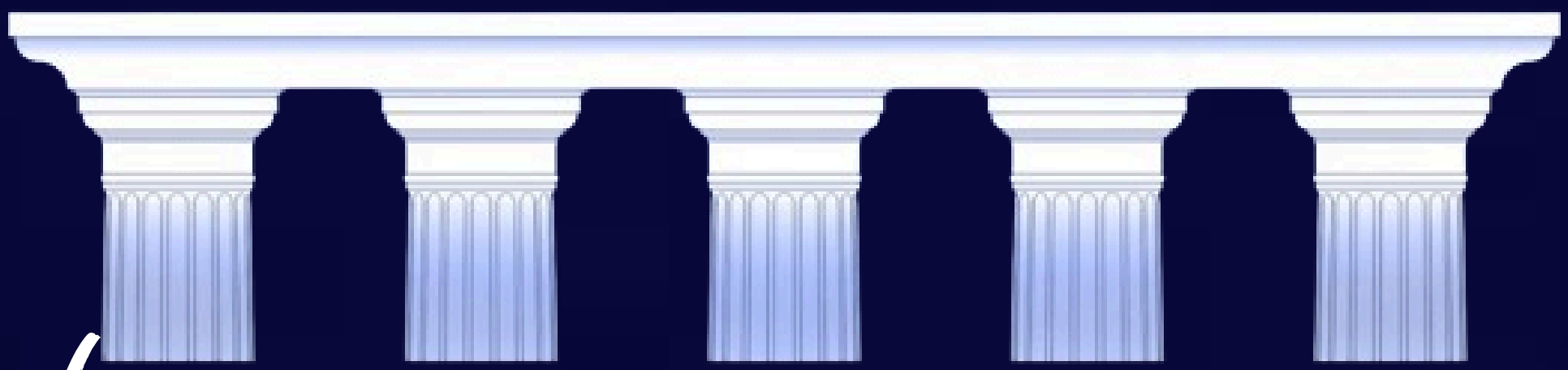
```ts
interface NotificationService {
  sendNotification(to: string, subject: string, body: string): void;
}

class EmailService implements NotificationService {
  sendNotification(to: string, subject: string, body: string): void {

  }
}

class OrderProcessor {
  constructor(
    private notificationService: NotificationService
  ) {}

  processOrder(orderId: string): void {
    this.notificationService.sendNotification(
      'customer@example.com',
      'Order Confirmation',
      `Your order ${orderId} has been processed.`
    );
  }
}
```

para **invertemos a dependências** criamos uma **interface** agora dependemos da interface e podemos invejar qualquer **implementação** que siga o que a interface requer

Isaac Gomes

TS

S O L I D



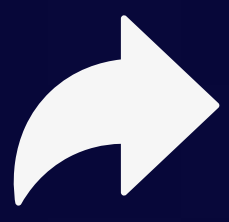
agora as letras começam
a tornar princípios
simples de entender

Isaac Gomes

TS

Gostou?

Curta



Compartilhe

Salve

Isaac Gomes