Tech Talks

SOLID Principles

SOLID is an acronym that represents a set of **5 design principles** that help create well-**structured, maintainable, and flexible** software.

These principles were introduced by **Robert C. Martin** and are widely used in object-oriented programming to guide the design of software systems.

Let's break down each principle with simple examples in Swift (iOS):

Points for discussion

1. **Single Responsibility Principle (SRP):**
2. **Open/Closed Principle (OCP):**

3. **Liskov Substitution Principle (LSP):**
4. **Interface Segregation Principle (ISP):**
5. **Dependency Inversion Principle (DIP):**

1. Single Responsibility Principle (SRP):

" A class should have only one reason to change, meaning it should have only one job or responsibility "

```swift
// Incorrect Example violating SRP
class User {
    var name: String
    var email: String

    func saveToDatabase() {
        // Code to save user data to database
    }

    func sendEmail() {
        // Code to send a welcome email
    }
}
```

In this case, the User class violates SRP by handling both database operations and sending emails. It's better to separate concerns:

1. Single Responsibility Principle (SRP):

" A class should have only one reason to change, meaning it should have only one job or responsibility "

```swift
// Incorrect Example violating SRP
class User {
    var name: String
    var email: String

    func saveToDatabase() {
        // Code to save user data to database
    }

    func sendEmail() {
        // Code to send a welcome email
    }
}
```

```swift
// Correct Example following SRP
class User {
    var name: String
    var email: String
}

class UserDatabase {
    func save(user: User) {
        // Code to save user data to database
    }
}

class EmailService {
    func sendWelcomeEmail(to user: User) {
        // Code to send a welcome email
    }
}
```

2. Open/Closed Principle (OCP):

Software entities should be open for extension but closed for modification.

```swift
// Incorrect Example violating OCP
class Shape {
    func calculateArea() -> Double {
        return 0
    }
}

class Rectangle: Shape {
    var width: Double
    var height: Double

    override func calculateArea() -> Double {
        return width * height
    }
}
```

The Shape class violates OCP because it needs modification whenever a new shape is added.

Instead, make it open for extension:

2. Open/Closed Principle (OCP):

Software entities should be open for extension but closed for modification.

```swift
// Incorrect Example violating OCP
class Shape {
    func calculateArea() -> Double {
        return 0
    }
}

class Rectangle: Shape {
    var width: Double
    var height: Double

    override func calculateArea() -> Double {
        return width * height
    }
}
```

```swift
// Correct Example following OCP
protocol Shape {
    func calculateArea() -> Double
}

class Rectangle: Shape {
    var width: Double
    var height: Double

    func calculateArea() -> Double {
        return width * height
    }
}

class Circle: Shape {
    var radius: Double

    func calculateArea() -> Double {
        return Double.pi * radius * radius
    }
}
```

3. Liskov Substitution Principle (LSP):

“This principle states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.”

```
// Incorrect Example violating LSP
class Bird {
    func fly() {
        // Code to fly
    }
}

class Ostrich: Bird {
    override func fly() {
        // Ostrich cannot fly
    }
}
```

Here, the Ostrich class violates LSP as it doesn't behave like other birds.

Correct it to follow LSP:

3. Liskov Substitution Principle (LSP):

“This principle states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.”

```
// Incorrect Example violating LSP
class Bird {
    func fly() {
        // Code to fly
    }
}

class Ostrich: Bird {
    override func fly() {
        // Ostrich cannot fly
    }
}
```

```
// Correct Example following LSP
protocol Bird {
    func fly()
}

class Sparrow: Bird {
    func fly() {
        // Code for Sparrow to fly
    }
}

class Ostrich: Bird {
    func fly() {
        // Ostrich does not fly
    }
}
```

4. Interface Segregation Principle (ISP):

“This principle suggests that a class should not be forced to implement interfaces it doesn’t use. It’s better to have smaller, specific interfaces.”

```swift
// Incorrect Example violating ISP
protocol Machine {
    func print()
    func scan()
    func fax()
}

class Printer: Machine {
    func print() {
        // Code to print
    }

    func scan() {
        // Not relevant for Printer
    }

    func fax() {
        // Not relevant for Printer
    }
}
```

Here, Printer is forced to implement methods it doesn't need.

Segregate interfaces to follow ISP:

4. Interface Segregation Principle (ISP):

“This principle suggests that a class should not be forced to implement interfaces it doesn’t use. It’s better to have smaller, specific interfaces.”

```swift
// Incorrect Example violating ISP
protocol Machine {
    func print()
    func scan()
    func fax()
}

class Printer: Machine {
    func print() {
        // Code to print
    }

    func scan() {
        // Not relevant for Printer
    }

    func fax() {
        // Not relevant for Printer
    }
}
```

```swift
// Correct Example following ISP
protocol Printer {
    func print()
}

protocol Scanner {
    func scan()
}

protocol Fax {
    func fax()
}

class LaserPrinter: Printer {
    func print() {
        // Code to print using laser
    }
}

class OfficeScanner: Scanner {
    func scan() {
        // Code for scanning documents
    }
}

// Any class can adopt relevant interfaces as needed
```

5. Dependency Inversion Principle (DIP):

"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions."

```
// Incorrect Example violating DIP
class LightBulb {
    func turnOn() {
        // Code to turn on the light bulb
    }

    func turnOff() {
        // logic for turning off
    }
}

class Switch {
    let bulb = LightBulb()

    func toggle() {
        bulb.turnOn()
    }
}
```

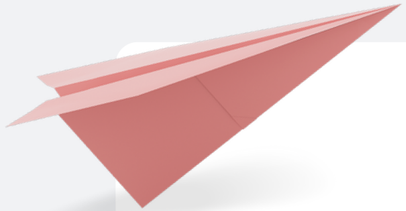
In this case, Switch depends directly on LightBulb. Use abstractions to invert the dependency:

5. Dependency Inversion Principle (DIP):

"High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions."

```swift
// Incorrect Example violating DIP
class LightBulb {
    func turnOn() {
        // Code to turn on the light bulb
    }

    func turnOff() {
        // logic for turning off
    }
}

class Switch {
    let bulb = LightBulb()

    func toggle() {
        bulb.turnOn()
    }
}
```

```swift
// Correct Example following DIP
protocol Switchable {
    func turnOn()
    func turnOff()
}

class LightBulb: Switchable {
    func turnOn() {
        // logic for turning on
    }

    func turnOff() {
        // logic for turning off
    }
}

class Switch {
    private let device: Switchable

    init(device: Switchable) {
        self.device = device
    }

    func operate() {
        device.turnOn()
    }
}
```



Do you have any questions?

Send it to us! We hope you learned something new.

shashidj206@gmail.com

@shashidj206

www.linkedin.com/in/iOSsj

+91 - 8951431160