04 · SOLID Principles

Flutter Practice Notes

Contents

04 · SOLID Principles

Introduction

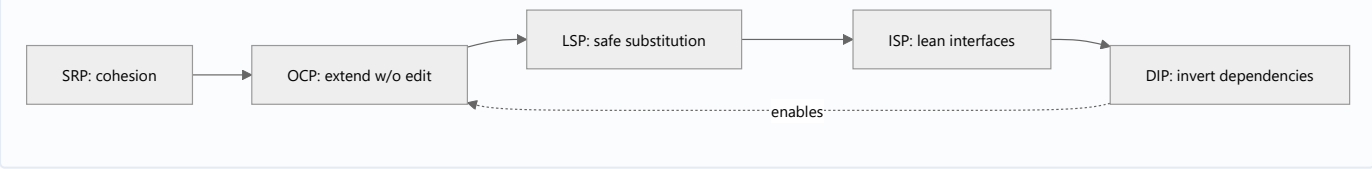
SOLID is five design principles that keep object-oriented code **flexible, testable, and maintainable** as it grows. They're not academic trivia — they're the reasoning senior engineers use to decide where to draw boundaries, what to abstract, and how to avoid the "change one thing, break five" trap. This module teaches each with a **wrong implementation, a corrected one, a Flutter example, and an enterprise example**.

Why this module exists

Most large codebases don't rot from bad syntax — they rot from bad *structure*: God classes, rigid `if/else` type checks, leaky abstractions, fat interfaces, and hard-wired dependencies. SOLID is the antidote. Interviewers ask about SOLID because it predicts whether you'll build systems that survive change.

The five principles

Letter	Principle	One-line intent	File
S	Single Responsibility	A class should have one reason to change	01_srp_single_responsibility.md
O	Open/Closed	Open for extension, closed for modification	02_ocp_open_closed.md
L	Liskov Substitution	Subtypes must be usable as their base	03_lsp_liskov_substitution.md
I	Interface Segregation	No client forced to depend on unused methods	04_isp_interface_segregation.md
D	Dependency Inversion	Depend on abstractions, not concretions	05_dip_dependency_inversion.md



The principles reinforce each other: DIP + polymorphism (Module 03) make OCP possible; ISP keeps LSP honest; SRP makes all of them easier.

Prerequisites

03 OOP — especially abstraction/interfaces, polymorphism, and composition. SOLID is applied OOP.

What you'll be able to do after this module

- Spot SOLID violations in a code review and name the fix.
- Refactor a God class/rigid hierarchy into cohesive, extensible components.
- Justify architectural boundaries (why an interface here, why inject there).
- Connect SOLID to design patterns (Module 05) and Clean Architecture (Module 40).

How each file is structured

Beyond the standard template, every principle file includes a ❌ **Violation** → ✅ **Refactor** pair, a **Flutter example**, and an **enterprise example**, plus the smells that signal the violation.

Capstone

SOLID refactor kata: Take a deliberately bad "order service" (a God class that parses, validates, saves, emails, logs, and formats — with `if (type == ...)` branching and `new` dependencies everywhere) and refactor it to satisfy all five principles. Provided in 05_dip_dependency_inversion.md as the culminating mini-project.

Summary

SOLID is the bridge from "knows OOP syntax" to "designs maintainable systems." Read the files in order (they build on each other), do each refactor, and you'll internalize the judgment behind clean architecture and design patterns.

S — Single Responsibility Principle (SRP)

> A class should have exactly one reason to change — one responsibility, one axis of change, one owner of that concern.

Introduction

SRP says each class (or function/module) should do **one thing** and have **one reason to change**. "Reason to change" means a distinct *actor* or concern: persistence, formatting, business rules, networking. When a class mixes concerns, a change to one drags in risk for the others.

Why this concept exists

God classes are the most common cause of unmaintainable code. When persistence, validation, UI formatting, and networking live in one class, every requirement touching any of them edits the same file — merge conflicts, fragile tests, and ripple bugs. SRP splits concerns so each changes independently.

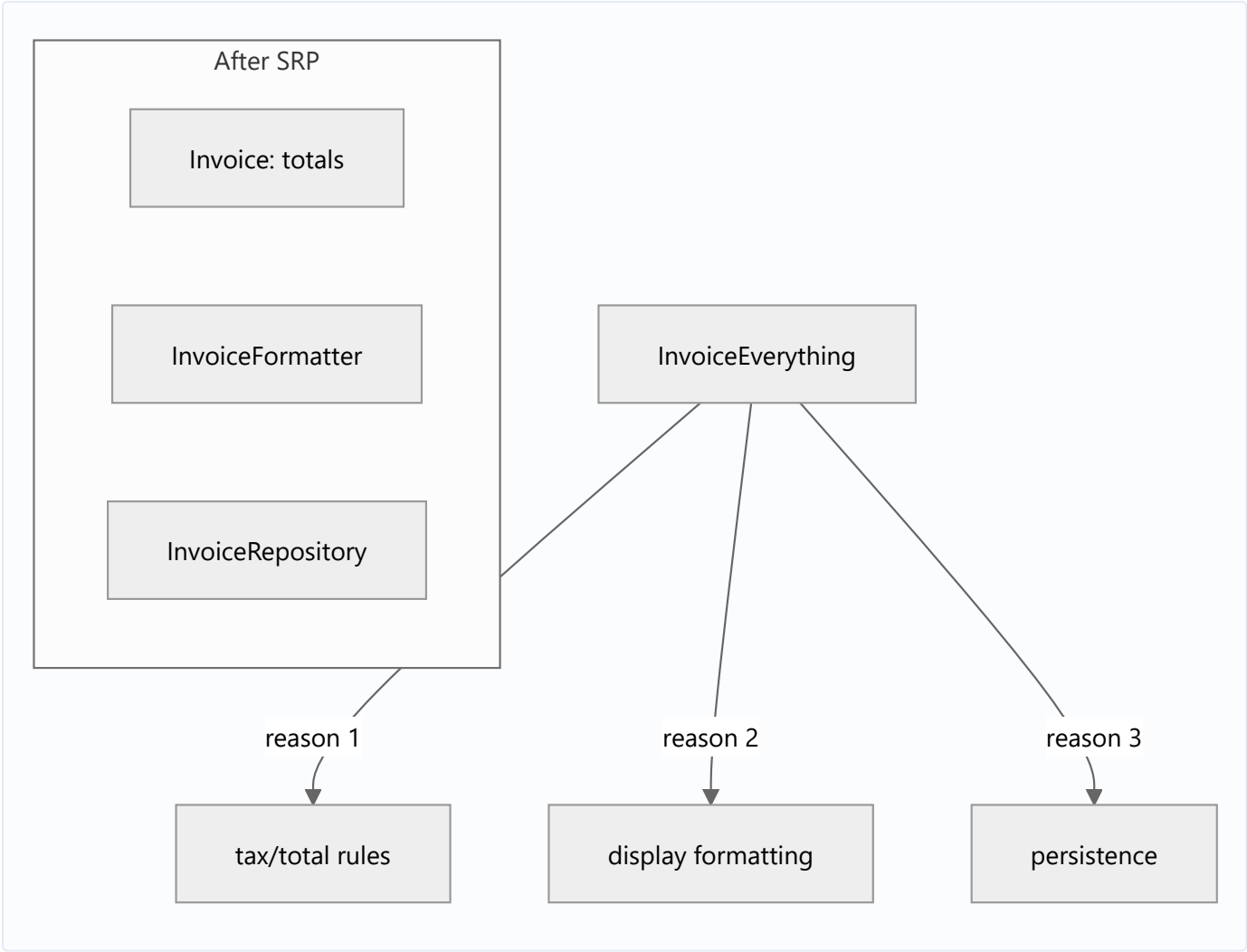
Real-world analogy

A restaurant separates **chef**, **waiter**, and **cashier**. If one person did all three, changing the payment system would risk breaking how food is cooked. Separate roles = changes stay contained. A class that "cooks, serves, and bills" is a God class.

Problem Statement

An `Invoice` class computes totals, formats them for display, and saves to a database. A DB change forces edits to the same class that owns tax rules and formatting — three reasons to change in one place. You'll split it into cohesive units.

Internal Working



- Identify the **axes of change** (who asks for changes: finance, UI, ops).
- Give each axis its own class with a focused API.
- Coordinate them via composition/injection (03 · composition), not by merging.

Memory Representation

Not applicable — SRP is a structural design principle, not a runtime data concern.

Compiler Behavior

Not applicable directly, though smaller cohesive classes give the analyzer clearer types and dead-code detection.

Runtime Behavior

Not applicable — behavior is unchanged by refactoring; *changeability* improves.

Flutter Engine Behavior

Not applicable. (But Flutter's widget/build vs state vs logic separation is SRP in practice — keep `build()` about UI, not business rules.)

Dart VM Behavior

Not applicable.

Examples

❌ Violation — one class, three reasons to change

```
class InvoiceEverything {
  final List<double> items;
  InvoiceEverything(this.items);

  double total() => items.fold(0, (s, x) => s + x) * 1.18; // tax rule (finance)

  String toHtml() => '<b>${total().toStringAsFixed(2)}</b>'; // formatting (UI)

  void saveToDb() {
    // SQL persistence (ops/infra) — DB change edits THIS class too
    print('INSERT INTO invoices ... ${total()}');
  }
}
```

✅ Refactor — one responsibility each

```
class Invoice {
  final List<double> items;
  const Invoice(this.items);
  double get subtotal => items.fold(0, (s, x) => s + x);
  double total({double taxRate = 0.18}) => subtotal * (1 + taxRate); // finance only
}

class InvoiceFormatter {
  String toHtml(Invoice inv) =>
      '<b>${inv.total().toStringAsFixed(2)}</b>'; // UI only
}

abstract interface class InvoiceRepository {
  Future<void> save(Invoice inv);
}
class SqlInvoiceRepository implements InvoiceRepository {
  @override
  Future<void> save(Invoice inv) async => print('INSERT ... ${inv.total()}');
}

Future<void> main() async {
  const inv = Invoice([100, 200]);
  print(InvoiceFormatter().toHtml(inv)); // <b>354.00</b>
  await SqlInvoiceRepository().save(inv);
}
```

Flutter example

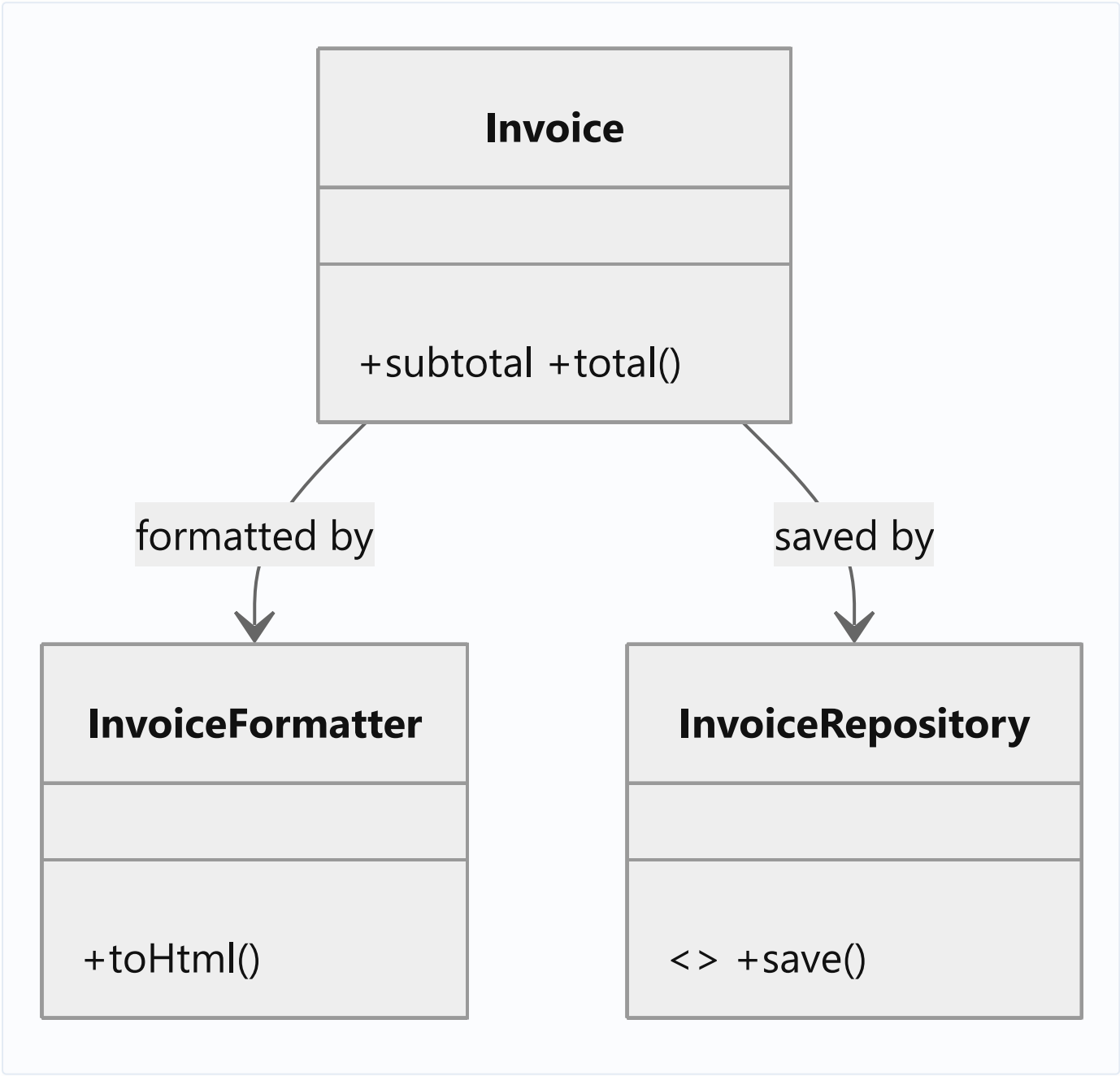
```
// ❌ Widget doing networking + parsing + UI (three reasons to change)
// class UserScreen extends StatelessWidget { ...http.get, jsonDecode, build... }

// ✅ Separate: Repository (data), ViewModel/Cubit (state/logic), Widget (UI only)
// UserRepository.fetch() -> UserViewModel exposes state -> UserScreen just renders it
// A UI change never risks the parsing logic, and vice versa. (See Modules 11, 40.)
```

Enterprise example

In a payments system, split `PaymentProcessor` into: `PaymentValidator` (rules), `PaymentGateway` (network), `PaymentRepository` (persistence), `ReceiptFormatter` (documents), `PaymentLogger` (observability). A gateway swap (Razorpay→Stripe) touches only `PaymentGateway`.

Diagrams



Common Mistakes

Mistake	Why	Fix
God class ("Manager"/"Helper"/"Utils" that does everything)	Many reasons to change	Split by concern/actor
Widgets containing business + network logic	Untestable, coupled	Extract repository + view model
Over-splitting into trivial classes	Ceremony, indirection overload	One responsibility ≠ one method — group cohesive behavior
"Utility" dumping grounds	Low cohesion	Group by domain, not by "misc"

Best Practices

- Ask: **"What are this class's reasons to change?"** More than one → split.
- Name classes by their single responsibility (`InvoiceFormatter`, not `InvoiceManager`).
- Keep `build()` methods about UI; push logic to view models/use cases.
- Balance: cohesion, not fragmentation — related behavior stays together.

Performance

Neutral. SRP is about maintainability; runtime cost is unchanged (a tiny extra indirection at most).

Advantages / Disadvantages

- **+** Localized changes, focused tests, reuse, parallel teamwork, clearer names.
- **−** More classes/files; over-application creates needless indirection.

Interview Questions

1. 🟢 **State SRP.** — A class should have one responsibility and therefore one reason to change.
2. 🟢 **What's a "reason to change"?** — A distinct concern/actor requesting changes (finance rules vs UI formatting vs persistence).
3. 🟡 **How does SRP improve testability?** — Small, single-purpose units have fewer dependencies and simpler tests; you can test rules without a database or UI.
4. 🟡 **How does SRP apply to Flutter widgets?** — Keep widgets rendering-only; move networking/parsing/business logic to repositories and view models so UI and logic change independently.
5. 🟡 **Can SRP be over-applied?** — Yes; splitting cohesive behavior into anemic one-method classes adds indirection without benefit. Group by responsibility, not by line count.

6. 🔴 **How does SRP relate to cohesion/coupling?** — SRP maximizes cohesion (a class's parts belong together) and reduces coupling (fewer cross-concern dependencies).
7. 🔴 **How do you detect an SRP violation in review?** — Look for "and" in the class's description, multiple unrelated dependencies (DB + HTTP + UI), or churn from unrelated features touching the same file.

Senior Engineer Tips

- The `Manager` / `Helper` / `Util` suffix is a code smell for hidden multi-responsibility — name the actual responsibility.
- Use git churn as evidence: files changed by many unrelated PRs often violate SRP.
- SRP at the module/package level (feature-first) matters as much as at the class level (Modules 44, 45).

Architect Perspective

SRP is the seed of layered/clean architecture: separate presentation, domain, and data so each evolves on its own axis. Applied at package granularity it enables independent deployability and team ownership. Violations compound — a God class becomes a God module becomes an unshippable monolith.

Summary

- One class, one responsibility, one reason to change.
- Split God classes by concern; compose the pieces; keep widgets UI-only.
- Balance cohesion vs fragmentation; name by responsibility.

Revision Notes

- SRP = one reason to change (one actor/concern).
- Smell: God class, `Manager` / `Util`, DB+HTTP+UI together, "and" in the description.
- Fix: split by concern, compose/inject.
- Flutter: widget=UI, repo=data, viewmodel=logic.

Practice Questions

1. List the reasons to change in a class that fetches, parses, caches, and renders data.
2. When is splitting a class *over*-applying SRP?
3. How does SRP make unit tests simpler?

Coding Questions

1. Refactor a `ReportManager` (compute + format + email + save) into four cohesive classes.
2. Extract business logic out of a Flutter widget into a testable view model.
3. Identify and split the responsibilities of a `UserUtils` grab-bag.

Mini Project

Invoice module refactor (pure Dart): Start from an `InvoiceEverything` God class; refactor into `Invoice` (rules), `InvoiceFormatter` (multiple formats), and `InvoiceRepository` (interface + impl). Add tests for each in isolation (rules without a DB, formatting without persistence). Acceptance: each class has one responsibility; tests need no unrelated dependencies; `dart analyze` clean.

O — Open/Closed Principle (OCP)

> Software entities should be **open for extension but closed for modification** — add new behavior by adding new code, not by editing tested, working code.

Introduction

OCP says you should be able to extend a system's behavior **without changing** its existing source. In practice: replace growing `if/else` / `switch`-on-type chains with **polymorphism** or **strategies**, so adding a case means adding a class, not editing a battle-tested method.

Why this concept exists

Every edit to working code risks regressions and re-testing. When a single function must change every time a new "type" appears, it becomes a bug magnet and a merge-conflict hotspot. OCP isolates variation behind an abstraction so new variants plug in without touching stable code.

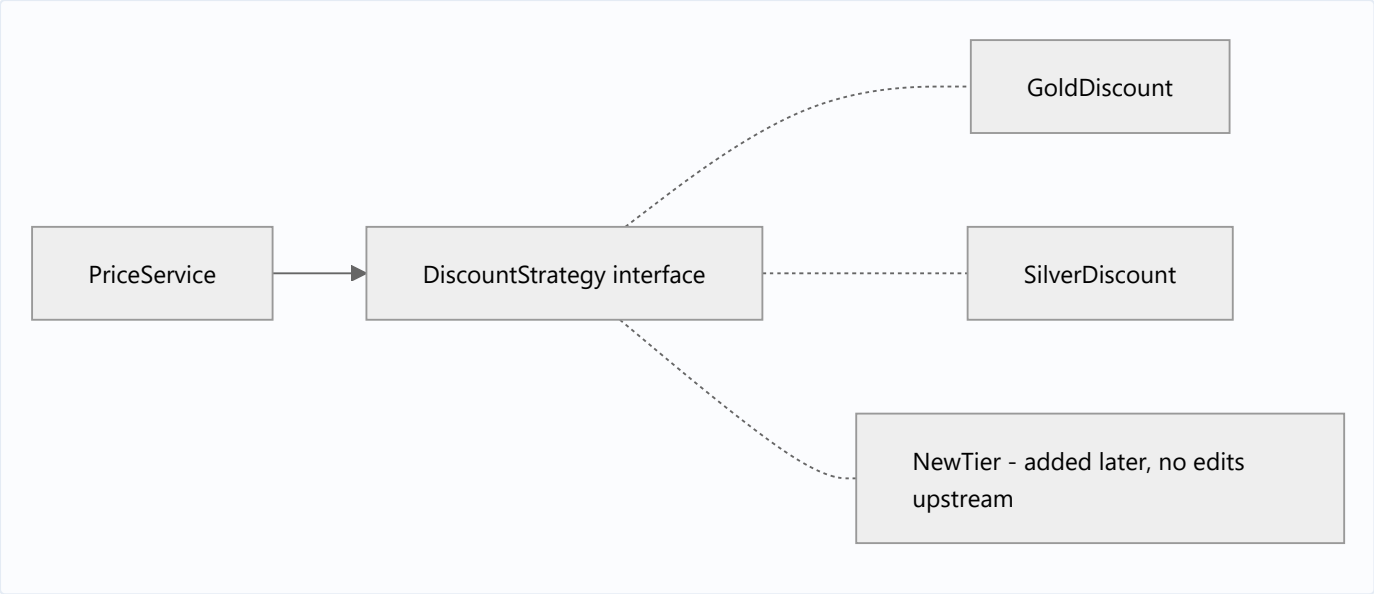
Real-world analogy

A **power strip**: to add a new appliance you plug it into an open socket — you don't rewire the wall. The socket (abstraction) is the stable, closed part; appliances (implementations) are the open, extensible part.

Problem Statement

A `DiscountCalculator` uses `if (customer.type == 'gold') ... else if ('silver') ...`. Every new tier edits (and risks breaking) this method. You'll refactor to a strategy abstraction so new tiers are new classes.

Internal Working



- Define a **stable abstraction** (interface/abstract class) for the varying behavior.
- Put each variation in its own implementation.
- The consumer depends on the abstraction and never changes when variants are added.
- Mechanisms: polymorphism (03 · polymorphism), Strategy/Factory patterns (Module 05), and DIP (05_dip_dependency_inversion.md).

Memory Representation

Not applicable — a structural principle.

Compiler Behavior

- With `sealed` types, an exhaustive `switch` gives a *controlled* form of OCP: adding a variant forces you to handle it (compile error) — appropriate for **closed** sets. For **open** sets (third parties add variants), use open interfaces.

Runtime Behavior

- New strategies dispatch polymorphically; existing code paths are untouched and don't need re-testing.

Flutter Engine Behavior

Not applicable. (Flutter's widget composition is OCP-friendly: extend UIs by composing new widgets, not editing existing ones.)

Dart VM Behavior

Not applicable beyond normal virtual dispatch.

Examples

❌ Violation — edit the method for every new type

```
class DiscountCalculatorBad {
  double discount(String tier, double amount) {
    if (tier == 'gold') return amount * 0.20;
    else if (tier == 'silver') return amount * 0.10;
    else if (tier == 'bronze') return amount * 0.05;
    // adding 'platinum' means EDITING this tested method — OCP violation
    else return 0;
  }
}
```

✅ Refactor — extend by adding a class

```
abstract interface class DiscountStrategy {
  double discount(double amount);
}

class GoldDiscount implements DiscountStrategy {
  @override
  double discount(double amount) => amount * 0.20;
}
class SilverDiscount implements DiscountStrategy {
  @override
  double discount(double amount) => amount * 0.10;
}
// New tier = NEW class, zero edits to PriceService:
class PlatinumDiscount implements DiscountStrategy {
  @override
  double discount(double amount) => amount * 0.30;
}

class PriceService {
  final DiscountStrategy strategy; // depends on abstraction
  PriceService(this.strategy);
  double finalPrice(double amount) => amount - strategy.discount(amount);
}

void main() {
  print(PriceService(GoldDiscount()).finalPrice(100));     // 80.0
  print(PriceService(PlatinumDiscount()).finalPrice(100)); // 70.0 — added w/o edits
}
```

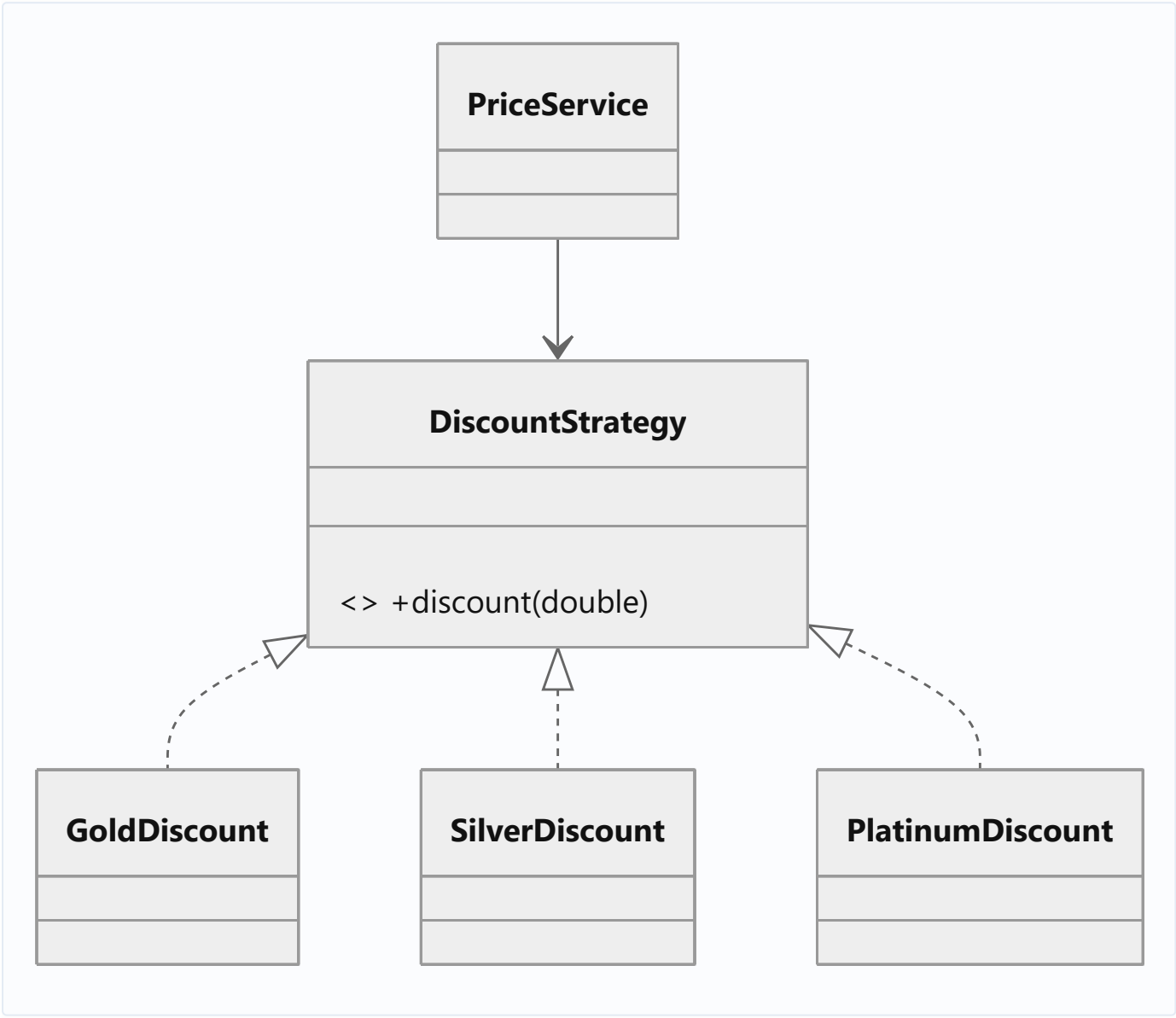
Flutter example

```
// ❌ A widget with switch(type) building different cards, edited per new type.
// ✅ Define an abstract CardBuilder / pass a WidgetBuilder per type via a registry/map:
//    final builders = <CardType, Widget Function(Data)>{ ... };
//    Widget build(...) => builders[type]!(data);
// Adding a new card type registers a new builder — no edits to the rendering widget.
```

Enterprise example

A notification system with `NotificationChannel` (email/SMS/push/WhatsApp). Adding a new channel (e.g., Slack) is a new `SlackChannel` class registered with the dispatcher; the dispatcher, tests, and existing channels are untouched — critical when the dispatcher is high-risk shared code.

Diagrams



Common Mistakes

Mistake	Why	Fix
`switch` / `if` -on-type chains that grow	Edit-per-variant (violates OCP)	Strategy/polymorphism
Abstracting *everything* preemptively	Speculative generality (YAGNI)	Abstract when a *second* variant appears
Using `sealed` for an externally-extensible set	Third parties can't add variants	Open interface for open sets
Editing shared code for each feature flag	Risky, conflict-prone	Register behaviors via a map/DI

Best Practices

- Introduce the abstraction when the **second** variant arrives (rule of three), not before.
- Prefer composition/strategy + registries/maps over type switches for open sets.
- Use `sealed` + exhaustive `switch` for **closed** sets where compiler-enforced completeness is the goal.
- Keep the abstraction stable; variants carry the change.

Performance

Neutral; virtual dispatch/registry lookup is cheap.

Advantages / Disadvantages

- **+** Add features without touching stable code; fewer regressions; parallel work; smaller diffs.
- **−** More types/indirection; premature abstraction hurts (speculative generality).

Interview Questions

1. 🟢 **State OCP.** — Entities should be open for extension but closed for modification: add behavior via new code, not by editing existing code.
2. 🟢 **What's the classic OCP smell?** — A growing `if/else` / `switch` on a type/enum that must be edited for every new variant.
3. 🟡 **How do you achieve OCP?** — Depend on a stable abstraction (interface); put each variation in its own implementation; add variants without editing the consumer (Strategy/Factory + DIP).
4. 🟡 **When is a `sealed` `switch` compatible with OCP?** — For **closed** sets: the compiler enforces handling every variant. For **open** sets (external extension), use open interfaces instead.
5. 🟡 **Doesn't OCP contradict "don't over-abstract"?** — No: apply it when a real second variant appears (rule of three); premature abstraction (speculative generality) is the failure

mode.

6. 🔴 **How do OCP and DIP relate?** — DIP (depend on abstractions) is the mechanism that makes OCP achievable; polymorphism dispatches to the new implementation.
7. 🔴 **Give an enterprise OCP win.** — A payment/notification dispatcher where new providers/channels are added as classes registered via DI, leaving the high-risk dispatcher untouched.

Senior Engineer Tips

- Watch for the *third* `else if` — that's usually the signal to extract a strategy.
- A `Map<Type, Builder>` registry is often the simplest OCP tool in Flutter (card builders, route factories, handlers).
- Balance: don't invert everything. Abstractions have carrying cost; introduce them where variation is real or clearly imminent.

Architect Perspective

OCP is what lets teams add features fast and safely: plugins, provider adapters, feature modules that register capabilities without editing the core. It underpins extensible architectures (Strategy, plugin systems, adapter layers) and reduces the blast radius of change — a direct driver of delivery velocity at scale.

Summary

- Open for extension, closed for modification: add variants as new classes behind a stable abstraction.
- Replace type-switch chains with polymorphism/strategy + registries; use `sealed` for closed sets.
- Introduce abstractions when real variation appears, not speculatively.

Revision Notes

- OCP: extend without editing existing code.
- Smell: growing `switch` / `if`-on-type.
- Fix: interface + implementations + DI/registry (Strategy/Factory).
- `sealed` +exhaustive `switch` = closed-set OCP; open interface = open-set.
- Apply on the 2nd–3rd variant (avoid speculative generality).

Practice Questions

1. Why does adding a `PlatinumDiscount` require no edits to `PriceService`?
2. When is a type `switch` acceptable (and even preferable)?
3. How does OCP reduce regression risk?

Coding Questions

1. Refactor a `ShippingCostCalculator` with `if(country==...)` into a strategy per region.
2. Build a `Map<EventType, Handler>` registry so new event types register handlers without editing the dispatcher.
3. Implement an open `ExportFormat` abstraction (CSV/JSON/PDF) and add a new format with zero edits upstream.

Mini Project

Pluggable discount engine (pure Dart): Implement a `DiscountStrategy` abstraction with several tiers and a `PriceService`, plus a registry that resolves strategies by tier. Add a brand-new tier and prove no existing file changed. Add tests. Acceptance: no type-switch in `PriceService`; new variant added without editing consumer/tests; `dart analyze` clean.

L — Liskov Substitution Principle (LSP)

> Subtypes must be substitutable for their base type without breaking correctness — if `S` is a subtype of `T`, code written against `T` must work when handed an `S`.

Introduction

LSP (Barbara Liskov, 1987) constrains inheritance: a subclass must honor the **behavioral contract** of its superclass. Overriding is allowed, but not in ways that surprise callers — no strengthening preconditions, no weakening postconditions, no throwing where the base wouldn't, no violating invariants. Violations make polymorphism unsafe.

Why this concept exists

Polymorphism (03 · polymorphism) only works if a base reference behaves predictably regardless of the concrete subtype. If a subtype breaks the base's promises, callers must special-case it (`if (x is Weird)`) — which destroys the extensibility inheritance was supposed to provide.

Real-world analogy

If a "coffee machine" contract says "press brew → get coffee," any machine you swap in must honor it. A machine that dispenses tea, or explodes when the tank is half-full (a stricter precondition the base didn't have), is **not substitutable** — anyone relying on the contract breaks.

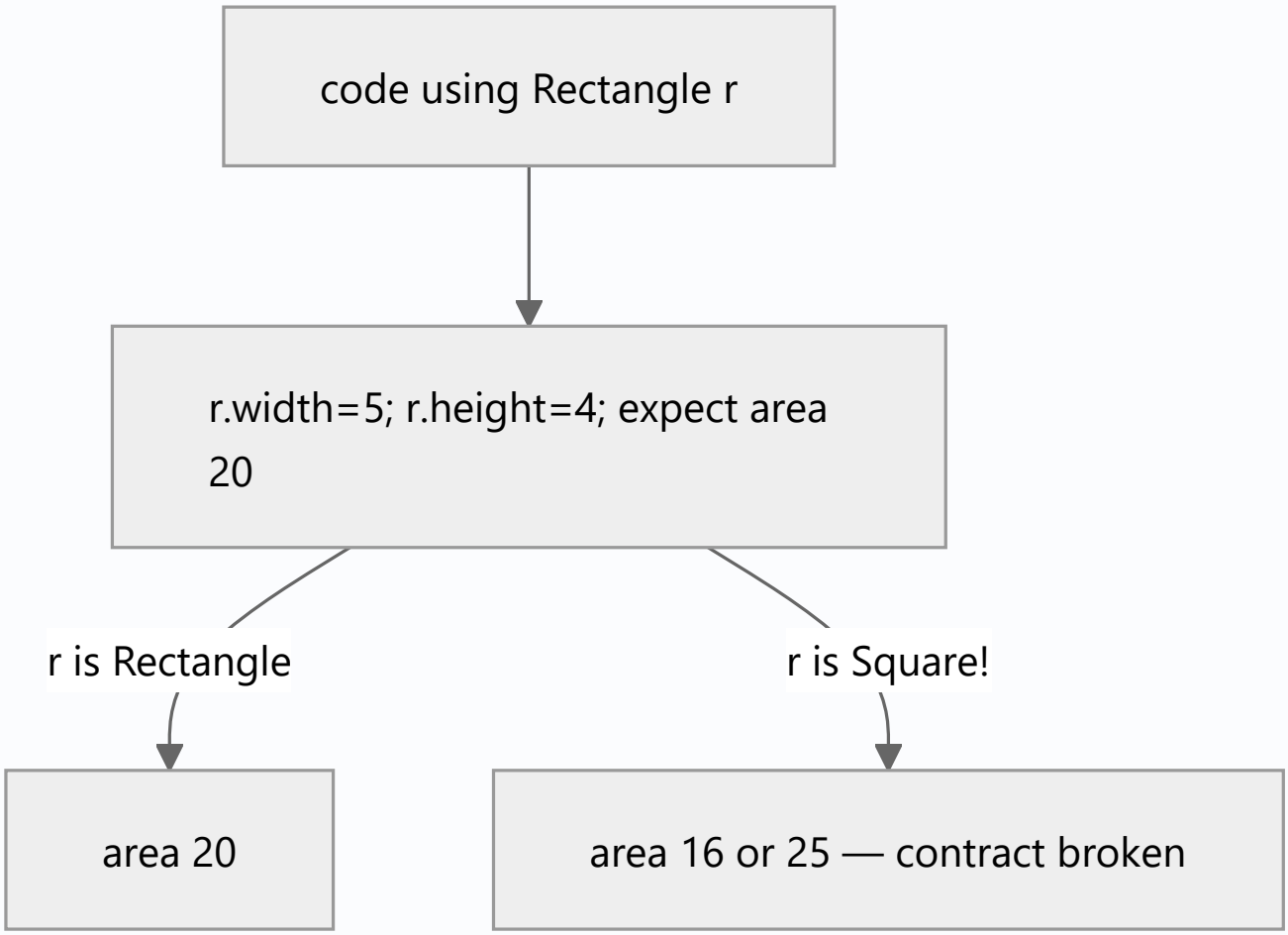
Problem Statement

The classic trap: `Square extends Rectangle`. A `Rectangle` lets you set width and height independently; a `Square` can't. Code that sets width then height and asserts area breaks for `Square`. You'll see the violation and fix it by rethinking the hierarchy.

Internal Working

LSP requires overrides to obey the base's contract:

Rule	Meaning
Preconditions	Not **stronger** in the subtype (don't demand more than the base)
Postconditions	Not **weaker** in the subtype (don't promise less)
Invariants	Preserved by the subtype
Exceptions	Don't throw new types the base's callers don't expect
History (mutation)	Don't allow state changes the base forbids



Memory Representation

Not applicable — a behavioral contract principle.

Compiler Behavior

- The compiler enforces *type* substitutability (signatures), **not behavioral** substitutability — LSP is about semantics the compiler can't check. `covariant` narrowing can even let unsafe overrides compile.

Runtime Behavior

- Violations manifest as wrong results or unexpected exceptions when a subtype is used through a base reference — often far from the subtype's definition, making them hard to debug.

Flutter Engine Behavior

Not applicable. (But a custom `ScrollController` / `Listenable` subclass that breaks the base contract will misbehave inside framework code that assumes the contract.)

Dart VM Behavior

Not applicable.

Examples

❌ Violation — Square is not a (behavioral) Rectangle

```
class Rectangle {
  int width, height;
  Rectangle(this.width, this.height);
  int area() => width * height;
}

class Square extends Rectangle {
  Square(int side) : super(side, side);
  // Force both dimensions equal — breaks Rectangle's contract:
  @override
  set width(int w) { super.width = w; super.height = w; }
  @override
  set height(int h) { super.width = h; super.height = h; }
}

void resizeAndCheck(Rectangle r) {
  r.width = 5;
  r.height = 4;
  assert(r.area() == 20); // holds for Rectangle...
}

void main() {
  resizeAndCheck(Rectangle(1, 1)); // ok: area 20
  resizeAndCheck(Square(1));       // ❌ area 16 — LSP violated, assert fails
}
```

✅ Refactor — model the real contract (no false is-a)

```
abstract class Shape {
  int area();
}

class Rectangle implements Shape {
  final int width, height; // immutable -> no independent-setter trap
  const Rectangle(this.width, this.height);
  @override
  int area() => width * height;
}

class Square implements Shape {
  final int side;
  const Square(this.side);
  @override
  int area() => side * side;
}

int totalArea(List<Shape> shapes) =>
    shapes.fold(0, (s, sh) => s + sh.area());

void main() {
  print(totalArea([const Rectangle(5, 4), const Square(3)])); // 20 + 9 = 29
  // No subtype pretends to be another; each honors the Shape contract.
}
```

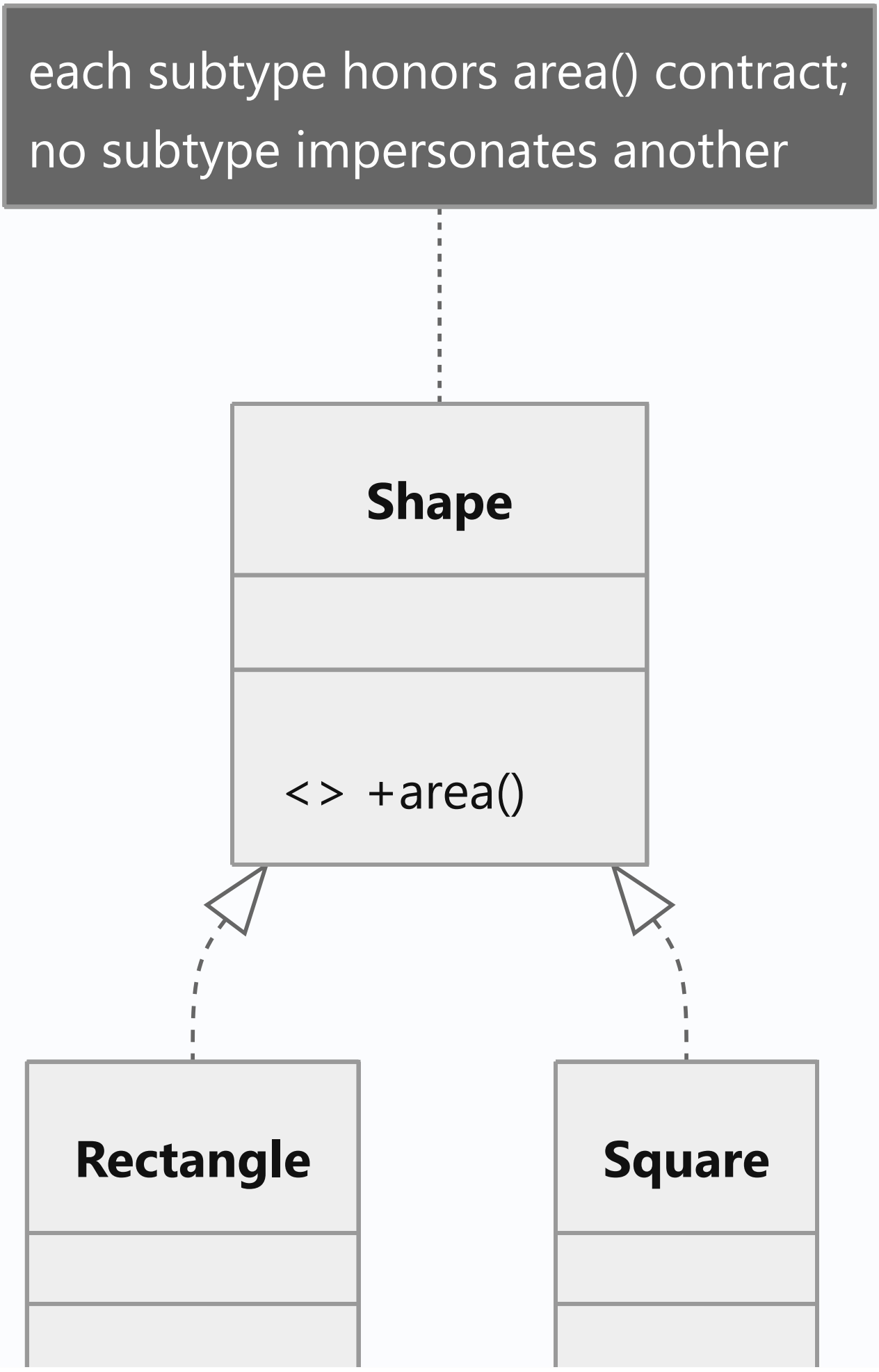
Flutter example

```
// ❌ A ReadOnlyList that extends List but throws on add() strengthens no-op into a
//    surprise UnsupportedError -> callers treating it as a List break at runtime.
// ✅ Expose an Iterable / UnmodifiableListView, or a separate read-only type,
//    so the declared type already communicates the contract (no false substitutability).
```

Enterprise example

A `PaymentGateway` base promises `charge()` never throws for a validated request (returns a failure result). A `LegacyGateway` subclass that *throws* on certain currencies violates LSP — every caller must now wrap it specially. Fix: make the base contract return a `Result` and have all gateways honor it uniformly.

Diagrams


each subtype honors area() contract;
no subtype impersonates another
Shape
<> +area()
Rectangle
Square

Common Mistakes

Mistake	Why	Fix
`Square extends Rectangle` with coupled setters	Breaks independent-dimension contract	Separate types under a `Shape` abstraction
Override throwing `UnsupportedError`	Weakens/breaks the base contract	Don't inherit a capability you can't fulfill
Strengthening preconditions in a subtype	Callers passing base-valid input fail	Keep preconditions ≤ base's
Returning `null` /less where base returns a value	Weakens postcondition	Honor the base's guarantees
`is` -checks to special-case a subtype	Symptom of an LSP violation	Fix the hierarchy so it's unnecessary

Best Practices

- Design the **contract** (pre/postconditions, invariants, exceptions) explicitly; subtypes must obey it.
- Prefer **immutability** to avoid mutation-based traps (the Rectangle/Square setter issue vanishes).
- If a subtype can't fulfill a capability, it **isn't** a subtype — use composition or a different abstraction.
- Watch for `UnsupportedError` overrides — a red flag.

Performance

Neutral; LSP is about correctness/design.

Advantages / Disadvantages

- **+** Safe polymorphism, no special-casing, reliable extensibility, honest hierarchies.
- **−** Requires disciplined contract thinking; sometimes forces flatter hierarchies (which is usually good).

Interview Questions

1. 🟢 **State LSP.** — Subtypes must be usable anywhere their base is expected without breaking correctness.

2. 🟢 **Give the classic LSP violation.** — `Square extends Rectangle`: coupling width/height setters breaks code relying on independent dimensions.
3. 🟡 **What rules must overrides obey?** — No stronger preconditions, no weaker postconditions, preserve invariants, don't throw unexpected exceptions, don't forbid base-allowed state changes.
4. 🟡 **Why can't the compiler enforce LSP?** — It checks type/signature substitutability, not *behavioral* semantics; an override can type-check yet violate the contract.
5. 🟡 **What's a runtime symptom of an LSP violation?** — Wrong results or surprise exceptions when a subtype flows through base-typed code; callers adding `is`-checks to compensate.
6. 🔴 **How does immutability help with LSP?** — It removes mutation-based contract traps (no independent setters to violate), making substitution safe by construction.
7. 🔴 **How do LSP and polymorphism/OCP connect?** — Polymorphism and OCP assume subtypes are substitutable; an LSP violation forces special-casing, breaking both.

Senior Engineer Tips

- If you're tempted to override a method to throw `UnsupportedError`, stop — the inheritance is wrong; use composition or split the abstraction.
- Encode contracts in types where possible (return `Result`, expose read-only interfaces) so violations are hard to write.
- A proliferation of `is` / `as` downcasts is often an LSP smell in disguise.

Architect Perspective

LSP keeps abstraction boundaries trustworthy: repositories, gateways, and strategies can be swapped (real/fake/provider-A/provider-B) only if every implementation honors the contract. This trust is what makes testing-with-fakes, provider migration, and plugin ecosystems safe at scale. Contract-first design (interfaces with documented guarantees) is the enterprise enforcement of LSP.

Summary

- Subtypes must honor the base's behavioral contract (pre/postconditions, invariants, exceptions).
- The compiler checks types, not behavior — LSP is a design discipline.
- Prefer immutability and honest abstractions; if a subtype can't fulfill the contract, it isn't one.

Revision Notes

- LSP: subtype substitutable for base without breaking callers.

- No stronger preconditions / weaker postconditions / broken invariants / surprise exceptions.
- Classic trap: Square/Rectangle setters. Fix: separate types + immutability.
- `UnsupportedError` override / `is`-special-casing = LSP smell. Compiler can't enforce it.

Practice Questions

1. Why does `resizeAndCheck(Square(1))` fail while `Rectangle` passes?
2. Give a non-geometry LSP violation from a service contract.
3. How does returning `Result` instead of throwing help honor a contract?

Coding Questions

1. Fix the Square/Rectangle hierarchy using a `Shape` abstraction + immutability.
2. Find and fix an LSP violation in a `Bird` / `Penguin` (can't `fly()`) hierarchy.
3. Refactor a gateway that throws on some inputs to a `Result`-returning contract all impls honor.

Mini Project

Contract-safe gateway (pure Dart): Define a `PaymentGateway` interface whose `charge()` returns a `Result` (never throws for validated input); implement `CardGateway`, `UpiGateway`, and a `LegacyGateway` adapter that *wraps* a throwing legacy API to honor the contract. Write tests substituting each gateway through the interface. Acceptance: all implementations honor the contract; no caller special-cases a subtype; `dart analyze` clean.

I — Interface Segregation Principle (ISP)

> No client should be forced to depend on methods it does not use — prefer many small, role-specific interfaces over one fat, do-everything interface.

Introduction

ISP says interfaces should be **narrow and client-focused**. A "fat" interface forces implementers to provide (and clients to depend on) members irrelevant to them, causing empty/`throw`-ing implementations and needless coupling. Split it into cohesive role interfaces.

Why this concept exists

Fat interfaces couple unrelated concerns. When one interface declares `print`, `scan`, `fax`, and `staple`, a simple printer must implement (or stub) scanning and faxing, and any change to the fax API recompiles/affects every implementer — even printers that can't fax. ISP decouples clients from capabilities they don't need.

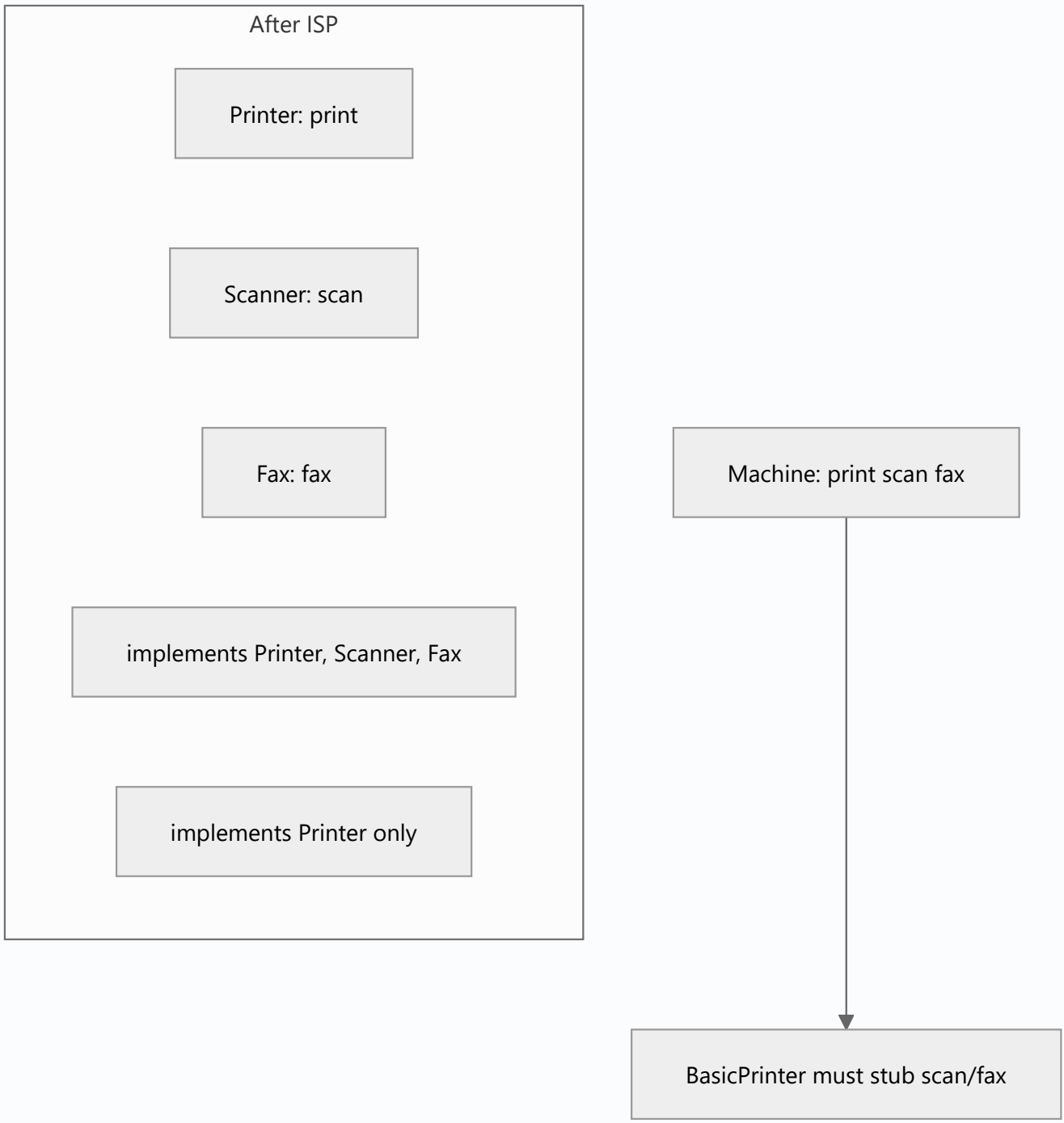
Real-world analogy

A **Swiss-army-knife job description** ("must cook, drive, code, and do surgery") forces every hire to fake most skills. Split roles (chef, driver, developer, surgeon) so each person implements only what they actually do — and changing the surgery requirements doesn't affect the chef.

Problem Statement

A `Worker` interface has `work()` and `eat()`. A `RobotWorker` can `work()` but doesn't `eat()` — forcing an empty/throwing `eat()`. And a `MultiFunctionDevice` interface forces a basic printer to stub `scan()` / `fax()`. You'll segregate into role interfaces.

Internal Working



- Break a fat interface into **role interfaces** (`Printer` , `Scanner` , `Fax`).
- Each class implements **only** the roles it fulfills; multi-capable classes implement several.
- Clients depend on the **narrowest** interface they need — often defined at the **consumer** side.
- Complements LSP: small honest interfaces are easy to satisfy without contract-breaking stubs.

Memory Representation

Not applicable — a structural/interface-design principle.

Compiler Behavior

- Implementing a fat interface forces providing every member (Dart `implements` requires all) — the compiler makes the pain visible as stubs.
- Segregated interfaces let the analyzer track precise capabilities per type.

Runtime Behavior

- No forced `UnsupportedError` stubs → fewer runtime surprises (also an LSP win — 03_lsp_liskov_substitution.md).

Flutter Engine Behavior

Not applicable. (Flutter favors small, focused contracts — `Listenable`, `ValueListenable<T>`, `Comparable<T>` — rather than one giant interface.)

Dart VM Behavior

Not applicable.

Examples

❌ Violation — fat interface forces useless members

```
abstract interface class Worker {
  void work();
  void eat();
}

class HumanWorker implements Worker {
  @override
  void work() => print('working');
  @override
  void eat() => print('lunch');
}

class RobotWorker implements Worker {
  @override
  void work() => print('working 24/7');
  @override
  void eat() => throw UnsupportedError('robots do not eat'); // ❌ forced stub
}
```

✅ Refactor — role interfaces

```
abstract interface class Workable {
  void work();
}
abstract interface class Feedable {
  void eat();
}

class HumanWorker implements Workable, Feedable {
  @override
  void work() => print('working');
  @override
  void eat() => print('lunch');
}

class RobotWorker implements Workable { // only what it can do
  @override
```

```
  void work() => print('working 24/7');
}

// Clients depend on the narrow role they need:
void runShift(Iterable<Workable> crew) {
  for (final w in crew) {
    w.work();
  }
}
void lunchBreak(Iterable<Feedable> people) {
  for (final p in people) {
    p.eat();
  }
}

void main() {
  final crew = [HumanWorker(), RobotWorker()];
  runShift(crew);                 // both work
  lunchBreak([HumanWorker()]);    // only feedable clients
}
```

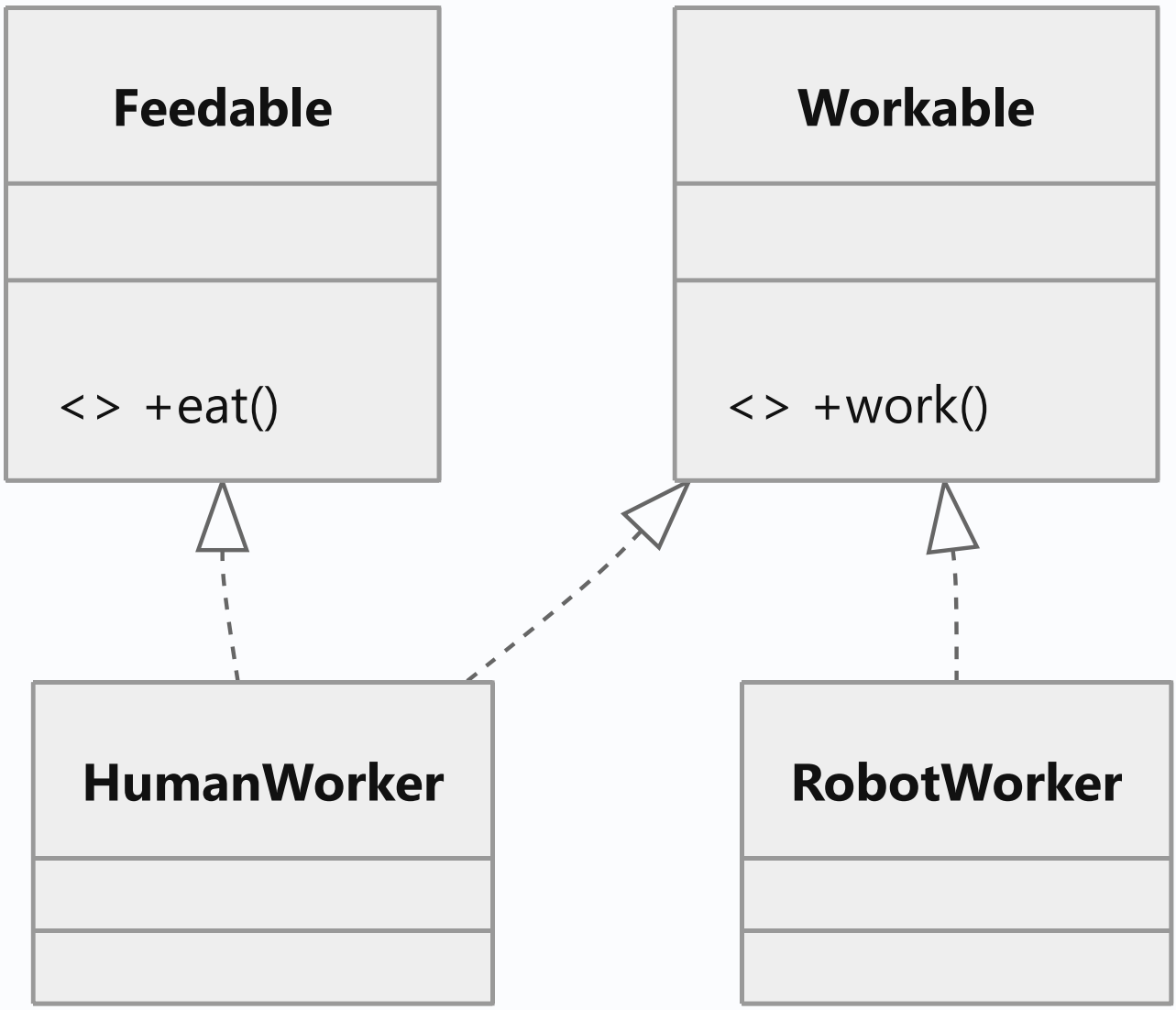
Flutter example

```
// ❌ A single `DataSource` interface with read+write+sync+cache methods forces a
//    read-only remote source to stub write/sync.
// ✅ Split into ReadableSource / WritableSource / SyncableSource; a read-only API
//    depends only on ReadableSource. Repositories compose the roles they need.
```

Enterprise example

An office-device driver: instead of one `MultiFunctionDevice` (print/scan/fax/staple), define `Printer`, `Scanner`, `FaxMachine`. A cheap printer implements `Printer` only; an all-in-one implements all three. Changing the fax spec never touches printer-only clients or code.

Diagrams



Common Mistakes

Mistake	Why	Fix
One fat interface for many clients	Forces useless stubs, wide coupling	Split into role interfaces
`throw UnsupportedError()` stubs	Signals a wrong/fat interface (also breaks LSP)	Segregate; implement only real roles
Interface mirroring an implementation's full API	Leaks capabilities clients don't need	Define interfaces from the *client's* needs
Over-segmenting into single-method interfaces everywhere	Ceremony without benefit	Group cohesive operations per role

Best Practices

- Define interfaces from the **consumer's** perspective (what that client needs), not the provider's full surface.
- Keep interfaces **cohesive and role-sized**; combine roles via multiple `implements` when needed.
- Treat `UnsupportedError` stubs as evidence of an ISP (and LSP) violation.
- Balance: cohesive roles, not a swarm of trivial one-method interfaces.

Performance

Neutral.

Advantages / Disadvantages

- + Lower coupling, no forced stubs, easier mocking (small fakes), safer changes, better LSP compliance.
- − More interfaces to manage; over-segmentation adds ceremony.

Interview Questions

1. 🟢 **State ISP.** — Clients shouldn't be forced to depend on interface members they don't use; prefer small role-specific interfaces.
2. 🟢 **What's the smell of an ISP violation?** — Implementations with empty or `throw UnsupportedError()` methods, and clients depending on capabilities they never call.
3. 🟡 **How do you apply ISP?** — Split fat interfaces into cohesive role interfaces; each class implements only the roles it supports; clients depend on the narrowest interface.
4. 🟡 **How does ISP relate to LSP?** — Fat interfaces push implementers to stub/throw, violating LSP; segregated interfaces are easy to honor fully, keeping substitution safe.
5. 🟡 **Where should interfaces be defined?** — At the consumer side, shaped by what the client needs — not as a mirror of a provider's entire API.
6. 🔴 **Can ISP be over-applied?** — Yes; a swarm of single-method interfaces adds indirection. Group cohesive operations into a role.
7. 🔴 **How does ISP improve testability?** — Small interfaces mean small fakes/mocks — you stub only the few methods a test needs.

Senior Engineer Tips

- Let **clients own the interfaces** they depend on (define `ReadableUserSource` where the reader lives), then implementations adapt to them — this also enables DIP (05_dip_dependency_inversion.md).

- Fewer methods per interface = smaller mocks = faster, clearer tests.
- If an implementer keeps throwing "not supported," your interface is doing too much.

Architect Perspective

ISP keeps module contracts lean and stable, minimizing the ripple of change across a system. Narrow, client-defined interfaces are the seams that make layers independently testable and swappable — the interface-level complement of SRP and the enabler of DIP and Clean Architecture boundaries (Module 40).

Summary

- Prefer many small, role-specific interfaces over one fat interface.
- Segregation removes forced stubs, lowers coupling, and supports LSP.
- Define interfaces from the client's needs; balance cohesion vs over-segmentation.

Revision Notes

- ISP: no client depends on unused members; split fat interfaces into roles.
- Smell: empty/ `UnsupportedError` stubs; clients coupled to unused methods.
- Define interfaces client-side; implement only real roles; combine via multiple `implements`.
- Supports LSP + testability (small mocks).

Practice Questions

1. Why does `RobotWorker` implementing `Worker` violate ISP?
2. How does ISP make mocking easier in tests?
3. When is splitting into more interfaces *not* worth it?

Coding Questions

1. Split a `MultiFunctionDevice` interface into `Printer` / `Scanner` / `Fax`; implement a printer-only and an all-in-one device.
2. Refactor a fat `Repository` into `ReadRepository` / `WriteRepository` and depend on the needed role per use case.
3. Define a client-side `ReadableSettings` interface a widget depends on, backed by a larger settings service.

Mini Project

Device driver roles (pure Dart): Model `Printer`, `Scanner`, `Fax` role interfaces; implement `BasicPrinter` (print only) and `AllInOne` (all three). Write client functions each depending on a single role, and tests using tiny fakes. Acceptance: no `UnsupportedError` stubs; clients depend only on needed roles; `dart analyze` clean.

D — Dependency Inversion Principle (DIP)

> High-level modules should not depend on low-level modules; both should depend on **abstractions**. And abstractions should not depend on details — details depend on abstractions.

Introduction

DIP inverts the "natural" dependency direction. Normally a high-level `OrderService` would directly `new` a low-level `MySqlDatabase`. DIP says: define an **abstraction** (`OrderRepository`) that the high-level module owns and depends on, and make the low-level detail (`MySqlOrderRepository`) implement it. The concrete dependency is then **injected**, not constructed inside.

Why this concept exists

Hard-wired dependencies (`final db = MySqlDatabase()`) make high-level policy code impossible to test in isolation and impossible to swap (change DB, mock in tests, support another provider). Inverting the dependency onto an abstraction decouples *what* the app does (policy) from *how* it's done (details), which is the core of testable, flexible architecture.

Real-world analogy

A **lamp** doesn't hard-wire itself to a specific power plant; it depends on the **socket standard** (abstraction). Any conforming power source plugs in. The lamp (high-level) and the power plant (low-level detail) both depend on the socket (abstraction) — not on each other.

Problem Statement

An `OrderService` directly instantiates a concrete `SmtpEmailSender` and `MySqlOrderRepository`, so you can't unit-test it without a real SMTP server and DB, and can't swap providers. You'll invert: depend on interfaces, inject implementations.

Internal Working

- The **high-level module owns the abstraction** (interface) it needs.

- **Low-level details implement** that abstraction.
- The concrete implementation is **injected** (constructor injection is the simplest form) — see Module 14 Dependency Injection.
- DIP is the mechanism that makes **OCP** and swap/mocking possible; it relies on polymorphism (03) and lean interfaces (ISP).

> **DIP vs DI:** DIP is the *principle* (depend on abstractions). **Dependency Injection** is a *technique* for supplying those dependencies. You can follow DIP with manual constructor injection; DI frameworks just automate wiring.

Memory Representation

Not applicable — a structural principle.

Compiler Behavior

- Depending on an interface type means the compiler accepts any implementation, enabling compile-time-safe substitution (real vs fake).

Runtime Behavior

- The injected implementation is used via dynamic dispatch; tests inject fakes, production injects real adapters — no code change in the high-level module.

Flutter Engine Behavior

Not applicable. (Flutter/Riverpod/get_it/Provider all exist to inject dependencies — DIP in practice. `InheritedWidget` provides dependencies down the tree.)

Dart VM Behavior

Not applicable beyond normal dispatch.

Examples

❌ Violation — high-level depends on concretes

```
class SmtpEmailSender {
  void send(String to, String body) => print('SMTP -> $to'); // low-level detail
}
class MySqlOrderRepository {
  void save(String order) => print('MySQL save $order'); // low-level detail
}

class OrderServiceBad {
  final _email = SmtpEmailSender();        // ❌ hard-wired
  final _repo = MySqlOrderRepository();    // ❌ hard-wired — untestable, unswappable
  void placeOrder(String order, String customer) {
    _repo.save(order);
    _email.send(customer, 'Order placed');
  }
}
```

✅ Refactor — depend on abstractions, inject details

```
// Abstractions OWNED by the high-level module:
abstract interface class OrderRepository {
  Future<void> save(String order);
}
abstract interface class EmailSender {
  Future<void> send(String to, String body);
}

class OrderService {
  final OrderRepository _repo;
  final EmailSender _email;
  OrderService(this._repo, this._email); // injected

  Future<void> placeOrder(String order, String customer) async {
    await _repo.save(order);
    await _email.send(customer, 'Order placed');
  }
}
```

```
// Low-level details implement the abstractions:
class MySqlOrderRepository implements OrderRepository {
  @override
  Future<void> save(String order) async => print('MySQL save $order');
}
class SmtpEmailSender implements EmailSender {
  @override
  Future<void> send(String to, String body) async => print('SMTP -> $to');
}

// Test doubles — no real DB/SMTP needed:
class FakeRepo implements OrderRepository {
  final saved = <String>[];
  @override
  Future<void> save(String order) async => saved.add(order);
}
class FakeEmail implements EmailSender {
  int count = 0;
  @override
  Future<void> send(String to, String body) async => count++;
}

Future<void> main() async {
  // production wiring:
  final prod = OrderService(MySqlOrderRepository(), SmtpEmailSender());
  await prod.placeOrder('o1', 'ada@x.com');

  // test wiring:
  final fakeRepo = FakeRepo();
  final fakeEmail = FakeEmail();
  final sut = OrderService(fakeRepo, fakeEmail);
  await sut.placeOrder('o2', 'test@x.com');
  print('${fakeRepo.saved} emails=${fakeEmail.count}'); // [o2] emails=1
}
```

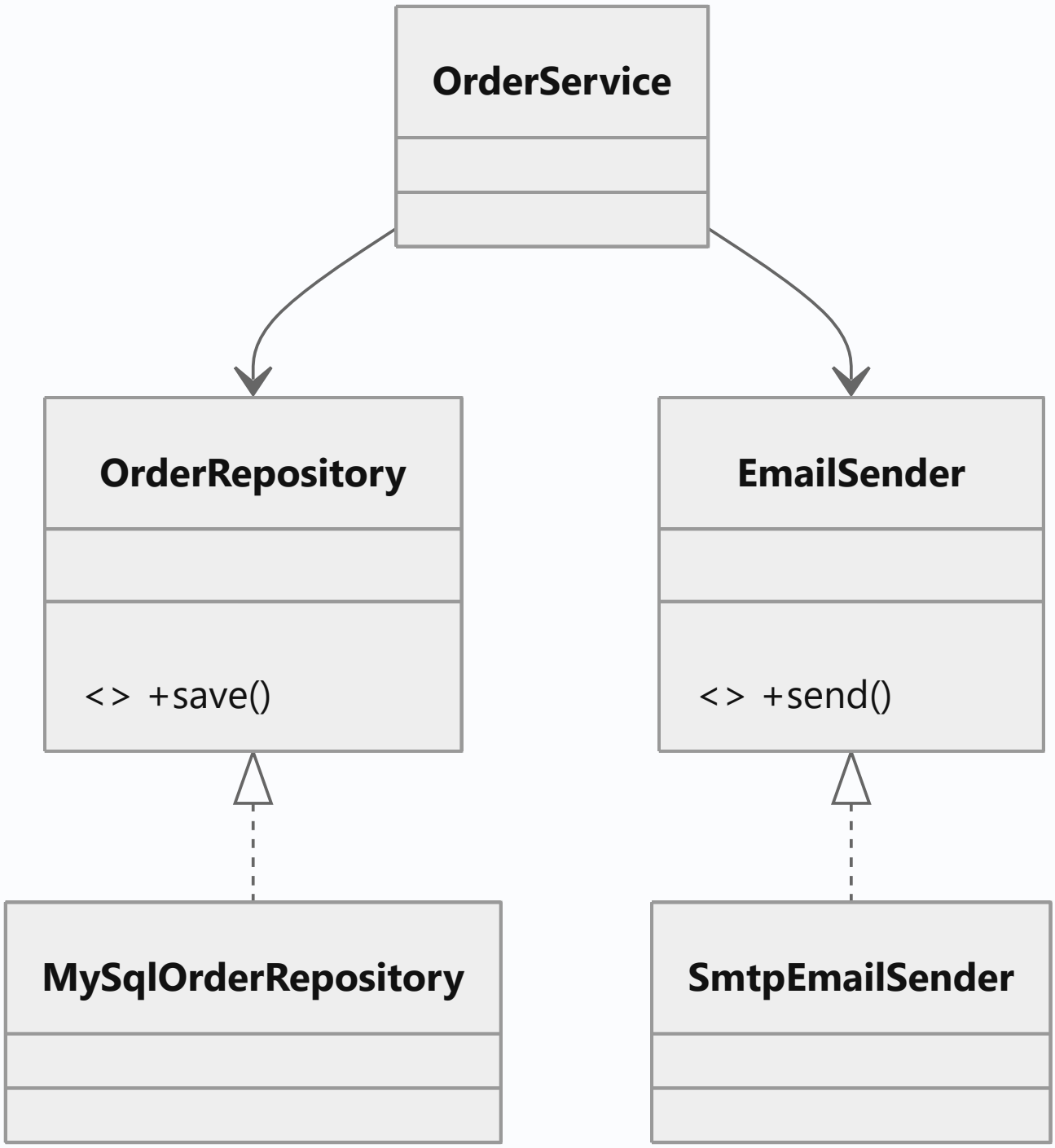
Flutter example

```
// ❌ Widget/Cubit that does `final repo = FirebaseUserRepo();` inside.
// ✅ Depend on an abstract UserRepository; inject FirebaseUserRepo in production and
//    FakeUserRepo in tests via Provider/Riverpod/get_it. UI/logic never names Firebase.
//    (See Modules 11, 14, 40.)
```

Enterprise example

Clean Architecture: the **domain** layer defines repository interfaces; the **data** layer implements them (REST/DB); the app **wires** them via DI at startup. The domain (policy) has zero imports of Flutter/HTTP/DB — it depends only on its own abstractions. This is DIP scaled to layers (Module 40).

Diagrams


OrderService
OrderRepository
<> +save()
EmailSender
<> +send()
MySqlOrderRepository
SmtpEmailSender

Common Mistakes

Mistake	Why	Fix
`new` /instantiating concretes inside high-level code	Untestable, unswappable	Depend on an interface, inject the impl
Interface owned by the low-level module	Dependency not truly inverted	Define the abstraction where it's *used*
Service locator overuse (hidden globals)	Opaque dependencies	Prefer constructor injection; use DI containers judiciously
Leaking implementation types through the interface	Re-couples to details	Keep interfaces detail-free (no `HttpResponse` in domain)
Injecting everything, even stable leaf values	Ceremony	Invert *volatile*/external dependencies, not trivial ones

Best Practices

- Depend on **abstractions**; inject concretes via constructors (simplest, most explicit DI).
- **The consumer owns the interface** (define `OrderRepository` in the domain, not the DB layer).
- Keep abstractions free of implementation detail types.
- Invert **volatile/external** dependencies (DB, network, clock, platform); don't over-inject stable, pure values.
- Wire the object graph at the **composition root** (app startup / DI container — Module 14).

Performance

Neutral; one indirection. The gain is testability and flexibility.

Advantages / Disadvantages

- **+** Testable (inject fakes), swappable (providers/DBs), decoupled layers, enables OCP + Clean Architecture.
- **−** More interfaces + wiring; over-injection adds ceremony; needs a composition root/DI strategy.

Interview Questions

1. 🟢 **State DIP.** — High- and low-level modules should both depend on abstractions, not on each other; details depend on abstractions, not vice versa.
2. 🟢 **DIP vs Dependency Injection?** — DIP is the principle (depend on abstractions); DI is the technique for supplying dependencies (e.g., constructor injection, DI containers).

3. 🟡 **Who should own the interface?** — The high-level consumer that uses it (define the repository interface in the domain), so the dependency truly points inward.
4. 🟡 **How does DIP enable testing?** — You inject fake implementations of the abstractions, testing high-level logic without real DBs/networks.
5. 🟡 **How do DIP and OCP relate?** — DIP's abstractions let you add/swap implementations (OCP) without editing the high-level module.
6. 🔴 **Constructor injection vs service locator?** — Constructor injection makes dependencies explicit and testable; service locators hide them (global lookup) and can obscure the graph — prefer constructor injection, use locators sparingly.
7. 🔴 **What's the composition root?** — The single place (app startup / DI setup) where concrete implementations are wired to abstractions; the rest of the app stays detail-agnostic.

Senior Engineer Tips

- Define interfaces by the **need**, not the provider's API (ties DIP to ISP): `Clock`, `UserRepository`, `PaymentGateway` — small and detail-free.
- Only invert *volatile* dependencies (things that change or reach outside the process). Inverting pure, stable helpers is noise.
- Keep the composition root thin and explicit; that's where the "dirty" concrete wiring is allowed to live.

Architect Perspective

DIP is the keystone of Clean/Hexagonal architecture: the domain defines ports (interfaces), adapters implement them, and DI wires them at the edge. This inversion is what keeps business rules independent of frameworks, databases, and UI — making the system testable, framework-agnostic, and adaptable to new providers or platforms (Modules 40, 45, 46).

Summary

- Depend on abstractions, not concretions; inject the details (constructor injection).
- The consumer owns the interface; wire concretes at the composition root.
- DIP enables OCP, testing-with-fakes, provider swaps, and Clean Architecture; invert volatile dependencies, not trivial ones.

Revision Notes

- DIP: both levels depend on abstractions; details depend on abstractions.
- DIP = principle; DI = technique (constructor injection preferred).
- Consumer owns the interface; keep it detail-free; wire at composition root.

- Invert volatile/external deps; enables OCP + testability + Clean Arch.

Practice Questions

1. Why is `OrderServiceBad` impossible to unit-test?
2. Why should the domain, not the data layer, define `OrderRepository`?
3. When is injecting a dependency *not* worth it?

Coding Questions

1. Invert a `WeatherWidget` that `new`s an `HttpWeatherApi` so it depends on a `WeatherRepository` and injects the impl.
2. Add a `Clock` abstraction so time-dependent logic is testable with a fake clock.
3. Wire a small object graph at a composition root (manual DI), then swap one impl for a fake.

Mini Project — SOLID capstone refactor

Order service refactor (pure Dart, the module capstone): You're given a deliberately bad `OrderEverything` class that: parses input, validates, computes totals with `if (type == ...)`, saves to a DB, sends email, formats a receipt, and logs — all with `new`ed dependencies. Refactor it to satisfy **all five** principles:

- **SRP:** split parsing/validation/pricing/persistence/notification/formatting/logging.
- **OCP:** discount/pricing as strategies added without editing the service.
- **LSP:** gateways/repositories honor contracts (return `Result`, no surprise throws).
- **ISP:** narrow role interfaces (`OrderRepository`, `EmailSender`, `Logger`), no fat interface.
- **DIP:** the `OrderService` depends only on abstractions; concretes injected at a composition root; fakes injected in tests.

Acceptance: `OrderService` has no `new`/type-switch; a full test suite runs with fakes (no real DB/SMTP); adding a new discount tier or notification channel edits no existing class; `dart analyze` clean.