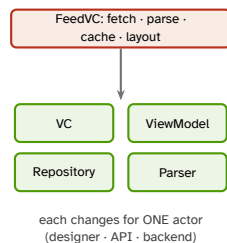
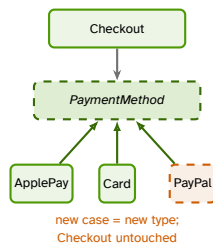
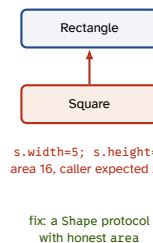
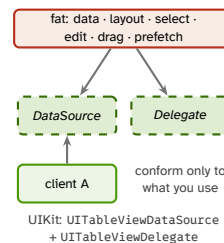
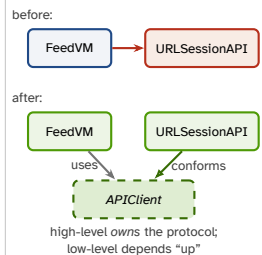


In one line: Single responsibility · Open/closed · Liskov substitution · Interface segregation · Dependency inversion — Robert C. Martin's rules for code whose parts can *change independently*. Interviewers want the **exact definition**, one **iOS example**, and **which pattern serves it**.

S one reason**O add, don't edit****L honest subtypes****I small protocols****D both → abstraction****How it works — definition · iOS example**

- **SRP** — “A module should have one, and only one, reason to change” (= answer to one actor). Not “one method”: a Money type with add/format is fine. Violation: the Massive View Controller; fix: VC · VM · Repository · Parser.
- **OCP** — “Software entities should be open for extension, but closed for modification” (Meyer 1988). Add behaviour by adding a type behind a protocol, not by editing a switch. Example: a new PaymentMethod; SwiftUI ViewModifier. It targets the volatile axes only; bug fixes still edit code.
- **LSP** — “Subtypes must be substitutable for their base types” — callers of the base must not notice (Martin, after Liskov 1987). A subtype may not **strengthen preconditions**, **weaken postconditions**, break invariants, or throw/fatalError/no-op where the base works. Swift smell: a read-only subclass whose setter crashes. Fine: UIButton used anywhere a UIView is.
- **ISP** — “Clients should not be forced to depend on methods they do not use.” It is judged from the caller's side: split fat protocols into role protocols. iOS: UITableViewDataSource vs UITableViewDelegate; compose with A & B; Codable = Encodable & Decodable.
- **DIP** — (a) “High-level modules should not depend on low-level modules. **Both** should depend on abstractions.” (b) “Abstractions should not depend on details. Details should depend on abstractions.” The *inversion*: the protocol belongs to the high-level (app/domain) side, and the low-level URLSession client conforms to it. Achieved by DI; tested with a MockAPIClient.

Example — Strategy = OCP + DIP

```
protocol PaymentMethod {           // the abstraction
    func pay(_ amount: Decimal) throws
}
struct ApplePay: PaymentMethod { func pay(_ a: Decimal) {} }
struct Card: PaymentMethod { func pay(_ a: Decimal) {} }
final class Checkout {             // high-level policy
    private let method: PaymentMethod // DIP: protocol, not Card
    init(method: PaymentMethod) { self.method = method } // DI
    func buy(_ total: Decimal) throws {
        try method.pay(total) // no switch on the type
    }
}
// OCP: a new case = a NEW type; Checkout is never edited
struct PayPal: PaymentMethod { func pay(_ a: Decimal) {} }
let checkout = Checkout(method: PayPal()) // swap at runtime
```

Which principle does a pattern serve?

Pattern	Serves	Why
Strategy	OCP + DIP	new algorithm = new type; client holds the protocol
Decorator	OCP	extend by wrapping, no 2 ^N subclasses
Adapter	DIP / ISP	client keeps its target protocol
Facade	ISP / SRP	one narrow surface over a subsystem
Composite	LSP	leaf and group used uniformly
Factory	DIP	one place touches the concrete type
Repository	DIP + SRP	domain depends on a data protocol
MVVM / MVP	SRP + DIP	logic out of the UIKit view

Interview traps

- ! **Strategy** → **ISP is wrong** (you said it, and described a Facade). Strategy serves **Open/Closed + Dependency Inversion**.
- ! **ISP** is about **clients**: “clients should not be forced to depend on methods they do not use” — not “methods describe behaviour”.
- ! **DIP** says **both** high- and low-level modules depend on the abstraction — not just “depend on abstractions”. Name who owns the protocol (the high-level side).
- ! SRP ≠ “one method”; OCP ≠ “never modify”.
- ! DI ≠ DIP: DI is a technique (pass deps in); DIP the principle. Injecting a concrete class is DI without DIP.
- ! Every pattern adds indirection — a protocol with one conformer “for the future” is over-engineering (rule of three, YAGNI).

Remember

Single reason · Open to add · Liskov no surprises · ISP clients pick · DIP both point at the protocol. Martin named the rules (~2000); Michael Feathers ordered them into “SOLID” — the order is a mnemonic, not a ranking.

Likely questions

1. Which SOLID does Strategy satisfy? — OCP + DIP.
2. ISP in UIKit? — UITableViewDataSource split from Delegate.
3. LSP violation in Swift? — override that throws/crashes where base works.
4. Who owns the protocol under DIP? — the high-level module.
5. Why is Massive VC bad? — many reasons to change: SRP.
6. OCP without subclassing in Swift? — protocols, generics, ViewModifier, closures.
7. Composition over inheritance — why? — no fragile base, no LSP traps; gives OCP + DIP.