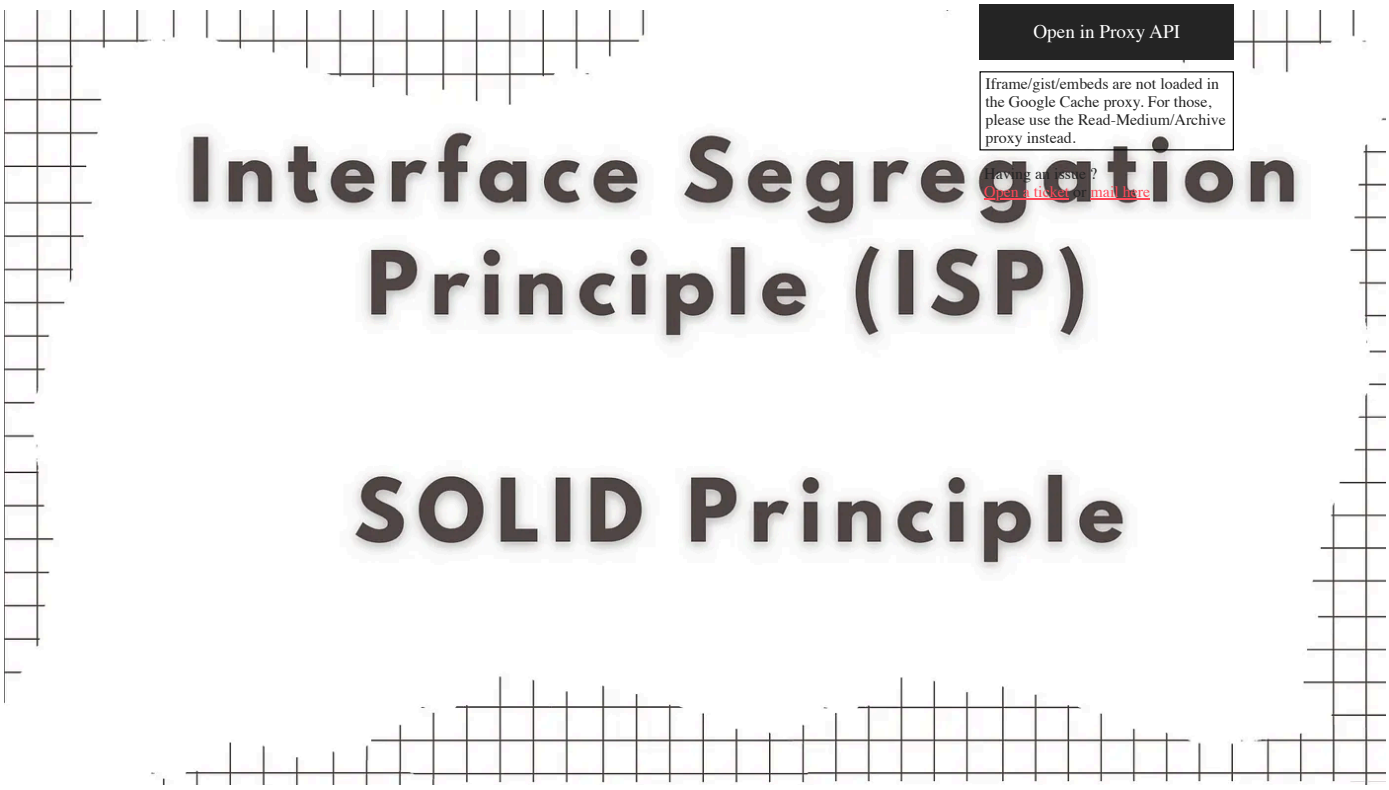
4. Interface Segregation Principle (ISP): SOLID Principle

Ramdhas · Follow
2 min read · Oct 25, 2023



The Interface Segregation Principle (ISP) suggests that a class should not be forced to implement methods it doesn't need. In other words, a class should have small, focused interfaces rather than large, monolithic ones. This helps to avoid unnecessary dependencies and ensures that classes only implement the methods they actually need.

Let's use a simple example with credit cards to explain this principle using Swift:

Before Applying ISP:

Suppose you have an interface called `PaymentMethod` that has methods for both credit card and PayPal payments:

```
protocol PaymentMethod {
    func processCreditCardPayment()
    func processPayPalPayment()
}
```

Now, let's create a class for a credit card payment:

```
class CreditCardPayment: PaymentMethod {
    func processCreditCardPayment() {
        // Code to process a credit card payment
    }

    func processPayPalPayment() {
        // This method is not relevant for credit card payments
```

```
    }
}
```

In this example, the `CreditCardPayment` class has to implement the `processPayPalPayment` method even though it doesn't use it. This violates the ISP because the class is forced to have unnecessary methods.

After Applying ISP:

To adhere to the ISP, you can create separate interfaces for different payment methods:

```
protocol CreditCardPayment {
    func processCreditCardPayment()
}

protocol PayPalPayment {
    func processPayPalPayment()
}
```

Now, when implementing a credit card payment class, you only need to conform to the relevant interface:

```
class CreditCardPaymentProcessor: CreditCardPayment {
    func processCreditCardPayment() {
        // Code to process a credit card payment
    }
}
```

With this approach, you follow the ISP by allowing classes to implement only the methods they actually need. This makes the code more focused, easier to understand, and avoids the burden of implementing unnecessary methods.

. . .

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions, making code more flexible.
Read more: *https://medium.com/@ramdhas/5-dependency-inversion-principle-dip-solid-principle-197517a3fd9a*

. . .

👏👏👏👏 Applaud to express your encouragement. Join me for additional insights and let's progress together.

Solid Principles | Clean Code | Software Design | Code Quality | Interface Segregation

Follow

Written by Ramdhas

225 Followers · 31 Following

Skills: iOS, Swift, SwiftUI, Html, Css, Javascript, React.js. Lives in Stockholm, Sweden.

Responses (1)