SOLID

PRINCIPLES

SOLID is a set of 5 design principles that help write **clean, maintainable, and scalable code**.

by

SHUBAM GUPTA

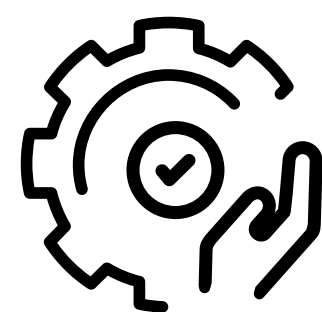
SOLID Principles in Swift (With Examples)

SOLID is a set of 5 design principles that help write clean, maintainable, and scalable code.

- 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)

🧠 Interview One-Liner

"SOLID principles help create **loosely coupled**, **testable**, and **maintainable code** by focusing on **responsibilities**, **abstractions**, and **extensibility**."



Single Responsibility Principle (SRP)

A class should have only one reason to change.

❌ Bad:

```swift
class UserManager {
    func saveUser() { }
    func sendEmail() { }
}
```

✅ Good:

```swift
class UserRepository {
    func saveUser() { }
}

class EmailService {
    func sendEmail() { }
}
```

If a component has multiple duties (like managing users and sending emails), it violates SRP.

Open/Closed Principle (OCP)

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

Swift example

❌ Bad

```swift
class DiscountCalculator {
    func discount(for type: String) -> Double {
        if type == "student" { return 10 }
        if type == "senior" { return 15 }
        return 0
    }
}
```

Swift example

✅ Good

```swift
protocol DiscountStrategy {
    func discount() -> Double
}

class StudentDiscount: DiscountStrategy {
    func discount() -> Double { 10 }
}

class SeniorDiscount: DiscountStrategy {
    func discount() -> Double { 15 }
}
```

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

Swift example

❌ Bad

```swift
class Bird {
    func fly() { }
}

class Penguin: Bird {
    override func fly() { fatalError() }
}
```

Swift example

✅ Good

```swift
protocol Flyable {
    func fly()
}

class Sparrow: Flyable {
    func fly() { }
}
```

❌ Problem it solves

Subclass breaks base-class behaviour.

Interface Segregation Principle (ISP)

Clients should not depend on methods they do not use.

Swift example

❌ Bad

```swift
protocol Worker {
    func work()
    func eat()
}
```

Swift example

✅ Good

```swift
protocol Workable {
    func work()
}

protocol Eatable {
    func eat()
}
```

Many small interfaces are better than one big interface. “Don’t force a class to implement methods it doesn’t need.”

Dependency Inversion Principle (DIP)

Depend on abstractions, not concrete implementations.

Swift example

❌ Bad

```swift
class OrderService {
    let api = APIService()
}
```

Swift example

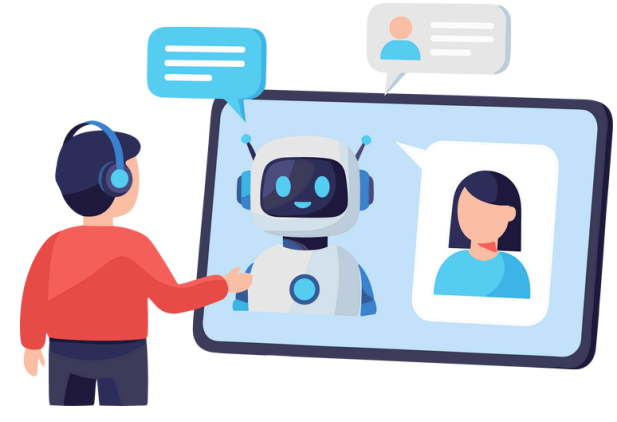
✅ Good

```swift
protocol APIServiceProtocol {
    func fetchData()
}

class OrderService {
    let api: APIServiceProtocol

    init(api: APIServiceProtocol) {
        self.api = api
    }
}
```

“High-level code should not depend on low-level code.”



Principle	Core Intent	What it protects
OCP	Add new behavior safely	**Stability**
LSP	Replace objects safely	**Correctness**
ISP	Don’t overforce contracts	**Simplicity**
DIP	Don’t hardcode dependencies	**Testability**

🔑 Why Protocols Show Up Everywhere

Because protocols give you:

- Abstraction → OCP
- Substitution → LSP
- Contract splitting → ISP
- Dependency decoupling → DIP

So yes, same tool, but different problem being solved.

Thank you

By Shubam Gupta