05 · Design Patterns

Flutter Practice Notes

Contents

Dependency Injection Pattern

05 · Design Patterns

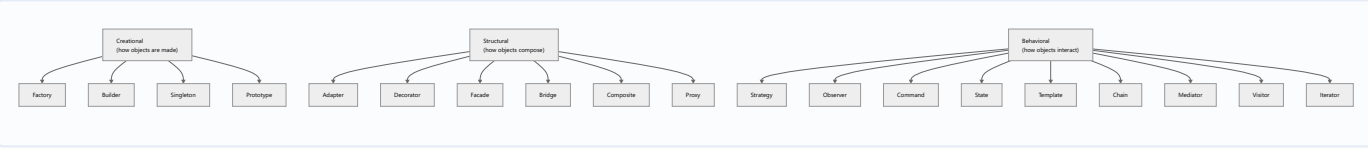
Introduction

Design patterns are **named, reusable solutions to recurring design problems**. They're a shared vocabulary ("let's use a Strategy here") and, more importantly, the *applied form* of SOLID. This module covers the classic Gang-of-Four (GoF) patterns across three families, plus two patterns essential to Flutter apps: **Repository** and **Dependency Injection**.

Why this module exists

Patterns turn SOLID principles (Module 04) into concrete structures. Knowing them lets you recognize a problem's shape and reach for a proven solution instead of reinventing (often badly). Interviewers use patterns to test design maturity; codebases use them to stay extensible.

The three families



Module map

Creational (object creation)

Pattern	Intent	File
Factory (Simple/Method/Abstract)	Create objects without naming concrete classes	01_factory.md
Builder	Construct complex objects step by step	02_builder.md
Singleton	One shared instance	03_singleton.md
Prototype	Clone existing objects	04_prototype.md

Structural (object composition)

Pattern	Intent	File
Adapter	Make incompatible interfaces work together	05_adapter.md
Decorator	Add behavior by wrapping	06_decorator.md
Facade	Simplify a complex subsystem	07_facade.md
Bridge	Decouple abstraction from implementation	08_bridge.md
Composite	Treat trees uniformly	09_composite.md
Proxy	Stand-in controlling access	10_proxy.md

Behavioral (object interaction)

Pattern	Intent	File
Strategy	Swappable algorithms	11_strategy.md
Observer	Notify dependents of change	12_observer.md
Command	Encapsulate a request as an object	13_command.md
State	Behavior varies with internal state	14_state.md
Template Method	Fixed skeleton, variable steps	15_template_method.md
Chain of Responsibility	Pass a request along handlers	16_chain_of_responsibility.md
Mediator	Centralize complex communication	17_mediator.md
Visitor	Add operations without editing types	18_visitor.md
Iterator	Traverse without exposing internals	19_iterator.md

Application patterns (Flutter-critical)

Pattern	Intent	File
Repository	Abstract data access behind a clean API	20_repository.md
Dependency Injection	Supply dependencies from outside	21_dependency_injection.md

Prerequisites

03 OOP and 04 SOLID. Patterns are OOP + SOLID applied.

How each file is structured

Beyond the standard template, each pattern file states: the **problem it solves**, a **UML diagram**, a **Dart implementation**, a **Flutter/real-world usage**, and **when NOT to use it** (patterns are tools, not goals).

What you'll be able to do after this module

- Recognize which pattern fits a problem — and when *no* pattern is the right answer.
- Implement each idiomatically in Dart (using records, sealed classes, closures where they simplify the classic form).
- Explain where Flutter/its ecosystem already uses these (e.g., Builder in widget builders, Observer in `ChangeNotifier`, Strategy in `ScrollPhysics`).

Summary

Patterns are the applied vocabulary of good design. Learn the intent and the smell each addresses; implement them in Dart; and resist over-engineering — reach for a pattern when the problem actually has its shape.

Factory Pattern (Simple Factory, Factory Method, Abstract Factory)

> A factory centralizes object creation so callers ask *what* they want, not *how* to build it — decoupling code from concrete classes.

Introduction

The "factory" family hides instantiation behind a method or class:

- **Simple Factory** — one method/class that returns the right concrete type.
- **Factory Method** — subclasses decide which concrete product to create.
- **Abstract Factory** — creates *families* of related products.

Why this concept exists

Direct `new ConcreteClass()` calls scatter concrete-type knowledge everywhere, violating OCP and DIP (Module 04). Factories centralize creation logic (selection, caching, validation) in one place, so adding a new product type or swapping implementations touches minimal code.

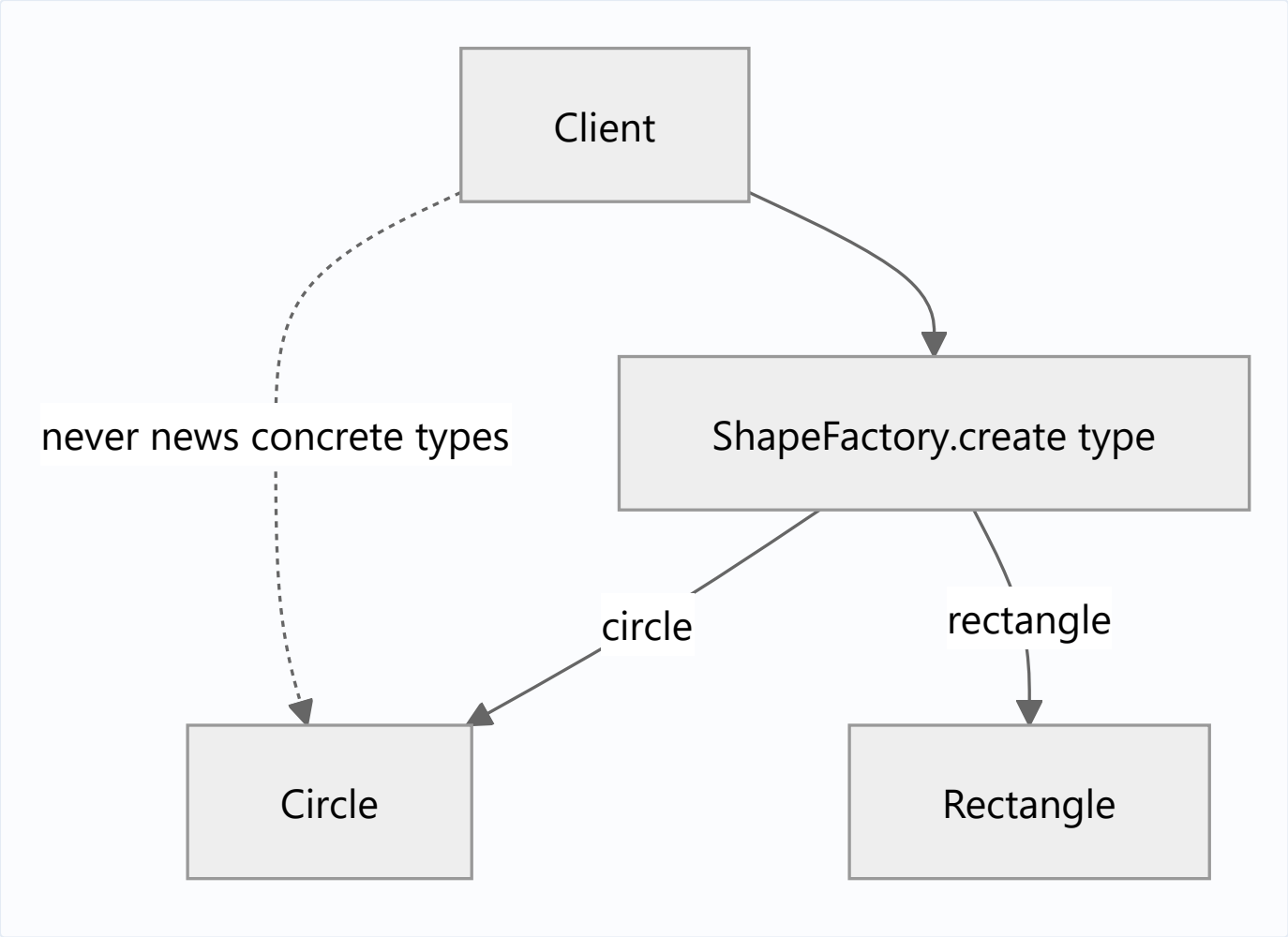
Real-world analogy

A **pizza shop**: you order "Margherita" (a name); the kitchen (factory) decides how to build it. You don't assemble dough and toppings yourself. An **abstract factory** is a shop that makes a whole *matching set* — a "Veg combo" (matching pizza + side + drink).

Problem Statement

You parse a `type` field from JSON (`'circle'` , `'rectangle'`) and must create the right `Shape` . Sprinkling `if (type == 'circle') Circle(...)` everywhere is fragile. You'll centralize it in a factory, then scale to families with an abstract factory (e.g., Material vs Cupertino widget sets).

Internal Working



- **Simple Factory:** a `static` method returning the base type based on input.
- **Factory Method:** an abstract `createProduct()` overridden per subclass (creation deferred to subtypes).
- **Abstract Factory:** an interface with multiple `createX()` methods producing a consistent family; concrete factories yield matching sets.
- Dart's `factory` constructor (02 · constructors) natively supports the Simple Factory idiom.

Memory Representation

Not applicable beyond normal allocation; factories may return cached instances (flyweight) instead of new ones.

Compiler Behavior

Not applicable — a design structure. Return type is the abstraction, so callers are statically decoupled from concretes.

Runtime Behavior

Factory logic runs at call time (parse `type`, choose subtype, optionally cache).

Flutter Engine Behavior

Not applicable. (Flutter uses factory-like selection everywhere: `Theme.of(context)` returns platform-appropriate values; adaptive widgets pick Material/Cupertino.)

Dart VM Behavior

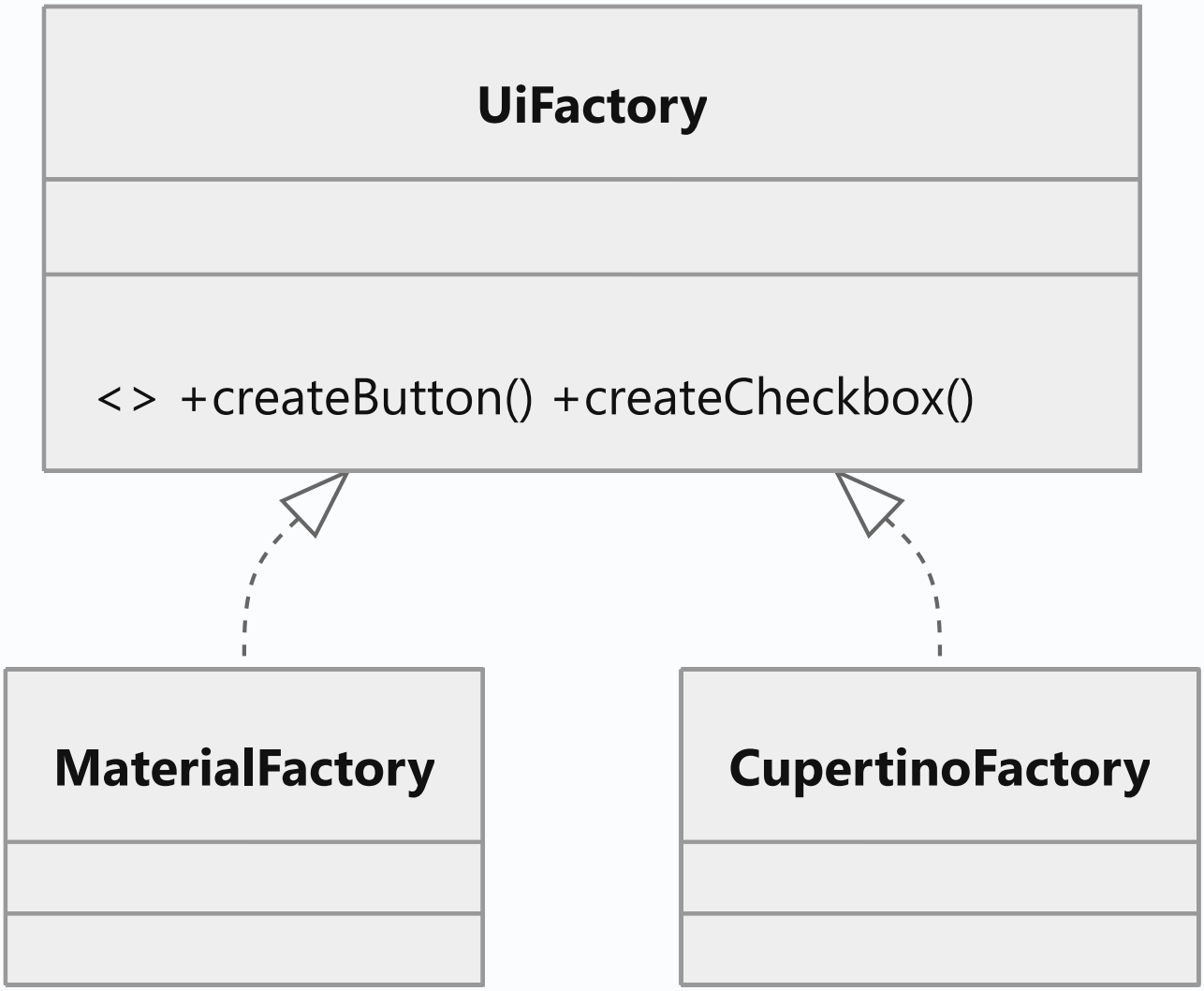
Not applicable.

Examples

```
// Simple Factory via Dart factory constructor
abstract class Shape {
  double area();
  factory Shape.fromType(String type, Map<String, double> p) => switch (type) {
        'circle' => Circle(p['r']!),
        'rectangle' => Rectangle(p['w']!, p['h']!),
        _ => throw ArgumentError('unknown shape $type'),
      };
}

class Circle implements Shape {
  final double r;
  Circle(this.r);
  @override
  double area() => 3.14159 * r * r;
}
class Rectangle implements Shape {
  final double w, h;
  Rectangle(this.w, this.h);
  @override
  double area() => w * h;
}

// Abstract Factory: families of related products (UI kits)
abstract interface class Button { String render(); }
abstract interface class Checkbox { String render(); }
```

```
abstract interface class UiFactory {
  Button createButton();
  Checkbox createCheckbox();
}

class MaterialFactory implements UiFactory {
  @override
  Button createButton() => _MButton();
  @override
  Checkbox createCheckbox() => _MCheckbox();
}
class CupertinoFactory implements UiFactory {
  @override
  Button createButton() => _CButton();
  @override
  Checkbox createCheckbox() => _CCheckbox();
}
class _MButton implements Button { @override String render() => '[Material Button]'; }
class _MCheckbox implements Checkbox { @override String render() => '[Material Checkbox]'; }
class _CButton implements Button { @override String render() => '(Cupertino Button)'; }
class _CCheckbox implements Checkbox { @override String render() => '(Cupertino Checkbox)'; }

void main() {
  final s = Shape.fromType('circle', {'r': 2});
  print(s.area()); // 12.56...

  final UiFactory ui = MaterialFactory(); // swap to CupertinoFactory for iOS
  print(ui.createButton().render());   // [Material Button]
  print(ui.createCheckbox().render()); // [Material Checkbox] — consistent family
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`new` concrete types scattered in callers	Coupling, OCP violation	Centralize in a factory
Factory returning concrete types	Recouples callers	Return the abstraction
Giant factory `switch` that also does business logic	SRP violation	Keep factories creation-only; consider a registry map
Abstract Factory when you only have one family	Over-engineering	Use a simple factory

Best Practices

- Return the **abstraction**, not concretes.

- Keep factories creation-focused (selection/validation/caching), not business logic.
- Use a `Map<Key, Builder>` registry for open extensibility (OCP) instead of a growing `switch`.
- Use Dart's `factory` constructor for the simple case.

Performance

Neutral; factories can *improve* it via caching/flyweight (return shared instances).

Advantages / Disadvantages

- **+** Decouples creation from use, centralizes selection, supports families, enables caching.
- **−** Extra indirection; Abstract Factory adds many types; overkill for trivial creation.

Interview Questions

1. 🟢 **What problem does a factory solve?** — It decouples clients from concrete classes by centralizing object creation behind a method/abstraction.
2. 🟢 **Factory Method vs Abstract Factory?** — Factory Method: subclasses decide which single product to create. Abstract Factory: creates *families* of related products consistently.
3. 🟡 **How does Dart support the factory pattern natively?** — `factory` constructors can return cached instances or subtypes without exposing `new`.
4. 🟡 **How do factories support OCP?** — New product types are added in the factory (ideally a registry), leaving callers unchanged.
5. 🟡 **When is Abstract Factory overkill?** — When there's only one product family or creation is trivial — use a simple factory.
6. 🔴 **How do you make a factory open for extension without editing it?** — Use a registry (`Map<Type, Builder>`) that new modules populate, instead of a hard-coded `switch`.
7. 🔴 **Where does Flutter use abstract-factory-like selection?** — Adaptive UI choosing Material vs Cupertino component families per platform.

Senior Engineer Tips

- Prefer a **registry map** over a `switch` for factories that grow; it turns OCP violations into registrations.
- Combine factory + DI: register factories in the container so creation and wiring stay centralized (21_dependency_injection.md).
- Keep the "dirty" concrete knowledge in the factory (composition root), not spread across the app.

Architect Perspective

Factories are a creation seam that keeps the bulk of the codebase depending on abstractions (DIP). Combined with DI, they localize concrete-type decisions to composition roots and platform adapters, enabling multi-platform (Material/Cupertino), multi-provider, and testable designs.

Summary

- Factories centralize object creation behind abstractions; variants: Simple, Factory Method, Abstract Factory.
- Return abstractions; prefer registries for extensibility; use Dart `factory` constructors.
- Great with DI; don't over-apply for trivial creation.

Revision Notes

- Simple factory = method returns base type; Factory Method = subclass chooses product; Abstract Factory = family of products.
- Dart `factory` ctor supports it natively.
- Return abstraction; registry map > switch (OCP).
- Flutter: adaptive Material/Cupertino families.

Practice Questions

1. When would you choose Abstract Factory over a simple factory?
2. How does a registry map make a factory OCP-compliant?
3. Why should a factory return the base type, not the concrete?

Coding Questions

1. Build a `NotificationFactory.create(channel)` returning `EmailNotifier` / `SmsNotifier` via a registry.
2. Implement an abstract `ThemeFactory` producing matching `Button` + `Card` for Light/Dark.
3. Convert a `switch`-based shape creator into a registry-backed factory.

Mini Project

UI-kit abstract factory (pure Dart): Implement `UiFactory` with `Material` / `Cupertino` families (`Button`, `Checkbox`, `Switch`), select by a `Platform` flag, and render a form using only the abstract types. Acceptance: callers never name concrete widgets; adding a third family (e.g., `Fluent`) edits no client code; `dart analyze` clean.

Builder Pattern

> Builder constructs a complex object step by step, separating *how* it's assembled from *what* it represents — ideal when an object has many optional parts or configuration.

Introduction

The Builder pattern assembles a complex object through a series of steps (often chained), then produces the finished product with a `build()`. It shines when a constructor would otherwise take a dozen optional parameters or require a valid multi-step assembly.

Why this concept exists

Telescoping constructors (`Pizza(size, cheese, [pepperoni], [mushroom], ...)`) and giant named-parameter lists become unreadable and error-prone. Builder gives a fluent, validated, incremental construction path and can enforce required-before-optional ordering.

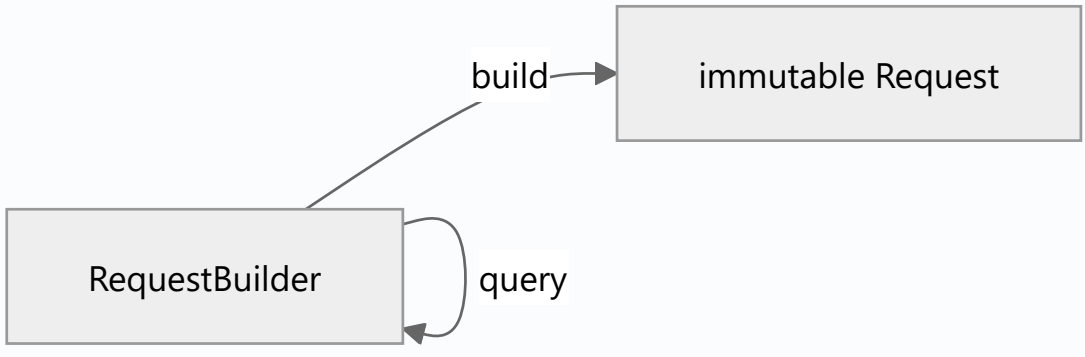
Real-world analogy

Ordering a **custom burger**: you add patty, then cheese, then sauce, then "done." The counter (builder) accumulates your choices and hands you the finished burger. You don't hand the kitchen one giant order string.

Problem Statement

Build an HTTP request with optional headers, query params, body, and timeout — without a 10-argument constructor. You'll use a fluent builder producing an immutable `Request`.

Internal Working



- A mutable **builder** accumulates state via chained methods returning `this` (or via cascades `..`).
- `build()` validates and returns an **immutable** product.
- Dart alternatives: **named parameters + `copyWith`** often replace Builder for simple cases; cascades (`..`) give fluent config natively.

Memory Representation

The builder is a short-lived mutable object; the product is typically immutable. Discard the builder after `build()`.

Compiler Behavior

Not applicable. (Fluent chaining is ordinary method calls returning the builder.)

Runtime Behavior

Each step mutates builder state; `build()` snapshots it into the product (defensive copies of collections).

Flutter Engine Behavior

Not applicable. (Flutter's `WidgetBuilder` / `builder:` callbacks are a *builder-callback* idiom; `ThemeData` and `TextStyle.copyWith` cover many builder use cases.)

Dart VM Behavior

Not applicable.

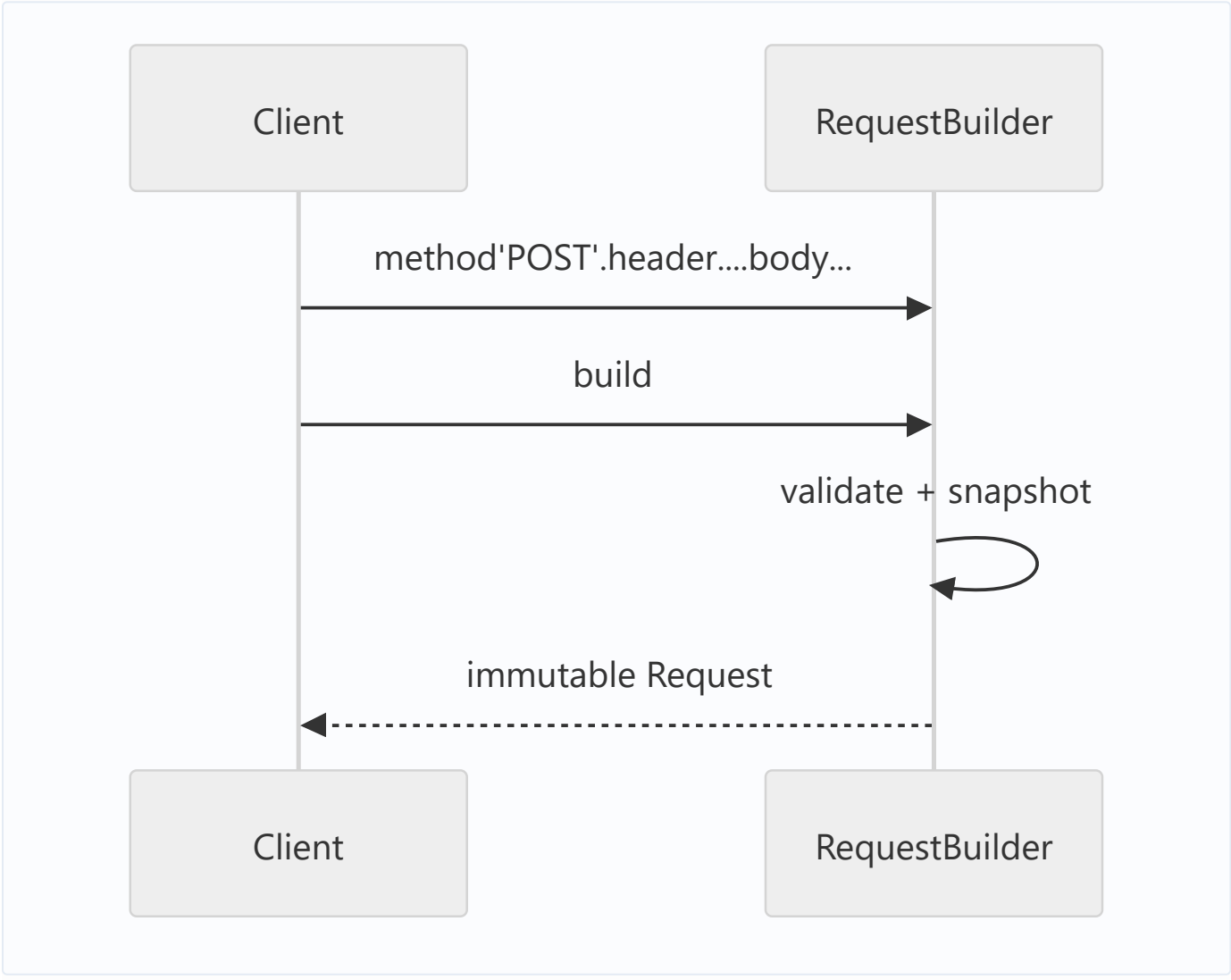
Examples

```
class Request {
  final String method, url;
  final Map<String, String> headers;
  final Map<String, String> query;
  final String? body;
  final Duration timeout;

  Request._({
    required this.method,
    required this.url,
```

```
    required this.headers,
    required this.query,
    required this.body,
    required this.timeout,
  });

  @override
  String toString() =>
      '$method $url q=$query h=$headers body=$body t=${timeout.inSeconds}s';
}

class RequestBuilder {
  final String url;
  String _method = 'GET';
  final _headers = <String, String>{};
  final _query = <String, String>{};
  String? _body;
  Duration _timeout = const Duration(seconds: 30);

  RequestBuilder(this.url);

  RequestBuilder method(String m) { _method = m; return this; }
  RequestBuilder header(String k, String v) { _headers[k] = v; return this; }
  RequestBuilder query(String k, String v) { _query[k] = v; return this; }
  RequestBuilder body(String b) { _body = b; return this; }
  RequestBuilder timeout(Duration d) { _timeout = d; return this; }

  Request build() {
    if (_method == 'GET' && _body != null) {
      throw StateError('GET cannot have a body'); // validation at build
    }
    return Request._(
      method: _method,
      url: url,
      headers: Map.unmodifiable(_headers), // defensive copy
      query: Map.unmodifiable(_query),
      body: _body,
      timeout: _timeout,
    );
  }
```

```
}


void main() {
  final req = RequestBuilder('https://api.example.com/users')
      .method('POST')
      .header('Authorization', 'Bearer x')
      .query('page', '1')
      .body('{"name":"Ada"}')
      .timeout(const Duration(seconds: 10))
      .build();
  print(req);
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Builder producing a mutable product	Loses immutability benefits	`build()` returns an immutable object
Not defensively copying collections	Post-build mutation leaks in	`Map.unmodifiable` / `List.of`
Using Builder where named params + copyWith suffice	Over-engineering	Prefer Dart idioms for simple cases
Forgetting validation in `build()`	Invalid objects created	Validate before returning

Best Practices

- Return an **immutable** product; defensively copy collections.
- Validate invariants in `build()`.
- Prefer Dart **named parameters + `copyWith`** or **cascades** for simple configuration; reserve full Builder for genuinely complex/multi-step assembly.
- Consider a `sealed`/staged builder if ordering must be enforced at compile time.

Performance

Neutral; one throwaway builder allocation per product.

Advantages / Disadvantages

- **+** Readable step-by-step construction, validation point, immutable result, avoids telescoping constructors.
- **−** Extra class/boilerplate; often unnecessary in Dart thanks to named params + `copyWith`.

Interview Questions

1. 🟢 **What problem does Builder solve?** — Constructing complex objects with many optional/variant parts without unwieldy constructors, with a clear validation point.
2. 🟢 **What does `build()` typically return?** — A finished, usually immutable product (with defensive copies of internal collections).
3. 🟡 **When is Builder unnecessary in Dart?** — When named parameters + defaults + `copyWith` (or cascades) already give readable, validated construction.
4. 🟡 **How do you enforce validation with Builder?** — Check invariants inside `build()` before returning the product.

5. 🟡 **Builder vs Factory?** — Factory decides *which* object to create; Builder decides *how to assemble* one complex object step by step.
6. 🔴 **How can you enforce required steps at compile time?** — A staged/typed builder where each step returns a different type exposing only the next valid methods.
7. 🔴 **Where does Flutter use builder idioms?** — `builder:` / `itemBuilder` callbacks (lazy widget construction) and `copyWith` on `ThemeData` / `TextStyle`.

Senior Engineer Tips

- In Dart, reach for `copyWith` /named params first; introduce a Builder only when assembly is truly multi-step or conditional.
- Use cascades (`..`) for fluent config of an existing mutable object without a dedicated builder.
- Make the product immutable so it can be shared/compared safely (02 · immutability).

Architect Perspective

Builder centralizes complex, validated construction — useful for query builders, request builders, and configuration DSLs. In Dart apps it's often subsumed by named params + `copyWith`, so the architectural value is mostly in DSL-like or staged-construction scenarios (e.g., building queries for Module 20 Database).

Summary

- Builder assembles complex objects step by step, validating and producing an immutable product.
- Prefer Dart named params + `copyWith` /cascades for simple cases; use Builder for genuinely complex assembly.
- Defensively copy collections; validate in `build()`.

Revision Notes

- Builder = step-by-step construction → `build()` returns immutable product.
- Chain methods return `this`; defensive-copy collections; validate in `build()`.
- Dart alt: named params + `copyWith` + cascades.
- Builder=how to assemble; Factory=which to create.

Practice Questions

1. Why should `build()` return an immutable object?
2. When do Dart named parameters make Builder redundant?

3. How would you enforce that `method` is set before `body`?

Coding Questions

1. Build a `SqlQueryBuilder` (`select` / `from` / `where` / `orderBy` / `build`) producing an immutable query string.
2. Implement a staged builder that only allows `.body()` after `.method('POST')` (type-enforced).
3. Rewrite a 9-parameter constructor as named params + `copyWith` and argue whether Builder is needed.

Mini Project

HTTP request builder (pure Dart): Implement a fluent `RequestBuilder` producing an immutable `Request`, with validation (no body on GET), defensive copies, and support for chaining. Add tests for valid/invalid builds. Acceptance: product immutable; validation enforced in `build()`; a note comparing to the named-params+ `copyWith` alternative; `dart analyze` clean.

Singleton Pattern

> Singleton guarantees a class has exactly one instance with a global access point — powerful, frequently overused, and best replaced by dependency injection in most apps.

Introduction

Singleton ensures one shared instance (a logger, a config, a cache). Dart makes it trivial with a private constructor + static instance or a `factory` constructor. This file covers the idioms, lazy vs eager init, thread/isolate considerations, and — critically — **why to prefer DI**.

Why this concept exists

Some resources genuinely should be single (a connection pool, an app config). A singleton provides controlled single-instance access. But it's also the most abused pattern: it's global mutable state in disguise, which hurts testability and hides dependencies — so this file teaches it *and its safer alternative*.

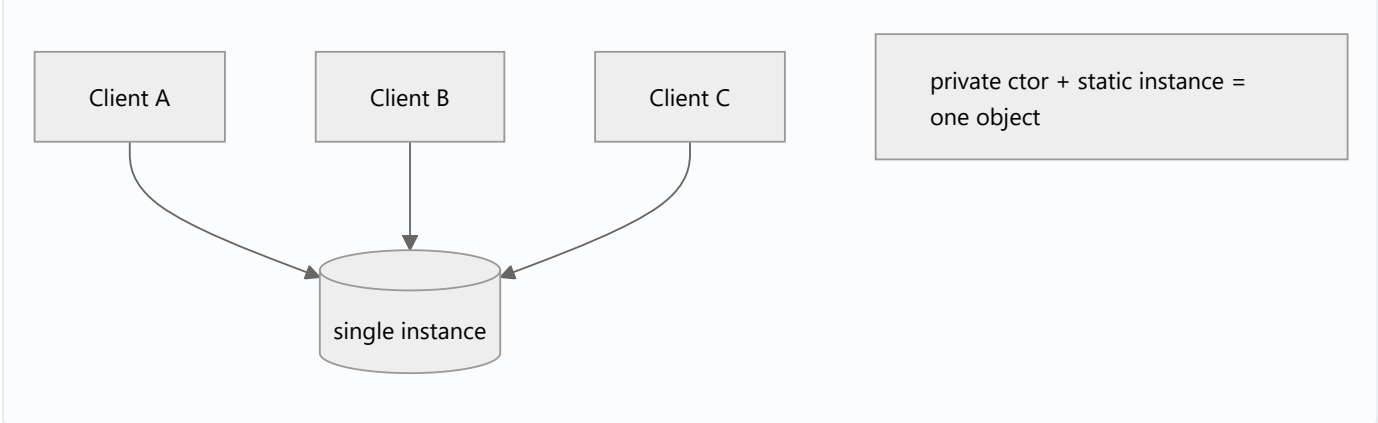
Real-world analogy

A country's **central bank**: there's exactly one, and everyone references the same one. Useful when singularity is real — but if *everything* routed through one office, you'd get bottlenecks and untraceable dependencies (the singleton overuse problem).

Problem Statement

You need one shared `AppConfig` and one `Logger`. You'll implement them as singletons — then see how a DI-managed single instance gives the same singularity *with* testability.

Internal Working



- **Idiom 1 (factory):** `factory Logger() => _instance;` with a private `Logger._()` and `static final _instance`.
- **Idiom 2 (static field):** `static final Logger instance = Logger._();` (eager, lazy-loaded on first access due to `final` top-level/static lazy init).
- **Lazy:** `static Logger? _i; static Logger get instance => _i ??= Logger._();`
- **Isolate caveat:** each isolate has its own memory → a "singleton" is single **per isolate**, not globally (02 · isolates).

Memory Representation

One heap instance referenced by the static field, alive for the isolate's lifetime (effectively a GC root — never collected while the class is loaded).

Compiler Behavior

`static final` fields are **lazily initialized** on first access in Dart — so even the "eager" idiom is effectively lazy.

Runtime Behavior

First access constructs the instance; subsequent accesses return the same object. No locking needed *within* an isolate (single-threaded event loop).

Flutter Engine Behavior

Not applicable. (Flutter apps often use singletons via DI containers like `get_it`; `WidgetsBinding.instance` is a framework singleton.)

Dart VM Behavior

Per-isolate static state; the instance lives in the isolate's heap. Background isolates get their *own* copy.

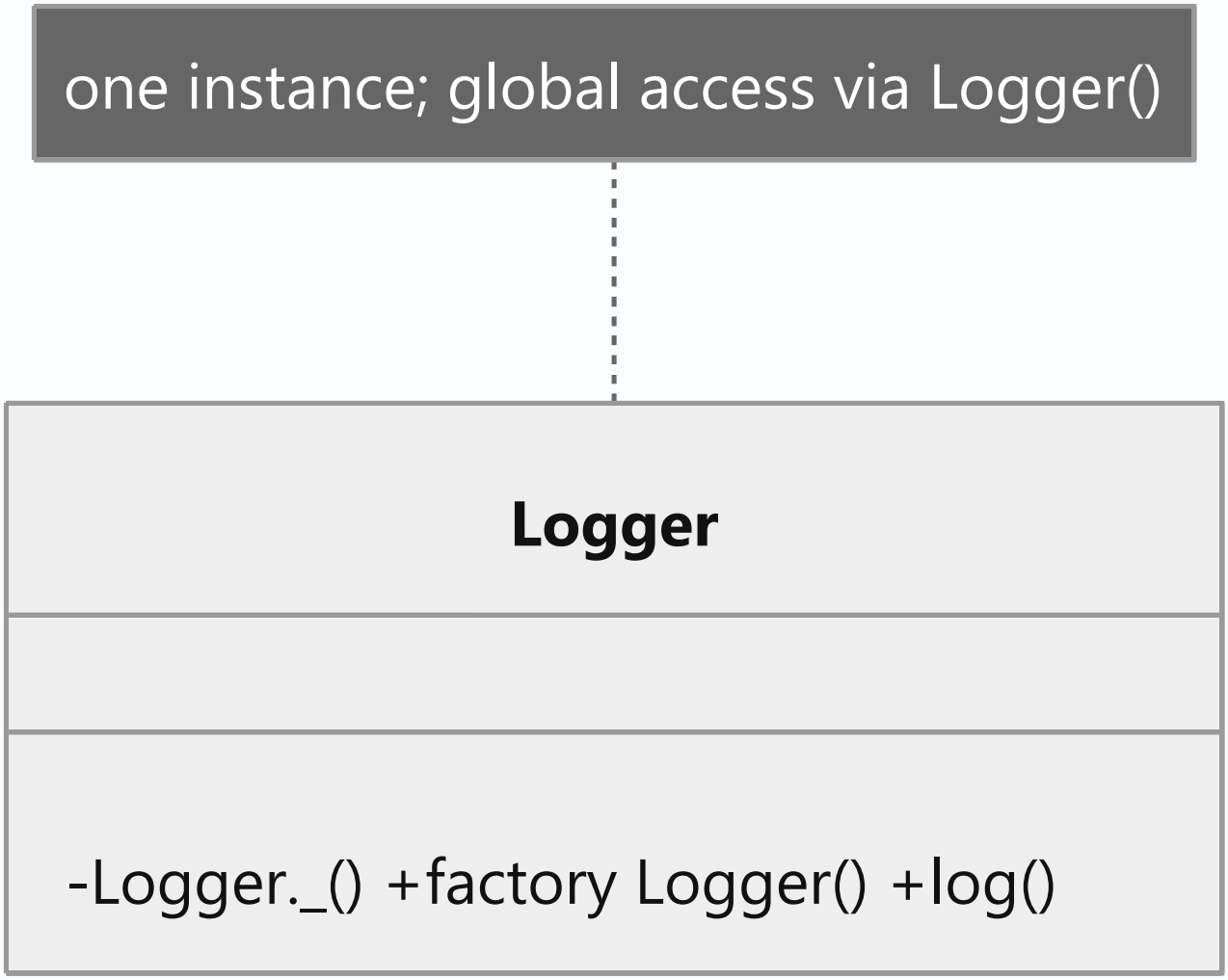
Examples

```
// Idiom 1: factory constructor returns the single instance
class Logger {
  Logger._(); // private
  static final Logger _instance = Logger._();
  factory Logger() => _instance; // Logger() always yields the same object
  void log(String m) => print('[LOG] $m');
}

// Idiom 2: static instance (lazy via `static final`)
class AppConfig {
  AppConfig._();
  static final AppConfig instance = AppConfig._();
  String env = 'prod';
}

// Idiom 3: explicit lazy
class Cache {
  Cache._();
  static Cache? _i;
  static Cache get instance => _i ??= Cache._();
  final _store = <String, Object>{};
}

void main() {
  print(identical(Logger(), Logger()));                // true
  print(identical(AppConfig.instance, AppConfig.instance)); // true
  Logger().log('hello');
  AppConfig.instance.env = 'staging';
  print(AppConfig.instance.env); // staging — shared state
}
```

The DI alternative (preferred)

```
// Instead of a hard singleton, register ONE instance in a DI container:
//   getIt.registerSingleton<Logger>(ConsoleLogger());
// Consumers receive it via constructor injection -> single instance AND testable
// (swap a FakeLogger in tests). See dependency_injection.md.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Using singletons as a default	Global mutable state; hidden deps; hard tests	Prefer DI-managed single instances
Mutable global state in a singleton	Spooky action at a distance	Keep it stateless or inject state
Assuming global uniqueness across isolates	Singletons are per-isolate	Don't rely on cross-isolate singleton state
Singleton holding `BuildContext`/large graphs	Memory leaks (never collected)	Don't store transient/UI objects in singletons

Best Practices

- Prefer **DI** (register one instance in the container) over hard singletons — same singularity, better testability.
- If you must use a raw singleton, keep it **stateless** or with carefully controlled state.
- Remember singletons are **per-isolate**; design background work accordingly.
- Never store `BuildContext` /short-lived objects in a singleton.

Performance

One allocation; lazy init defers cost to first use. Negligible overhead; the risks are architectural, not performance.

Advantages / Disadvantages

- **+** Guaranteed single instance, easy global access, lazy init.
- **−** Global mutable state, hidden dependencies, hard to mock/test, per-isolate (not truly global), lifecycle you don't control.

Interview Questions

1. 🟢 **What is a Singleton?** — A class with exactly one instance and a global access point.
2. 🟢 **How do you implement one in Dart?** — Private constructor + a static instance, optionally exposed via a `factory` constructor; `static final` gives lazy init.
3. 🟡 **Why is Singleton often considered an anti-pattern?** — It's global mutable state: it hides dependencies, couples code, and makes testing/mocking hard.
4. 🟡 **What's the DI alternative and why is it better?** — Register one instance in a DI container and inject it; you get singularity *and* the ability to swap fakes in tests.
5. 🟡 **Are Dart singletons globally unique?** — No — per isolate. Background isolates get their own copy.
6. 🔴 **Is locking needed for a Dart singleton?** — Not within an isolate (single-threaded event loop); cross-isolate sharing isn't possible via memory anyway.
7. 🔴 **What lifecycle risk do singletons carry?** — They live for the isolate's lifetime (GC root); storing large/transient objects causes leaks.

Senior Engineer Tips

- Treat "I need a singleton" as "I need one instance" — satisfy it via DI registration, not a hard-coded static, so tests can substitute it.

- Keep any legitimate singleton **immutable/stateless**; put mutable state behind injected, testable services.
- Audit singletons for retained references — they never get collected.

Architect Perspective

Singletons centralize but also globalize. In scalable architectures, single-instance services are managed by a **DI container at the composition root**, preserving singularity while keeping the dependency graph explicit and testable. Raw ad-hoc singletons are a common source of coupling and untestable code at scale (Module 14).

Summary

- Singleton = one instance + global access; trivial in Dart via private ctor + static/factory.
- It's global state — prefer DI-managed single instances for testability.
- Per-isolate, lazily initialized; keep stateless; avoid storing transient objects.

Revision Notes

- Dart singleton: `Foo._()` + `static final instance` (lazy) or `factory Foo() => _i`.
- Per-isolate (not global); no locking needed within an isolate.
- Anti-pattern risk: global mutable state, hidden deps, hard tests.
- Prefer DI single instance; keep stateless; watch retained refs (leaks).

Practice Questions

1. Why does `identical(Logger(), Logger())` return true?
2. Give two reasons to prefer DI over a hard singleton.
3. Why isn't a Dart singleton shared across isolates?

Coding Questions

1. Implement `Logger` three ways (factory, static-final, explicit-lazy); prove single-instance.
2. Convert a hard `Analytics` singleton into a DI-registered instance and inject it into a service.
3. Show a leak: a singleton holding a growing list; then bound/clear it.

Mini Project

Config & logging via DI-singleton (pure Dart): Implement `AppConfig` and `Logger`, first as raw singletons, then refactor to register single instances in a tiny DI container and inject them. Write a test that swaps a `FakeLogger`. Acceptance: single-instance guaranteed; production uses real, tests use fake; no `BuildContext`/transient stored; `dart analyze` clean.

Prototype Pattern

> Prototype creates new objects by **cloning** an existing instance rather than constructing from scratch — useful when construction is expensive or the concrete type isn't known statically.

Introduction

Prototype produces new objects by copying a prototypical instance (`clone()`), optionally tweaking the copy. In Dart this most often appears as `copyWith` on immutable classes and as registries of pre-configured prototypes.

Why this concept exists

Sometimes building an object is costly (heavy computation, config loading) or you only have an *instance* (not its class) to base a new one on. Cloning sidesteps re-running expensive construction and lets you vary configuration from a known-good baseline.

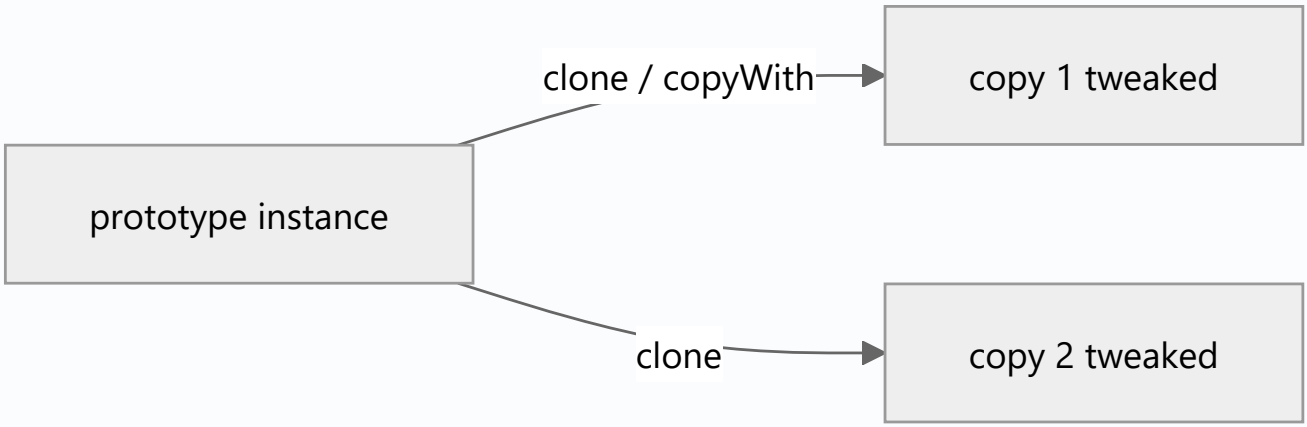
Real-world analogy

A **cookie cutter with pre-set dough**: instead of mixing new dough each time, you stamp copies of a prepared master and decorate each differently. Or "Save As" on a document — start from an existing one and modify.

Problem Statement

You have a fully-configured default `Document` template (margins, fonts, header). Creating variants by re-specifying everything is wasteful and error-prone. You'll clone the prototype and adjust only what differs.

Internal Working



- Define a `clone()` (or `copyWith`) that returns a copy.
- **Shallow vs deep** clone matters when the object has mutable nested state (03 · equality_and_copying).
- A **prototype registry** maps keys → pre-built prototypes to clone on demand.
- Dart idiom: immutable class + `copyWith` *is* Prototype for most cases.

Memory Representation

Each clone is a new heap object; deep clones duplicate nested mutable objects, shallow clones share them.

Compiler Behavior

Not applicable.

Runtime Behavior

Cloning copies fields at runtime; deep clone recurses. Immutable clones can share nested immutables safely (structural sharing).

Flutter Engine Behavior

Not applicable. (`ThemeData.copyWith`, `TextStyle.copyWith`, and widget config cloning are Prototype-flavored.)

Dart VM Behavior

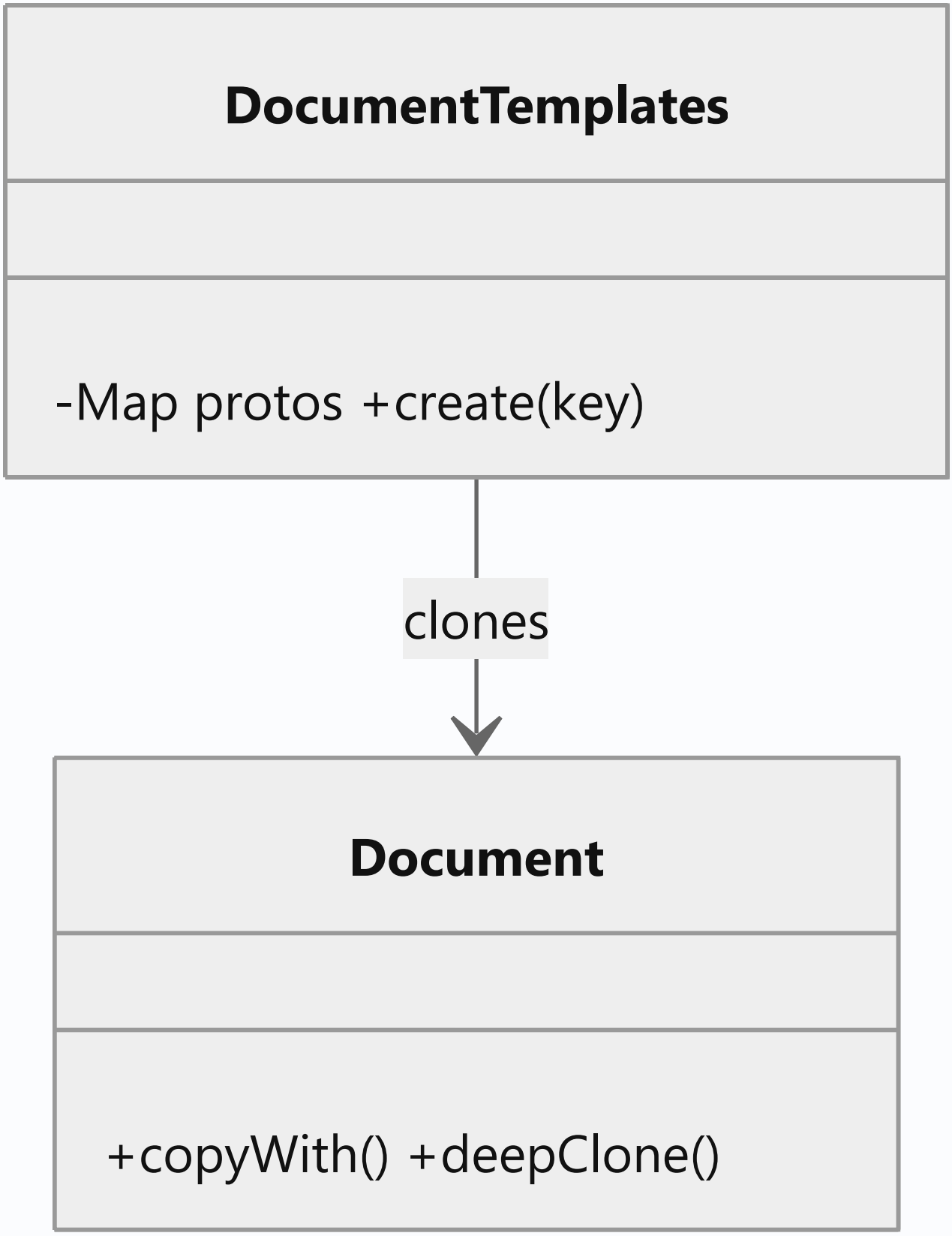
Not applicable.

Examples

```
class Document {
  final String title;
  final int marginPt;
  final List<String> sections; // mutable nesting

  const Document({
    required this.title,
    this.marginPt = 20,
    this.sections = const [],
  });

  // copyWith == Prototype clone-with-tweaks (shallow: shares/replaces list)
  Document copyWith({String? title, int? marginPt, List<String>? sections}) =>
      Document(
        title: title ?? this.title,
        marginPt: marginPt ?? this.marginPt,
        sections: sections ?? this.sections,
      );

  // explicit DEEP clone (new sections list)
  Document deepClone() =>
      Document(title: title, marginPt: marginPt, sections: List.of(sections));

  @override
  String toString() => '$title (m=$marginPt, $sections)';
}

// Prototype registry
class DocumentTemplates {
  final _protos = <String, Document>{
    'letter': const Document(title: 'Letter', marginPt: 25, sections: ['Header', 'Body']),
    'memo': const Document(title: 'Memo', marginPt: 15, sections: ['To', 'Message']),
  };
  Document create(String key) => _protos[key]!.copyWith(); // clone from prototype
}

void main() {
  const master = Document(title: 'Master', marginPt: 20, sections: ['Intro']);
```

```
  final variant = master.copyWith(title: 'Variant', marginPt: 30);
  print(master);  // Master (m=20, [Intro])
  print(variant); // Variant (m=30, [Intro])

  final tpl = DocumentTemplates();
  print(tpl.create('memo')); // Memo (m=15, [To, Message])
}
```

Diagrams


DocumentTemplates
-Map protos +create(key)
clones
Document
+copyWith() +deepClone()

Common Mistakes

Mistake	Why	Fix
Shallow clone when deep is needed	Copies share mutable nested state	Deep-clone nested mutables (or use immutables)
Reinventing `copyWith` manually everywhere	Boilerplate/bugs	Use `freezed` / `copyWith` codegen
Prototype for cheap objects	Unnecessary indirection	Just construct them
Mutating a shared prototype in a registry	Corrupts all future clones	Keep prototypes immutable

Best Practices

- Prefer **immutable classes + `copyWith`** — that's idiomatic Prototype in Dart.
- Keep registry prototypes **immutable** so clones start from a stable baseline.
- Choose deep vs shallow clone deliberately based on nested mutability.
- Use codegen (`freezed`) to avoid hand-written clone boilerplate.

Performance

Cloning avoids expensive re-construction; deep clones cost O(size). Immutability + structural sharing keeps it cheap.

Advantages / Disadvantages

- **+** Avoids costly construction, varies from a known-good baseline, decouples from concrete type.
- **−** Deep-clone complexity; unnecessary for cheap/simple objects; manual clones are error-prone.

Interview Questions

1. 🟢 **What is the Prototype pattern?** — Creating new objects by cloning an existing instance rather than constructing from scratch.
2. 🟢 **How does Dart express Prototype idiomatically?** — Immutable classes with `copyWith` (clone-with-tweaks).
3. 🟡 **Shallow vs deep clone — why does it matter?** — Shallow clones share nested mutable objects (edits leak); deep clones duplicate them for independence.
4. 🟡 **When is Prototype worth it over `new`?** — When construction is expensive, or you must copy from an instance whose concrete type you don't know.

5. 🟡 **What's a prototype registry?** — A map of pre-configured prototypes cloned on demand by key.
6. 🔴 **Why keep registry prototypes immutable?** — A mutated shared prototype corrupts every subsequent clone.
7. 🔴 **Where does Flutter use Prototype-like cloning?** — `copyWith` on `ThemeData` / `TextStyle` /config objects.

Senior Engineer Tips

- In Dart, you rarely write a classic `clone()` — `copyWith` (hand-written or `freezed`) covers it and preserves immutability.
- Reach for a prototype registry when you have many pre-baked configurations (themes, templates, default entities).
- Be explicit about clone depth in tests to prevent shared-mutation surprises.

Architect Perspective

Prototype (via `copyWith`) is the everyday mechanism for evolving immutable state — the backbone of reducers, `copyWith`-based state updates, and template systems. Deep-clone decisions intersect with immutability strategy (02 · immutability); standardize on immutable models so cloning is safe and cheap.

Summary

- Prototype clones an existing instance instead of constructing anew.
- In Dart it's `copyWith` on immutable classes; use a registry for pre-baked prototypes.
- Mind shallow vs deep clone; keep prototypes immutable; codegen the boilerplate.

Revision Notes

- Prototype = clone existing object (`clone()` / `copyWith`).
- Dart idiom: immutable + `copyWith` ; registry of prototypes.
- Shallow shares nested mutables; deep duplicates them.
- Keep prototypes immutable; Flutter: `ThemeData/TextStyle.copyWith` .

Practice Questions

1. Why is `copyWith` considered a Prototype implementation?
2. When would a shallow clone cause a bug?
3. When is Prototype unnecessary?

Coding Questions

1. Add `copyWith` + `deepClone` to a mutable `Board` (grid) and show the difference.
2. Build a `ThemeTemplates` registry cloning base themes with tweaks.
3. Demonstrate a shared-mutation bug from shallow cloning, then fix it.

Mini Project

Document template engine (pure Dart): Implement an immutable `Document` with `copyWith`, a `DocumentTemplates` prototype registry, and both shallow/deep clone paths. Write tests proving clones are independent where required. Acceptance: prototypes immutable; documented shallow vs deep behavior; `dart analyze` clean.

Adapter Pattern

> An adapter wraps an incompatible interface so it looks like the interface your code expects — the "plug converter" between two systems that can't talk directly.

Introduction

Adapter (a.k.a. Wrapper) translates one interface into another. You have a class your code needs to use, but its API doesn't match what your code expects; the adapter sits between them, forwarding calls and converting shapes.

Why this concept exists

You can't always change third-party or legacy code, yet your app depends on a clean, consistent interface (often for DIP — Module 04). Adapter lets incompatible components collaborate without modifying either — essential when integrating SDKs, legacy APIs, or swapping vendors.

Real-world analogy

A **travel power adapter**: your laptop plug (your code's expected interface) doesn't fit the foreign socket (the third-party API). The adapter converts between them so both work unchanged.

Problem Statement

Your app depends on a clean `PaymentGateway` interface, but the vendor SDK exposes `StripeSdk.createCharge(cents, currencyCode)` with a totally different shape. You'll write a `StripeAdapter implements PaymentGateway` that translates.

Internal Working

- **Target:** the interface your code wants (`PaymentGateway.charge(amount)`).
- **Adaptee:** the existing incompatible class (`StripeSdk`).
- **Adapter:** implements Target, holds/uses the Adaptee, and translates calls/data.
- **Object adapter** (composition — preferred in Dart) vs **class adapter** (inheritance — limited by single inheritance).

Memory Representation

The adapter holds a reference to the adaptee (composition); negligible overhead.

Compiler Behavior

Not applicable. (The adapter satisfies the target interface at compile time so clients depend only on it.)

Runtime Behavior

Each call is forwarded with data conversion (units, shapes, error mapping).

Flutter Engine Behavior

Not applicable. (Adapters are common at plugin/platform-channel boundaries and when wrapping REST/SDK responses into domain models.)

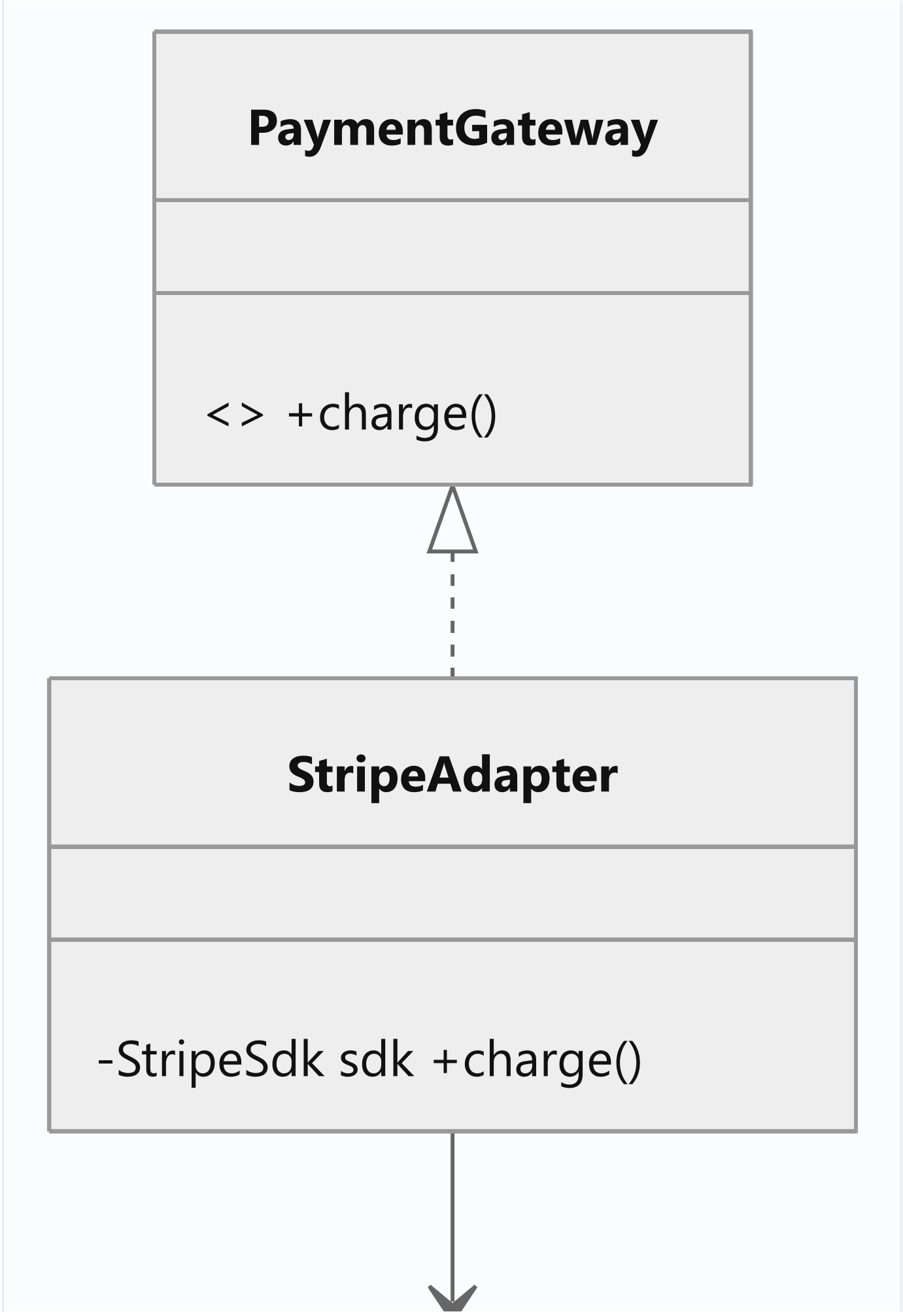
Dart VM Behavior

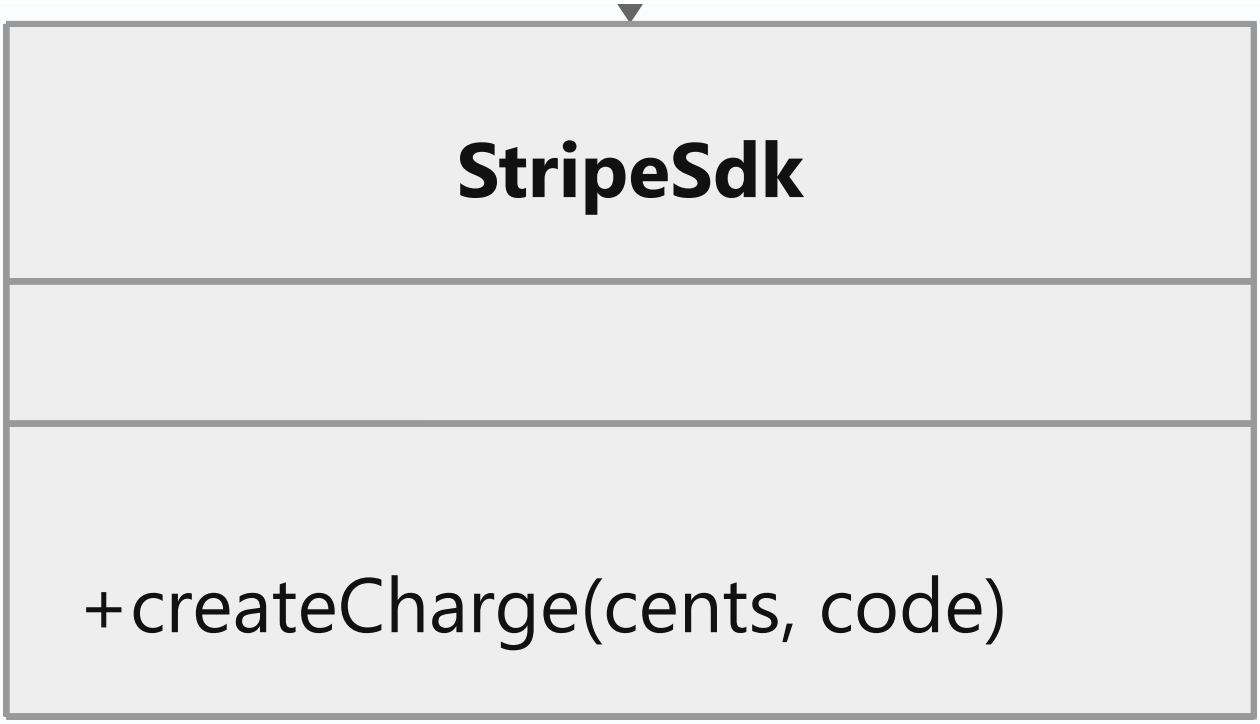
Not applicable.

Examples

```
// Target: what your app expects
abstract interface class PaymentGateway {
  Future<bool> charge({required double amount, required String currency});
}

// Adaptee: incompatible third-party SDK (can't modify)
class StripeSdk {
  Future<String> createCharge(int amountCents, String currencyCode) async {
    return 'ch_123'; // returns a charge id, uses cents + different names
  }
}

// Adapter: translate app interface -> SDK
class StripeAdapter implements PaymentGateway {
  final StripeSdk _sdk;
  StripeAdapter(this._sdk);

  @override
  Future<bool> charge({required double amount, required String currency}) async {
    final cents = (amount * 100).round();               // convert units
    final id = await _sdk.createCharge(cents, currency.toLowerCase());
    return id.startsWith('ch_');                          // map result -> bool
  }
}

Future<void> main() async {
  final PaymentGateway gateway = StripeAdapter(StripeSdk()); // app depends on Target
  print(await gateway.charge(amount: 19.99, currency: 'USD')); // true
}
```

Diagrams


PaymentGateway
<> +charge()
StripeAdapter
-StripeSdk sdk +charge()



Common Mistakes

Mistake	Why	Fix
Leaking the adaptee's types through the target	Recouples clients to the SDK	Convert fully to your domain types
Putting business logic in the adapter	SRP violation	Adapter only translates; logic elsewhere
Class adapter via inheritance in Dart	Single-inheritance limits	Prefer object adapter (composition)
One giant adapter for many SDKs	Low cohesion	One adapter per adaptee

Best Practices

- Prefer **object adapters** (composition) in Dart.
- Fully translate to your **domain types** (units, enums, error models) — don't leak SDK types.
- Keep adapters thin: translation only, no business rules.
- Combine with DIP: your app depends on the target interface; the adapter is wired at the composition root.

Performance

Negligible (a forwarding call + small conversions).

Advantages / Disadvantages

- + Integrate incompatible/legacy/third-party code without changing it; enables vendor swaps; supports DIP.
- − Extra layer; risk of leaking adaptee details if done carelessly.

Interview Questions

1. 🟢 **What does Adapter do?** — Converts one interface into another so incompatible classes can work together.
2. 🟢 **Object vs class adapter?** — Object adapter uses composition (holds the adaptee); class adapter uses inheritance. Dart favors object adapters (single inheritance).
3. 🟡 **How does Adapter support DIP?** — Your app depends on the target interface; the adapter implements it over a concrete SDK, keeping the app decoupled from the vendor.
4. 🟡 **Adapter vs Facade?** — Adapter changes an interface to match an expected one; Facade simplifies a complex subsystem behind a new, easier interface.
5. 🟡 **What should an adapter NOT do?** — Contain business logic; it should only translate shapes/units/errors.
6. 🔴 **How do adapters enable vendor migration?** — Swap the adapter (e.g., Stripe→Razorpay) while the app keeps using the unchanged target interface.
7. 🔴 **Why avoid leaking adaptee types?** — It recouples clients to the third party, defeating the adapter's purpose.

Senior Engineer Tips

- Adapters are the natural home for a data layer's DTO↔domain mapping and error translation.
- Keep one adapter per external system; test it against a fake adaptee to lock the translation.
- Pair with the Repository pattern (20_repository.md) — the repository interface is the target, the adapter/impl wraps the SDK/HTTP.

Architect Perspective

Adapters are the boundary translators of clean architecture: they keep the domain independent of external frameworks/SDKs by converting at the edge. This is what makes vendor swaps, platform differences, and legacy integration manageable without rippling change inward (Modules 40, 16).

Summary

- Adapter wraps an incompatible interface to match what your code expects.

- Prefer object adapters; translate fully to domain types; keep them thin.
- Enables integration, vendor swaps, and DIP; don't leak adaptee details.

Revision Notes

- Adapter = interface converter (Target ← Adapter → Adaptee).
- Object adapter (composition) preferred in Dart.
- Translate units/shapes/errors to domain types; no business logic.
- Adapter changes interface; Facade simplifies subsystem.

Practice Questions

1. Why prefer an object adapter in Dart?
2. How does Adapter differ from Facade?
3. What happens if the adapter leaks SDK types to callers?

Coding Questions

1. Adapt a legacy `XmlLogger.writeLine(String)` to a modern `Logger.log(level, msg)` interface.
2. Wrap two weather SDKs behind one `WeatherService` interface via adapters.
3. Adapt a callback-based API to a `Future`-returning target interface.

Mini Project

Payment gateway adapters (pure Dart): Define a `PaymentGateway` target; implement `StripeAdapter` and `RazorpayAdapter` over two mismatched fake SDKs; write a checkout that depends only on the target and swap gateways via injection. Acceptance: no SDK type leaks; unit conversions/error mapping done in adapters; swapping vendors edits only wiring; `dart analyze` clean.

Decorator Pattern

> A decorator wraps an object to add behavior dynamically, keeping the same interface — layering features without subclass explosion.

Introduction

Decorator attaches responsibilities to an object at runtime by wrapping it in another object that implements the same interface and delegates to it, adding behavior before/after. Stack decorators to compose features (logging + caching + retry over a base service).

Why this concept exists

Adding every feature combination via subclasses explodes (`LoggingCachingRetryRepository` ...). Decorators let you **compose** features independently and stack them in any order, honoring the Open/Closed Principle — add a new decorator without touching existing ones.

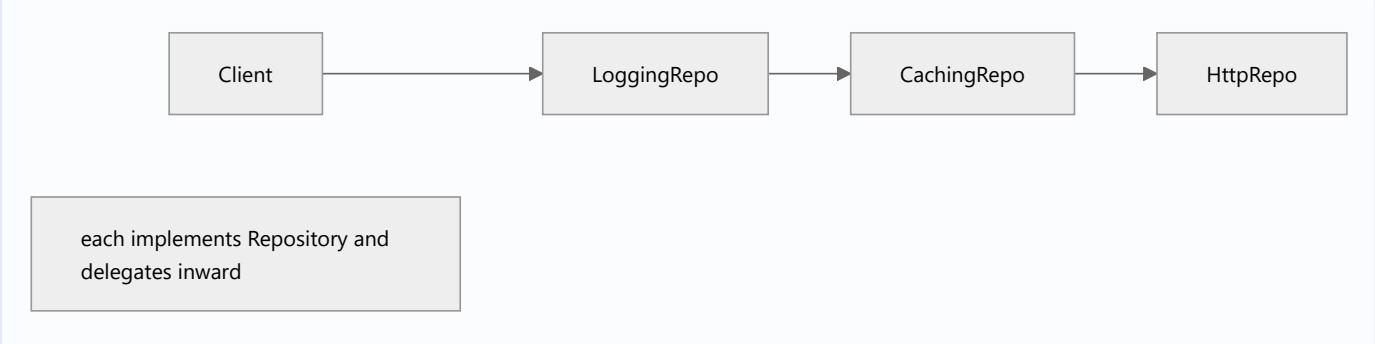
Real-world analogy

Coffee add-ons: start with espresso, wrap with "add milk," then "add caramel," then "add whip." Each wrapper is still "a drink" (same interface) and adds cost/behavior. You mix and match without a class per combination.

Problem Statement

You have a `Repository` and want to optionally add logging, caching, and retry — in different combinations per environment — without a subclass per combo. You'll wrap the base repository in stackable decorators.

Internal Working



- **Component interface** (`Repository`) shared by the base and all decorators.

- **Concrete component** (`HttpRepository`) does the real work.
- **Decorator** implements the interface, holds a component, delegates, and adds behavior.
- Decorators **stack**: each wraps the next; call order follows the wrapping order.

Memory Representation

A chain of wrapper objects, each referencing the inner one; linear in the number of decorators.

Compiler Behavior

Not applicable. (All layers share the component interface, so clients are agnostic.)

Runtime Behavior

A call traverses the wrapper chain outermost→innermost, each adding pre/post behavior around `super` /inner call.

Flutter Engine Behavior

Not applicable. (Flutter's *widget* composition is decorator-like: `Padding(child: DecoratedBox(child: ...))` layers visual behavior around a child.)

Dart VM Behavior

Not applicable.

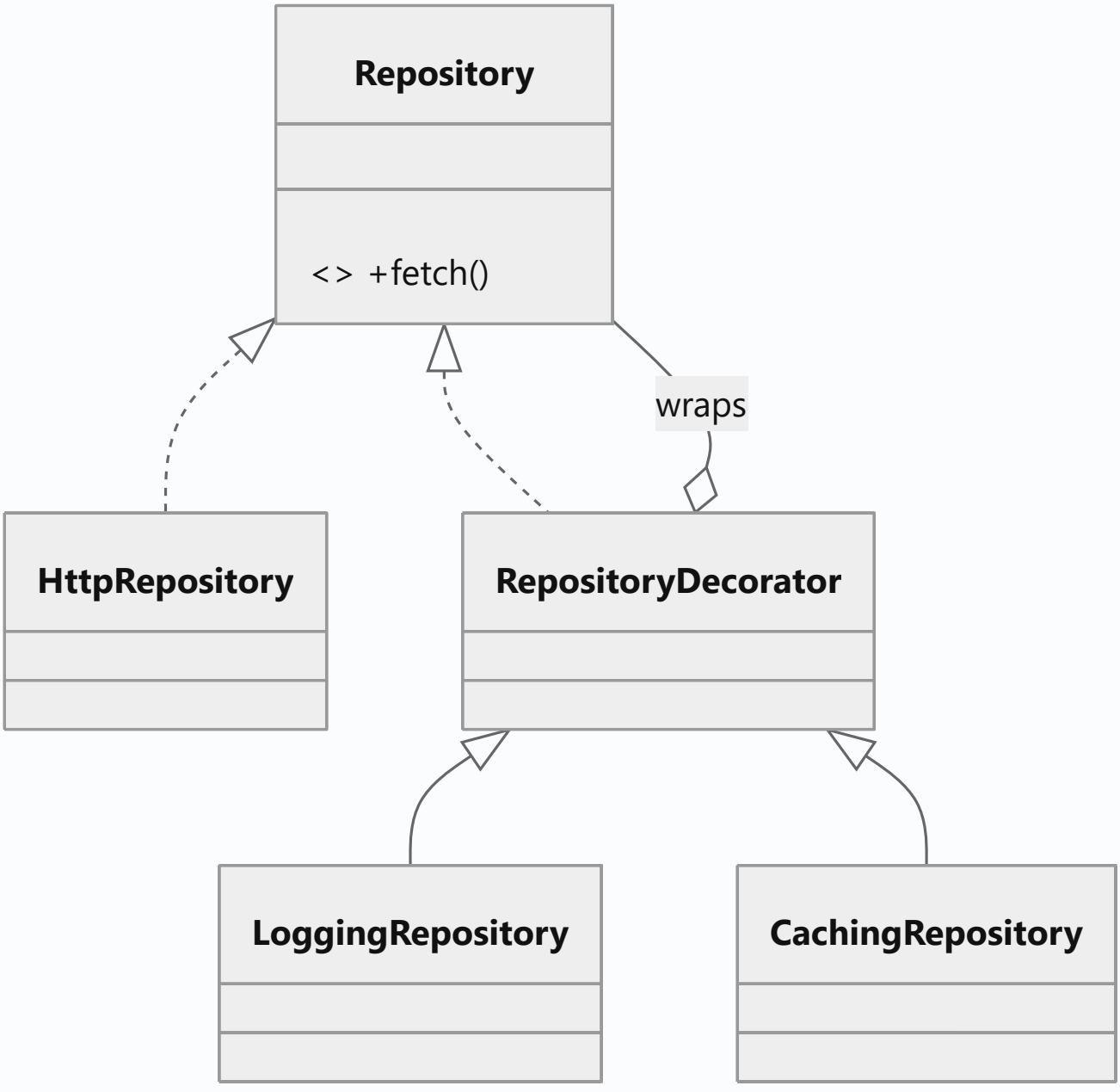
Examples

```
abstract interface class Repository {
  Future<String> fetch(String id);
}

class HttpRepository implements Repository {
  @override
  Future<String> fetch(String id) async => 'data($id)';
}

// Base decorator: delegates by default
abstract class RepositoryDecorator implements Repository {
  final Repository inner;
  RepositoryDecorator(this.inner);
```

```
}

class LoggingRepository extends RepositoryDecorator {
  LoggingRepository(super.inner);
  @override
  Future<String> fetch(String id) async {
    print('-> fetch $id');
    final result = await inner.fetch(id);
    print('<- $result');
    return result;
  }
}

class CachingRepository extends RepositoryDecorator {
  CachingRepository(super.inner);
  final _cache = <String, String>{};
  @override
  Future<String> fetch(String id) async =>
      _cache[id] ??= await inner.fetch(id); // add caching
}

class RetryRepository extends RepositoryDecorator {
  final int times;
  RetryRepository(super.inner, {this.times = 3});
  @override
  Future<String> fetch(String id) async {
    for (var i = 0; i < times; i++) {
      try {
        return await inner.fetch(id);
      } catch (_) {
        if (i == times - 1) rethrow;
      }
    }
    throw StateError('unreachable');
  }
}

Future<void> main() async {
  // stack features in any order — no subclass per combination
  final Repository repo =
```

```
      LoggingRepository(CachingRepository(RetryRepository(HttpRepository())));
  print(await repo.fetch('42'));
  print(await repo.fetch('42')); // second call served from cache
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Subclass explosion for feature combos	Unmaintainable	Compose decorators
Decorators changing the interface	Breaks transparency	Keep the same interface
Order-dependent bugs (cache before retry?)	Wrong wrapping order	Decide/ document ordering intent
Heavy logic making the chain slow	Latency per layer	Keep decorators focused/light

Best Practices

- All decorators share the **component interface** (transparent wrapping).
- Keep each decorator **single-purpose** (SRP): one concern per wrapper.
- Be intentional about **stacking order** (e.g., retry inside cache vs outside).
- Provide a base delegating decorator to reduce boilerplate.

Performance

Each layer adds a delegation hop; keep decorators light. Caching/retry decorators can *improve* net performance/resilience.

Advantages / Disadvantages

- **+** Compose behavior at runtime, avoid subclass explosion, OCP-friendly, single-purpose layers.
- **−** Many small classes; deep chains harder to debug; order sensitivity.

Interview Questions

1. 🟢 **What does Decorator do?** — Adds behavior to an object dynamically by wrapping it in an object of the same interface that delegates and augments.
2. 🟢 **Decorator vs inheritance for adding behavior?** — Decorator composes at runtime and avoids the subclass explosion of every feature combination.
3. 🟡 **Why must decorators share the component interface?** — So they're transparent — clients treat a decorated object like the original.
4. 🟡 **Give a real stack example.** — `Logging(Caching(Retry(HttpRepository)))` — cross-cutting concerns layered over a base.
5. 🟡 **Why does stacking order matter?** — Behavior differs (e.g., caching a retried result vs retrying a cache miss); order encodes intent.

6. 🔴 **Decorator vs Proxy?** — Both wrap and share the interface; Decorator *adds behavior*, Proxy *controls access* (lazy/remote/protection) — intent differs.
7. 🔴 **Where is Decorator visible in Flutter?** — Widget composition layering visual/behavioral wrappers around a child (`Padding` , `Opacity` , `DecoratedBox`).

Senior Engineer Tips

- Decorators are the cleanest way to add cross-cutting concerns (logging, caching, metrics, retry) to repositories/services without editing them.
- Keep the base delegating decorator so new decorators override only what they change.
- Document the intended composition order; wrap it in a factory so call sites don't get it wrong.

Architect Perspective

Decorator enables cross-cutting concerns as composable layers, keeping core services clean and OCP-compliant. It's a key tool in data/service layers (resilience, observability) and mirrors Flutter's compositional UI philosophy — behavior by wrapping, not by inheritance (Modules 16, 21, 39).

Summary

- Decorator wraps an object (same interface) to add behavior; stack for composition.
- Avoids subclass explosion; keep layers single-purpose; mind ordering.
- Ideal for cross-cutting concerns over repositories/services; mirrors Flutter widget composition.

Revision Notes

- Decorator = wrap + same interface + add behavior; stackable.
- Avoids subclass explosion (OCP); one concern per decorator.
- Order matters; base delegating decorator reduces boilerplate.
- Decorator adds behavior; Proxy controls access.

Practice Questions

1. Why does Decorator avoid a subclass per feature combination?
2. How does caching-then-retry differ from retry-then-caching?
3. How is Flutter widget nesting decorator-like?

Coding Questions

1. Add a `MetricsRepository` decorator timing each `fetch` .

2. Build a stackable `Stream` decorator adding logging to events.
3. Wrap a `Logger` with a `TimestampLogger` and a `LevelFilterLogger`.

Mini Project

Resilient repository stack (pure Dart): Implement `Repository` with `HttpRepository` and stackable `Logging` / `Caching` / `Retry` decorators, plus a factory that composes them in the correct order per environment. Test each decorator in isolation and the full stack. Acceptance: no subclass-per-combo; decorators single-purpose; order documented; `dart analyze` clean.

Facade Pattern

> A facade provides a single, simple interface over a complex subsystem — hiding the messy coordination of many parts behind one easy entry point.

Introduction

Facade offers a high-level, unified API that delegates to a set of lower-level classes. Clients use the simple facade instead of orchestrating many subsystem objects themselves.

Why this concept exists

Complex operations often require coordinating several components (validate → charge → persist → notify). Making every caller wire those together is repetitive and coupling-heavy. A facade centralizes that orchestration behind one method, simplifying usage and decoupling clients from the subsystem's structure.

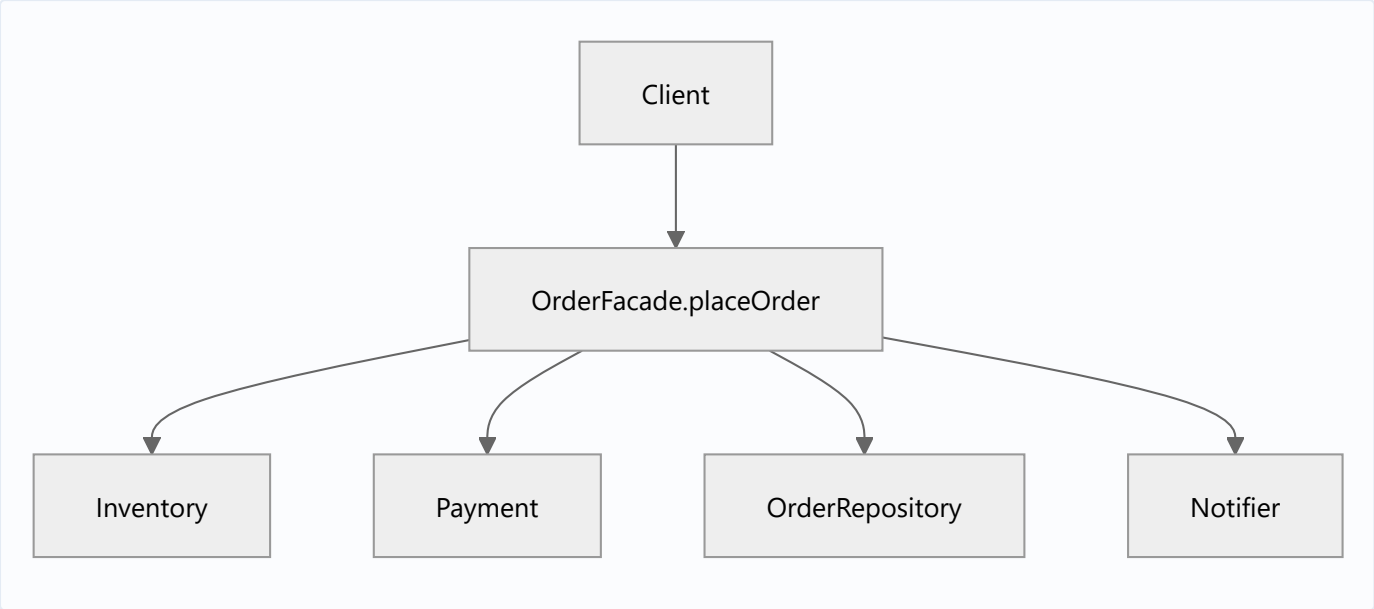
Real-world analogy

A **hotel concierge**: instead of you calling housekeeping, the restaurant, and the taxi company separately, you ask the concierge, who coordinates everything. One simple interface over many services.

Problem Statement

Placing an order requires inventory checks, payment, persistence, and notifications across four subsystems. Callers shouldn't orchestrate all four. You'll expose an `OrderFacade.placeOrder(...)` that hides the coordination.

Internal Working



- The **facade** holds/uses subsystem components and exposes coarse-grained operations.
- It **doesn't** prevent direct subsystem access — it's a convenience layer, not a hard boundary.
- Often combined with DIP: the facade depends on subsystem *interfaces* and is itself injectable.

Memory Representation

The facade holds references to subsystem components; negligible.

Compiler Behavior

Not applicable.

Runtime Behavior

One facade call fans out to ordered subsystem calls, handling their sequencing and errors.

Flutter Engine Behavior

Not applicable. (Service classes/use cases in Flutter apps are often facades over repositories + clients; `SharedPreferences` is a facade over platform storage.)

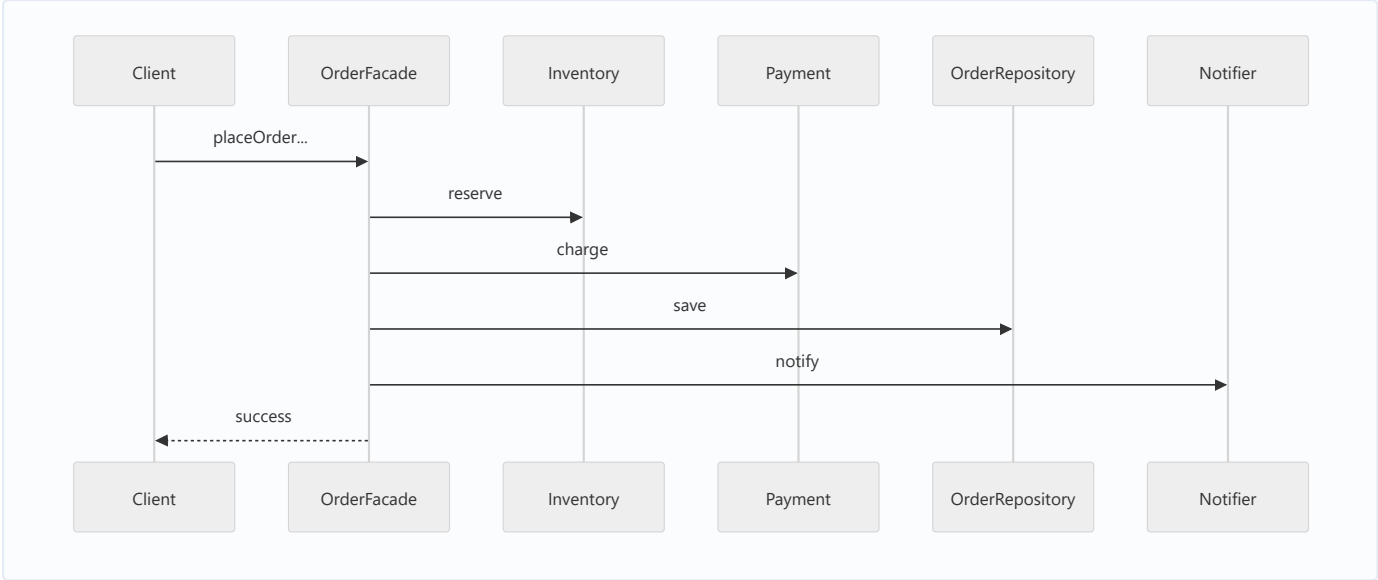
Dart VM Behavior

Not applicable.

Examples

```
// Subsystems (each independently testable)
class Inventory {
  bool reserve(String sku, int qty) => qty <= 10;
}
class Payment {
  Future<bool> charge(double amount) async => true;
}
class OrderRepository {
  Future<void> save(String order) async => print('saved $order');
}
class Notifier {
  Future<void> notify(String to) async => print('notified $to');
}

// Facade: one simple entry point over the messy coordination
class OrderFacade {
  final Inventory inventory;
  final Payment payment;
  final OrderRepository repo;
  final Notifier notifier;
  OrderFacade(this.inventory, this.payment, this.repo, this.notifier);

  Future<bool> placeOrder(String sku, int qty, double amount, String customer) async {
    if (!inventory.reserve(sku, qty)) return false;       // 1 reserve
    if (!await payment.charge(amount)) return false;       // 2 pay
    await repo.save('$sku x$qty');                         // 3 persist
    await notifier.notify(customer);                       // 4 notify
    return true;
  }
}

Future<void> main() async {
  final facade = OrderFacade(Inventory(), Payment(), OrderRepository(), Notifier());
  print(await facade.placeOrder('BOOK1', 2, 39.98, 'ada@x.com')); // true (one call)
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Facade becoming a God object	Absorbs all logic (SRP violation)	Delegate to subsystems; keep facade orchestration-only
Facade with business rules baked in	Hard to reuse/test	Push rules into subsystem/domain services
Hiding subsystems you still need directly	Over-restriction	Facade is convenience, not a wall — allow direct access when needed
Facade depending on concrete subsystems	Coupling	Depend on subsystem interfaces (DIP)

Best Practices

- Keep the facade **thin**: orchestration/sequencing, not business logic.
- Depend on subsystem **interfaces**; make the facade injectable.
- Provide coarse-grained operations that match real use cases.
- Don't forbid direct subsystem use — facade is a simplification layer.

Performance

Neutral; it just sequences existing calls.

Advantages / Disadvantages

- **+** Simplifies client code, decouples clients from subsystem structure, centralizes orchestration.
- **−** Risk of becoming a God object; can hide capabilities; another layer.

Interview Questions

1. 🟢 **What does Facade do?** — Provides a simple unified interface over a complex subsystem, delegating to its parts.
2. 🟢 **Facade vs Adapter?** — Facade *simplifies* a subsystem behind a new easier API; Adapter *converts* one interface to another expected one.
3. 🟡 **What should a facade NOT contain?** — Heavy business logic; it should orchestrate, delegating rules to subsystem/domain services.
4. 🟡 **Does a facade block direct subsystem access?** — No; it's a convenience layer, not an enforced boundary.
5. 🟡 **How does Facade relate to the Use Case/Interactor idea?** — A use case is often a facade over repositories/services for one application operation (Clean Architecture).
6. 🔴 **How do you keep a facade from becoming a God object?** — Keep it orchestration-only, depend on interfaces, and split facades by cohesive area.
7. 🔴 **Facade vs Mediator?** — Facade is a one-way simplifier over subsystems; Mediator coordinates *bidirectional* interactions among peers.

Senior Engineer Tips

- Application **use cases/interactors** are the archetypal facade in clean architecture — one operation, orchestrating repositories/services (Module 40).
- Keep facades injectable and interface-dependent so they're testable and swappable.
- Resist creeping logic into the facade; when it grows, split by domain area.

Architect Perspective

Facades define the application's coarse-grained API surface — the operations the UI actually needs — decoupling presentation from the intricate wiring of data/services. This is the role of use cases in layered/clean architecture and keeps the UI thin and the subsystem replaceable (Modules 40, 43).

Summary

- Facade = one simple interface over a complex subsystem; delegates and orchestrates.
- Keep it thin and interface-dependent; it simplifies but doesn't wall off subsystems.
- Application use cases are facades; contrast with Adapter (convert) and Mediator (coordinate peers).

Revision Notes

- Facade = simple unified API over complex subsystem (orchestration only).
- Facade simplifies; Adapter converts; Mediator coordinates peers.
- Keep thin, interface-dependent, injectable; use-case = facade.
- Risk: God object — split by area.

Practice Questions

1. How does Facade differ from Adapter and Mediator?
2. Why keep business logic out of the facade?
3. How is a Clean Architecture use case a facade?

Coding Questions

1. Build a `MediaFacade` over `Downloader`, `Decoder`, and `Cache` exposing `loadImage(url)`.
2. Create a `CheckoutUseCase` facade orchestrating cart, payment, and order repos.
3. Wrap a multi-step auth flow (validate/login/store token) behind an `AuthFacade`.

Mini Project

Order placement facade (pure Dart): Implement `Inventory`, `Payment`, `OrderRepository`, `Notifier` behind interfaces and an `OrderFacade.placeOrder` orchestrating them, injected via constructor. Test with fakes. Acceptance: facade contains only orchestration; depends on interfaces; subsystems independently testable; `dart analyze` clean.

Bridge Pattern

> Bridge decouples an abstraction from its implementation so the two can vary independently — preventing a combinatorial explosion of subclasses.

Introduction

Bridge splits a hierarchy into two: an **abstraction** (the high-level control) and an **implementation** (the low-level engine), connected by composition. Each can be extended on its own axis without multiplying classes.

Why this concept exists

When a class varies along **two independent dimensions** (e.g., shape *type* × rendering *API*), inheritance forces a class per combination (`CircleSvg`, `CircleCanvas`, `SquareSvg`, `SquareCanvas`...). Bridge composes the two dimensions instead, so `M shapes × N renderers` needs `M + N` classes, not `M × N`.

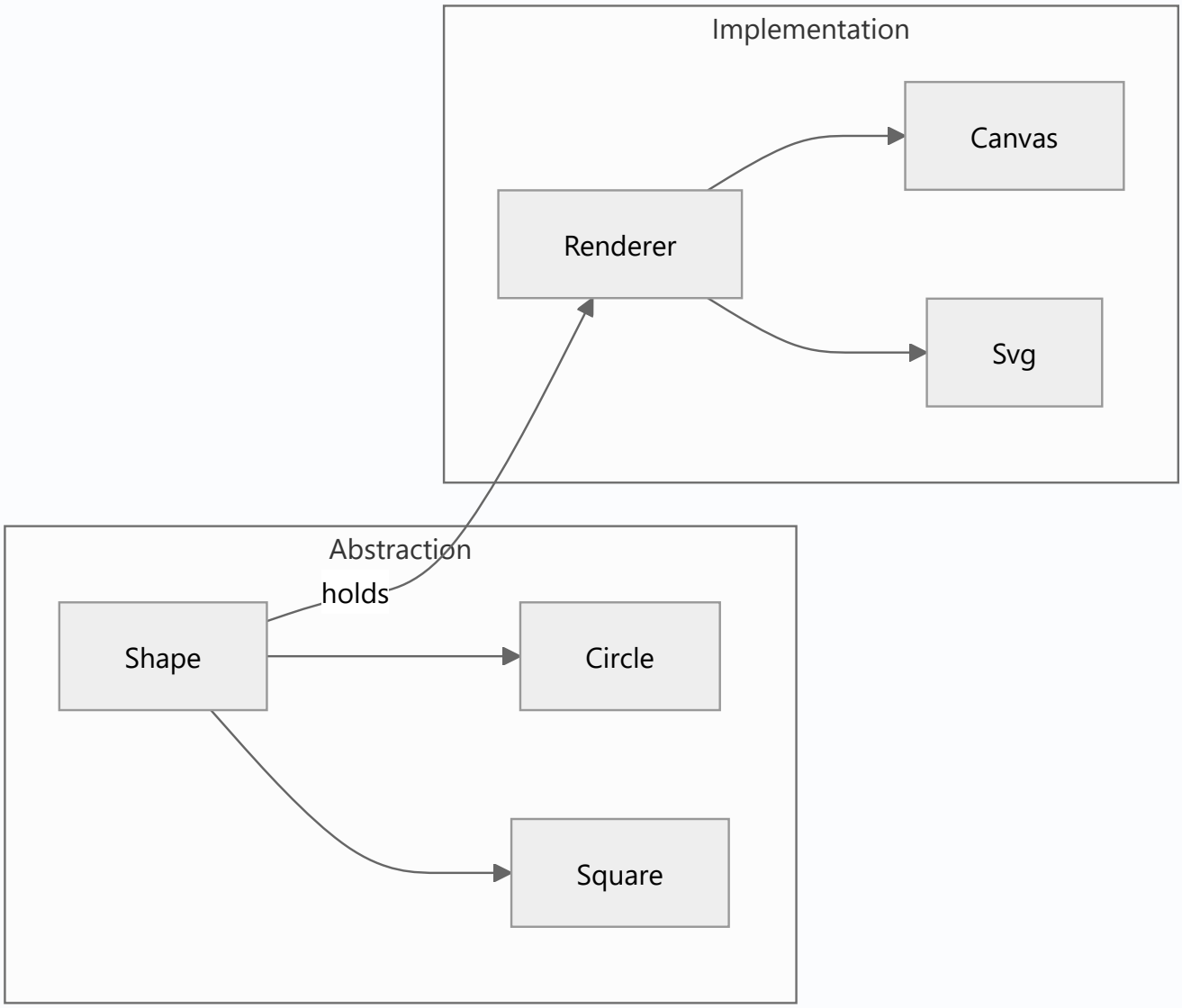
Real-world analogy

A **TV and its remote**: the remote (abstraction) works with any brand of TV (implementation). Add a new remote type or a new TV brand independently — you don't build a specific remote for every TV model.

Problem Statement

You have shapes (`Circle`, `Square`) that must render on multiple backends (`Canvas`, `SVG`). Subclassing per combination explodes. You'll bridge `Shape` (abstraction) to `Renderer` (implementation) via composition.

Internal Working



- **Abstraction** holds a reference to an **Implementor** interface and delegates the low-level work to it.
- Both hierarchies extend independently.
- It's composition-over-inheritance applied to a two-axis variation problem.

Memory Representation

The abstraction holds one reference to an implementor; negligible.

Compiler Behavior / Runtime Behavior

Not special — delegation via an interface. Abstraction methods call implementor methods at runtime.

Flutter Engine Behavior

Not applicable directly. (Conceptually similar to how a `RenderObject` delegates painting to a `Canvas`, or how platform-agnostic widgets delegate to platform renderers.)

Dart VM Behavior

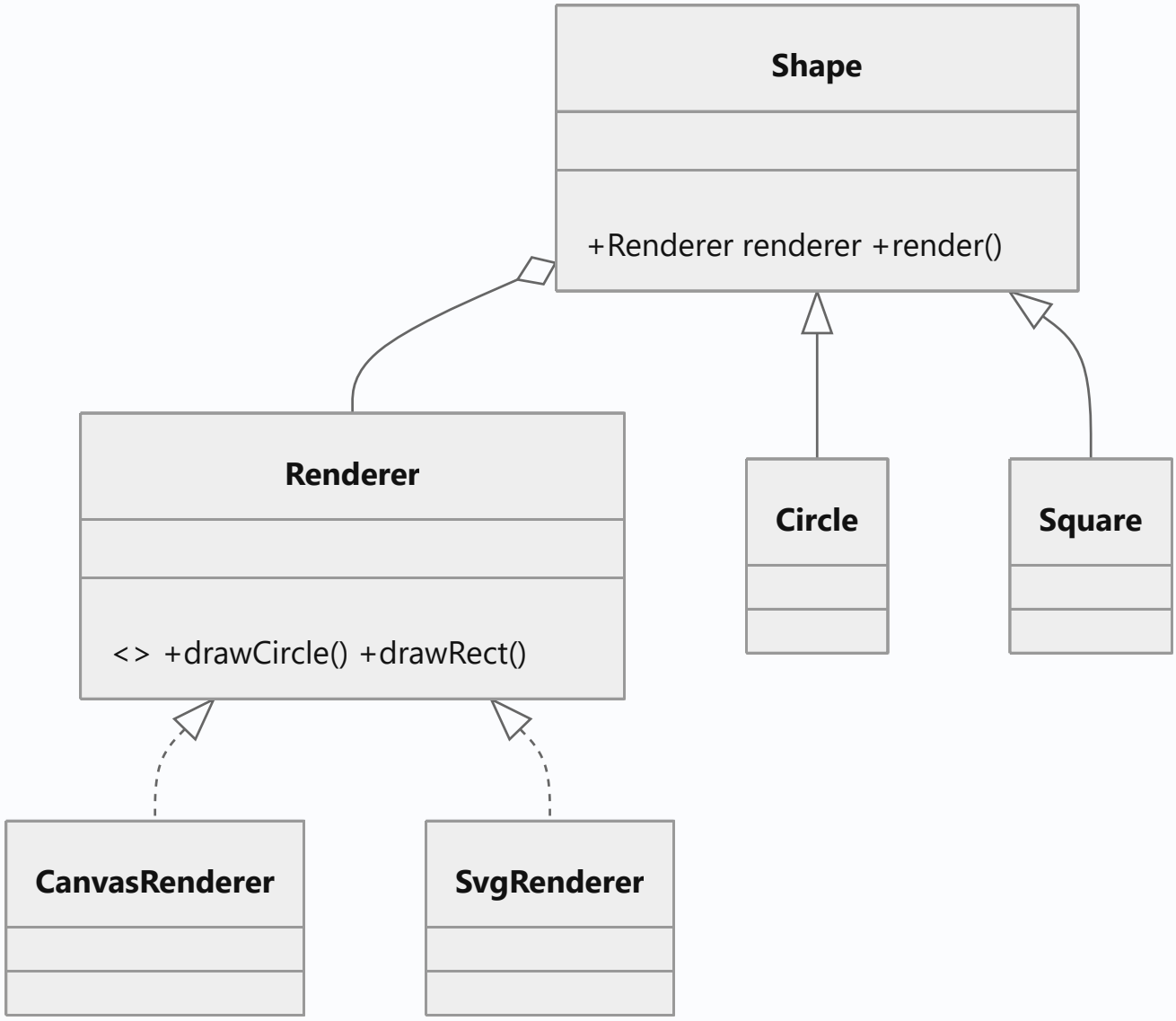
Not applicable.

Examples

```
// Implementor axis
abstract interface class Renderer {
  String drawCircle(double r);
  String drawRect(double w, double h);
}
class CanvasRenderer implements Renderer {
  @override
  String drawCircle(double r) => 'Canvas: circle r=$r';
  @override
  String drawRect(double w, double h) => 'Canvas: rect ${w}x$h';
}
class SvgRenderer implements Renderer {
  @override
  String drawCircle(double r) => '<circle r="$r"/>';
  @override
  String drawRect(double w, double h) => '<rect w="$w" h="$h"/>';
}

// Abstraction axis (bridged to Renderer via composition)
abstract class Shape {
  final Renderer renderer;
  Shape(this.renderer);
  String render();
}
class Circle extends Shape {
  final double r;
  Circle(super.renderer, this.r);
  @override
  String render() => renderer.drawCircle(r);
```

```
}
class Square extends Shape {
  final double side;
  Square(super.renderer, this.side);
  @override
  String render() => renderer.drawRect(side, side);
}

void main() {
  // Mix any shape with any renderer — no CircleSvg/SquareCanvas classes needed
  print(Circle(CanvasRenderer(), 2).render()); // Canvas: circle r=2.0
  print(Circle(SvgRenderer(), 2).render());     // <circle r="2.0"/>
  print(Square(SvgRenderer(), 3).render());      // <rect w="3.0" h="3.0"/>
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
One subclass per combination	Class explosion	Bridge the two axes via composition
Confusing Bridge with Adapter	Different intent	Bridge is designed up-front; Adapter fixes an existing mismatch
Bridging when there's only one axis of change	Over-engineering	Use plain inheritance/composition

Best Practices

- Use Bridge when there are **two (or more) independent axes** of variation.

- Design both hierarchies to interfaces; connect via composition.
- Don't apply it speculatively — only when the second axis is real.

Performance

Negligible (one delegation hop).

Advantages / Disadvantages

- **+** Independent extension of both axes, avoids `M×N` explosion, composition-based flexibility.
- **−** More upfront structure; unnecessary for single-axis variation.

Interview Questions

1. 🟢 **What does Bridge decouple?** — An abstraction from its implementation so both can vary independently.
2. 🟢 **What problem does it prevent?** — The `M×N` subclass explosion when a class varies on two independent dimensions.
3. 🟡 **Bridge vs Adapter?** — Bridge is designed up front to separate two axes; Adapter retrofits compatibility onto an existing mismatched interface.
4. 🟡 **Bridge vs Strategy?** — Structurally similar (composition + interface), but Strategy swaps an *algorithm*; Bridge separates an *abstraction hierarchy* from an *implementation hierarchy*.
5. 🟡 **When is Bridge overkill?** — With a single axis of variation — plain composition/inheritance suffices.
6. 🔴 **How does Bridge relate to "composition over inheritance"?** — It's that principle applied to convert a two-axis inheritance explosion into composition.
7. 🔴 **Give a real two-axis example.** — Shapes × renderers, messages × transport channels, documents × export formats.

Senior Engineer Tips

- Spot Bridge opportunities when class names start combining two concepts (`PdfInvoice` , `HtmlInvoice` , `PdfReport` , `HtmlReport`) — separate the axes.
- Bridge and Strategy often look identical in code; the distinction is intent/scale (algorithm swap vs two-hierarchy decoupling).

Architect Perspective

Bridge keeps orthogonal concerns independently evolvable — e.g., business abstractions vs platform/transport/rendering implementations. This supports multi-platform and multi-backend designs without exponential class growth, aligning with clean-architecture separation of policy and mechanism.

Summary

- Bridge separates abstraction from implementation via composition so both vary independently.
- Prevents `M×N` subclass explosion; use only with genuine two-axis variation.
- Structurally like Strategy; differs in intent (two hierarchies vs algorithm swap).

Revision Notes

- Bridge = abstraction ⊥ implementation via composition (avoid M×N).
- Two independent axes → M+N classes, not M×N.
- Bridge (design-time, two hierarchies) vs Adapter (retrofit) vs Strategy (algorithm swap).

Practice Questions

1. Why does bridging shapes×renderers give M+N instead of M×N classes?
2. How is Bridge different from Adapter in intent?
3. When is Bridge unnecessary?

Coding Questions

1. Bridge `Message` (`Text` / `Rich`) to `Channel` (`Email` / `Sms`).
2. Bridge `Report` (`Summary` / `Detailed`) to `Exporter` (`Pdf` / `Csv`).
3. Add a new renderer to the shapes example and show zero shape edits.

Mini Project

Notification bridge (pure Dart): Bridge a `Notification` abstraction (`Alert` , `Reminder`) to a `Channel` implementation (`Email` , `Sms` , `Push`); render any combination without per-combo classes; add a new channel with no notification edits. Acceptance: M+N structure; both axes extend independently; `dart analyze` clean.

Composite Pattern

> Composite lets you treat individual objects and compositions of objects uniformly — a tree where leaves and branches share one interface.

Introduction

Composite arranges objects into **tree structures** and lets clients treat a single leaf and a whole subtree the same way through a common interface. Operations recurse through the tree automatically.

Why this concept exists

Hierarchical structures (file systems, UI trees, org charts, menus) need uniform operations (`size()` , `render()` , `total()`) whether applied to one item or a group. Without Composite, clients must constantly distinguish "is this a single thing or a group?" — branching everywhere. Composite erases that distinction.

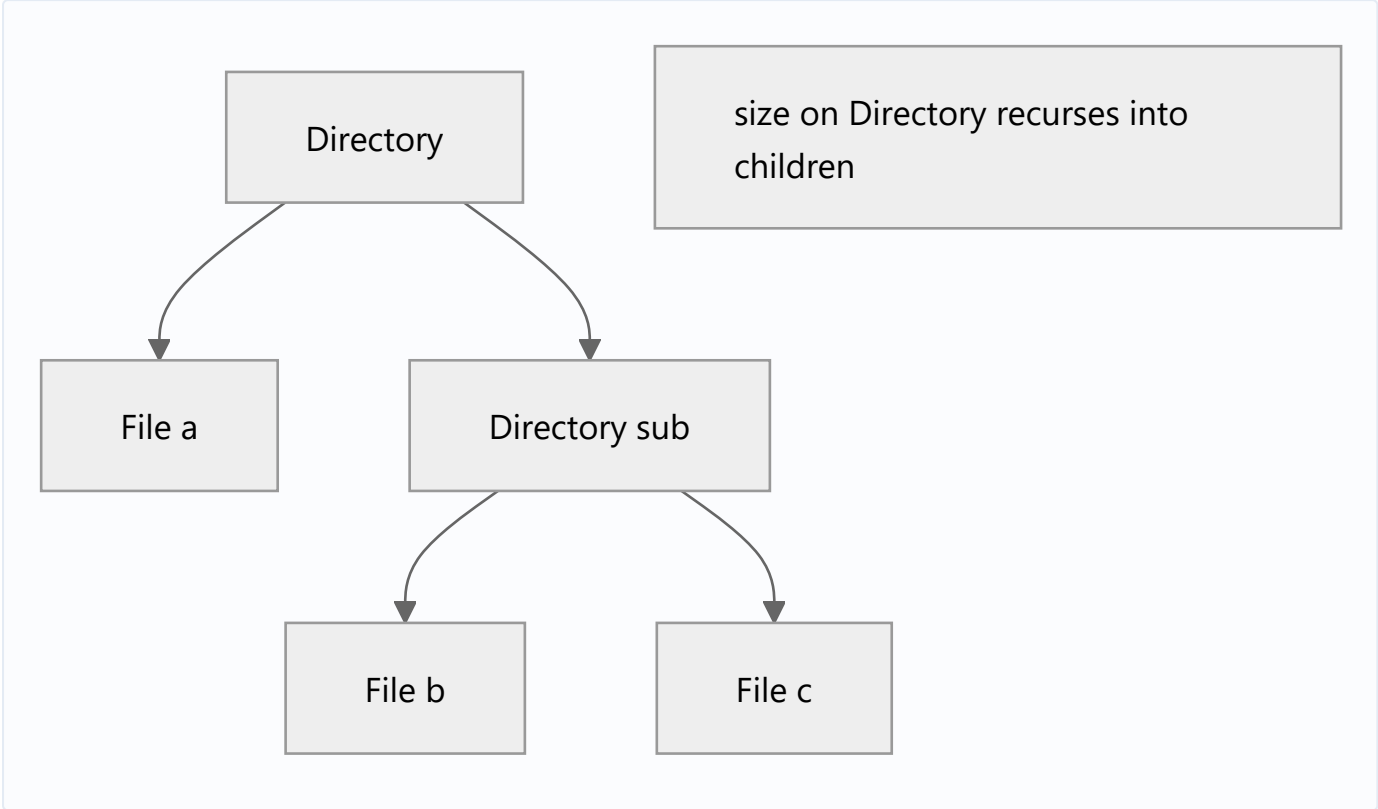
Real-world analogy

A **folder**: it contains files and other folders. Asking "how big are you?" works the same on a file (its own size) or a folder (sum of children). You don't care which — you just ask.

Problem Statement

Model a file system where `File` and `Directory` both answer `size()` , and a directory's size is the sum of its children (recursively). You'll define one `FileSystemNode` interface for both.

Internal Working



- **Component:** the shared interface (`FileSystemNode` with `size()`).
- **Leaf:** a node with no children (`File`).
- **Composite:** a node holding children (`Directory`), implementing operations by aggregating children's results.
- Operations recurse through the composite transparently.

Memory Representation

A tree of node objects; a composite holds a list of child references. Depth affects recursion stack.

Compiler Behavior / Runtime Behavior

Not special. Recursive traversal at runtime; watch for very deep trees (stack depth) — use iteration if needed.

Flutter Engine Behavior

Highly relevant conceptually: the **widget tree** is a composite — a `Column` (composite) holds children; layout/paint recurse uniformly over leaves (`Text`) and branches (Module 09).

Dart VM Behavior

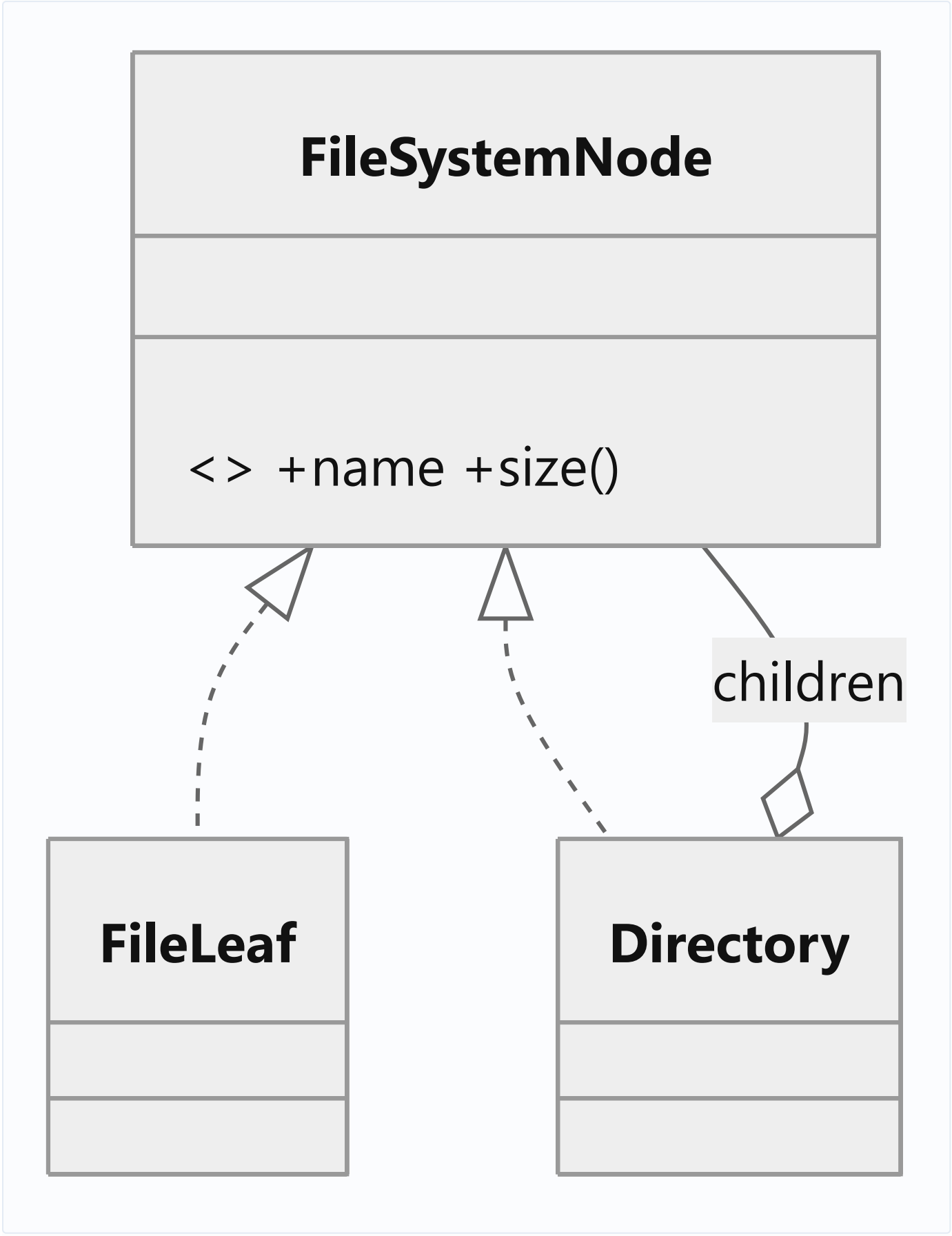
Not applicable.

Examples

```
abstract interface class FileSystemNode {
  String get name;
  int size(); // bytes
}

class FileLeaf implements FileSystemNode {
  @override
  final String name;
  final int bytes;
  FileLeaf(this.name, this.bytes);
  @override
  int size() => bytes;
}

class Directory implements FileSystemNode {
  @override
  final String name;
  final List<FileSystemNode> children = [];
  Directory(this.name);

  Directory add(FileSystemNode node) { children.add(node); return this; }

  @override
  int size() => children.fold(0, (sum, c) => sum + c.size()); // recurse uniformly
}

void main() {
  final root = Directory('root')
    ..add(FileLeaf('a.txt', 100))
    ..add(Directory('sub')
      ..add(FileLeaf('b.txt', 200))
      ..add(FileLeaf('c.txt', 300)));

  // treat leaf and composite identically:
```

```
  print(root.size());                    // 600
  print(FileLeaf('x.txt', 50).size());   // 50
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Different interfaces for leaf vs composite	Clients must branch	Share one component interface
Leaf forced to implement child ops (`add`)	Awkward/ `UnsupportedError` (ISP/LSP issue)	Keep child-management on composite only, or use safe defaults
Deep recursion on huge trees	Stack overflow	Iterative traversal / bounded depth
Cyclic references	Infinite recursion	Enforce a true tree (no cycles)

Best Practices

- Share **one component interface** for leaves and composites.
- Keep **child-management methods** (`add` / `remove`) on the composite (not forced on leaves) to respect ISP/LSP.
- Guard against cycles; consider iteration for very deep trees.
- Combine with Visitor (18_visitor.md) to add operations over the tree without editing node classes.

Performance

Traversal is O(nodes); recursion depth = tree depth. Fine for typical trees; iterate for pathological depth.

Advantages / Disadvantages

- **+** Uniform treatment of parts and wholes, recursive operations for free, natural for hierarchies.
- **−** Leaf/composite interface tension (child ops), risk of overly general interfaces, recursion depth.

Interview Questions

1. 🟢 **What does Composite do?** — Lets clients treat individual objects and groups uniformly via a shared interface in a tree structure.
2. 🟢 **Leaf vs Composite?** — Leaf has no children; Composite holds children and implements operations by aggregating them.
3. 🟡 **Where should `add` / `remove` live?** — On the composite; forcing them on leaves creates `UnsupportedError` stubs (ISP/LSP smell).
4. 🟡 **Give a Flutter example of Composite.** — The widget tree: containers hold children; layout/paint recurse over leaves and branches uniformly.

5. 🟡 **How do Composite and Visitor combine?** — Visitor adds new operations over the composite tree without modifying node classes.
6. 🔴 **What are the risks with large/deep trees?** — Stack overflow from recursion and cycle-induced infinite loops; use iteration and enforce acyclicity.
7. 🔴 **How does Composite support Open/Closed?** — New node types plug into the tree implementing the component interface, without changing traversal code.

Senior Engineer Tips

- Composite pairs naturally with recursion and `fold`; keep node operations pure for easy testing.
- When you need many operations over the tree, add a Visitor rather than bloating node classes.
- Enforce tree invariants (no cycles, single parent) to keep traversal safe.

Architect Perspective

Composite models any hierarchical domain (UI trees, category trees, permission trees, document structures) with uniform operations, keeping client code simple and extensible. It underlies Flutter's rendering model and is foundational for tree-shaped data throughout apps (Module 09).

Summary

- Composite = tree of leaves + composites sharing one interface; operations recurse uniformly.
- Keep child-management on composites; guard depth/cycles.
- Powers the Flutter widget tree; pairs with Visitor for added operations.

Revision Notes

- Composite = uniform leaf/branch via shared interface (tree).
- Composite holds children + aggregates; leaf has none.
- `add` / `remove` on composite (ISP/LSP); watch recursion depth/cycles.
- Flutter widget tree = composite; pairs with Visitor.

Practice Questions

1. Why keep `add` / `remove` off the leaf?
2. How is the widget tree a composite?
3. What risks come with deep/cyclic trees?

Coding Questions

1. Add a `count()` (number of files) to the file-system composite.
2. Model a UI menu tree (`MenuItem` / `Menu`) with a uniform `render()` .
3. Add a `find(name)` search across the composite tree.

Mini Project

File system model (pure Dart): Implement `FileSystemNode` with `File` / `Directory` , supporting `size()` , `count()` , and `find(name)` recursively, with cycle guards. Add tests over a multi-level tree. Acceptance: uniform interface; child ops only on directories; safe on deep trees; `dart analyze` clean.

Proxy Pattern

> A proxy is a stand-in that implements the same interface as a real object and controls access to it — for lazy loading, caching, access control, or remote calls.

Introduction

Proxy provides a surrogate for another object, sharing its interface, to control access. Common flavors: **virtual** (lazy/expensive init), **caching**, **protection** (authorization), and **remote** (network stand-in).

Why this concept exists

Sometimes you want to interpose logic *around* using an object — defer its costly creation until needed, cache results, check permissions, or represent a remote resource locally — without changing the object or its callers. Proxy inserts that control transparently behind the same interface.

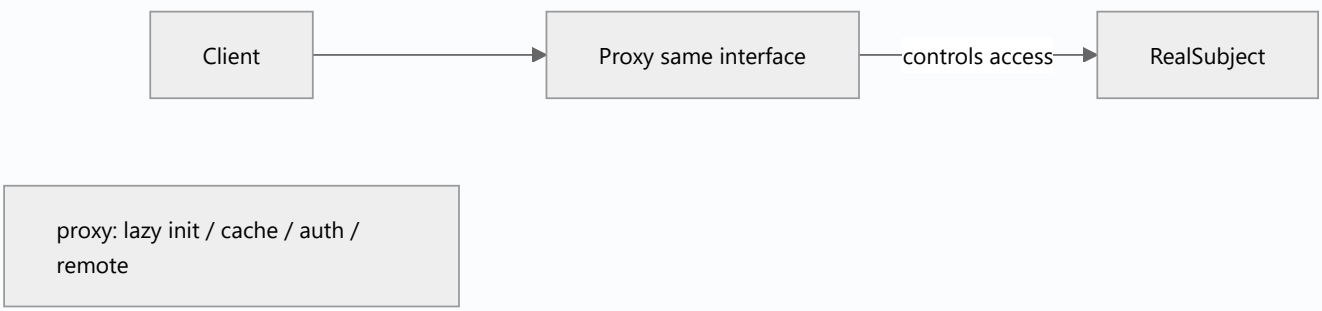
Real-world analogy

A **debit card** is a proxy for your bank account: same "pay" interface, but it controls access (PIN check = protection), and doesn't carry the actual cash (remote/virtual). You use it exactly like money, but it mediates.

Problem Statement

Loading a high-res image is expensive; you want to defer it until first display (virtual proxy) and cache it. And some documents require authorization to open (protection proxy). You'll wrap the real object behind a proxy sharing its interface.

Internal Working



- **Subject interface** shared by proxy and real subject.
- **Proxy** holds/creates the real subject and adds control logic (lazy create, cache, check, remote call) before/after delegating.
- Clients depend on the interface and can't tell proxy from real.

Memory Representation

The proxy holds a (possibly-null until lazy-created) reference to the real subject.

Compiler Behavior / Runtime Behavior

Not special; proxy delegates at runtime after its control logic. Virtual proxy creates the real subject on first use.

Flutter Engine Behavior

Not applicable directly. (Flutter's image caching, lazy `ListView.builder` item creation, and network layers embody proxy-like deferral/caching.)

Dart VM Behavior

Not applicable.

Examples

```
abstract interface class Image {
  String display();
}

// Real subject: expensive to create
class HighResImage implements Image {
  final String path;
  HighResImage(this.path) {
    print('loading $path (expensive)'); // simulates heavy load
  }
  @override
  String display() => 'showing $path';
}

// Virtual + caching proxy: defer creation until first display
```

```
class LazyImageProxy implements Image {
  final String path;
  HighResImage? _real; // created on demand
  LazyImageProxy(this.path);
  @override
  String display() {
    _real ??= HighResImage(path); // lazy init (only once)
    return _real!.display();
  }
}

// Protection proxy: gate access
class ProtectedDoc implements Image {
  final Image _real;
  final bool Function() _isAuthorized;
  ProtectedDoc(this._real, this._isAuthorized);
  @override
  String display() =>
      _isAuthorized() ? _real.display() : 'access denied';
}

void main() {
  final img = LazyImageProxy('photo.png'); // NOT loaded yet
  print('created proxy, nothing loaded');
  print(img.display()); // loads now, then shows
  print(img.display()); // reuses loaded instance (cached)

  var loggedIn = false;
  final doc = ProtectedDoc(LazyImageProxy('secret.png'), () => loggedIn);
  print(doc.display()); // access denied
  loggedIn = true;
  print(doc.display()); // loads + shows
}
```

Diagrams

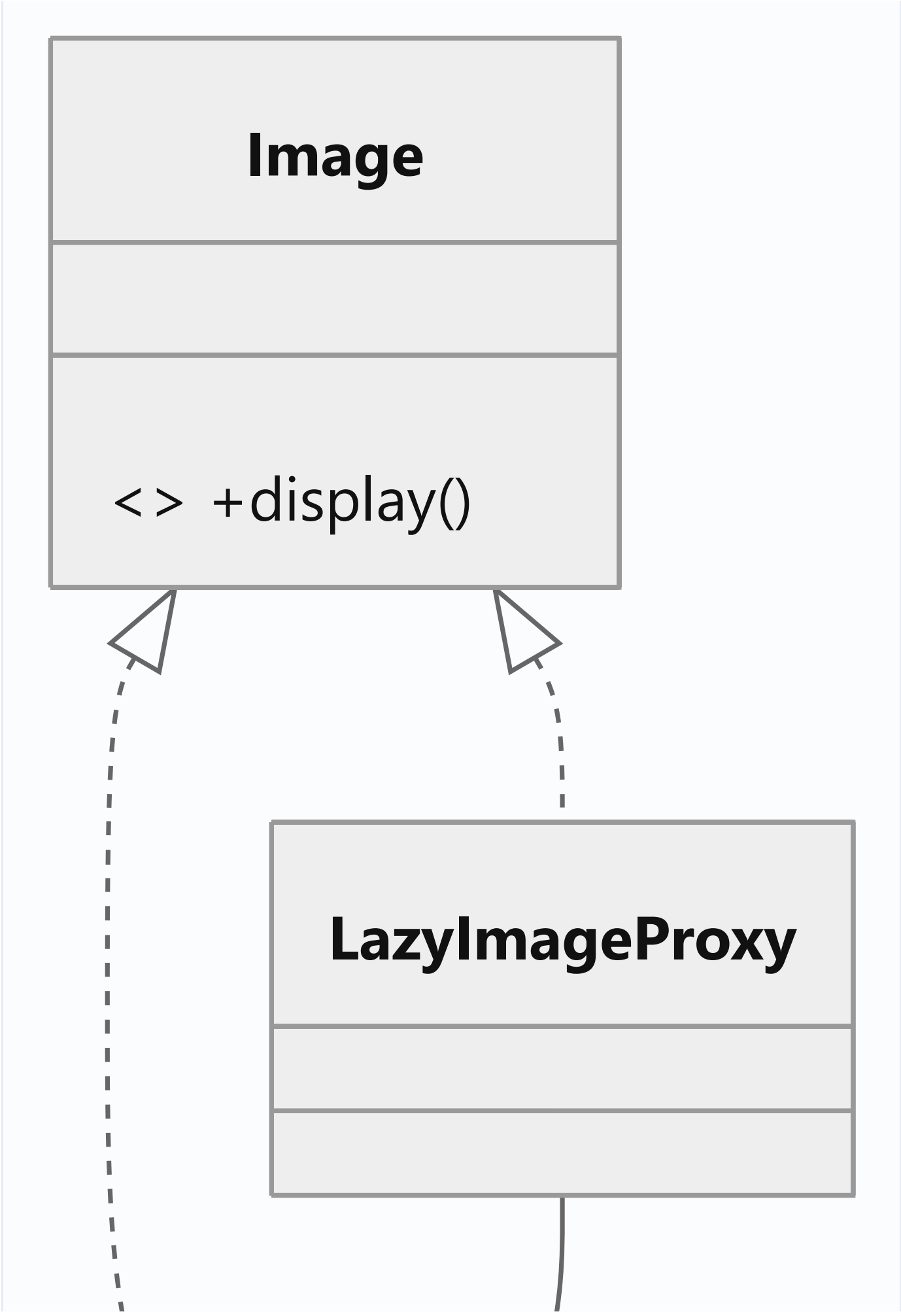
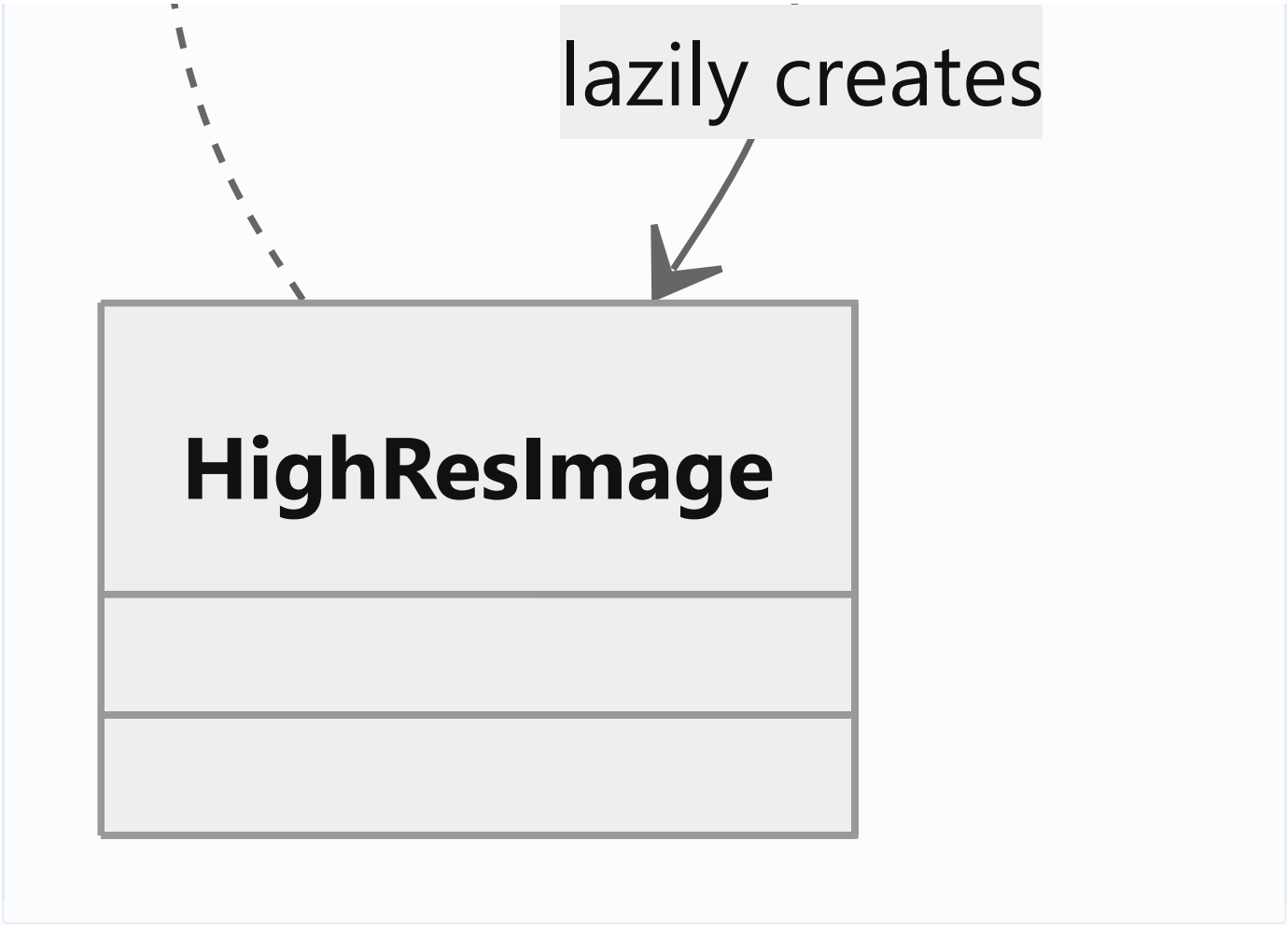


Image
<> +display()
LazyImageProxy



Common Mistakes

Mistake	Why	Fix
Proxy changing the interface	Not transparent	Share the subject interface
Business logic in the proxy	SRP violation	Keep proxy to access control only
Confusing Proxy with Decorator	Same structure, different intent	Proxy controls access; Decorator adds behavior
Not caching in a virtual proxy	Re-creates the expensive object	Create once, reuse

Best Practices

- Share the **subject interface**; keep the proxy transparent.
- Keep proxy logic to **access concerns** (lazy/cache/auth/remote), not business rules.
- Create the real subject once in virtual proxies (cache it).
- Combine with DIP: clients depend on the interface; proxy vs real is a wiring choice.

Performance

Virtual/caching proxies **improve** performance (defer/avoid work). Protection/remote proxies add a small check/round-trip.

Advantages / Disadvantages

- **+** Lazy loading, caching, access control, remote representation — transparently.
- **−** Extra layer; easy to conflate with Decorator; can hide latency (remote proxy).

Interview Questions

1. 🟢 **What is a Proxy?** — A stand-in sharing a real object's interface that controls access to it.
2. 🟢 **Name proxy types.** — Virtual (lazy), caching, protection (authorization), remote (network stand-in).
3. 🟡 **Proxy vs Decorator?** — Same wrapping structure; Proxy *controls access*, Decorator *adds behavior*. Intent differs.
4. 🟡 **How does a virtual proxy help performance?** — It defers expensive creation until first use and caches it thereafter.
5. 🟡 **What should a proxy avoid doing?** — Business logic or changing the interface; it should only mediate access.
6. 🔴 **Give Flutter analogues.** — Image cache, lazy `ListView.builder` item creation, network client wrappers.
7. 🔴 **How does a remote proxy hide complexity, and what's the risk?** — It represents a network resource behind a local interface; the risk is hidden latency/failure the caller doesn't expect.

Senior Engineer Tips

- Use a virtual/caching proxy to make "expensive if used, free if not" behavior transparent.
- Protection proxies centralize authorization checks at the resource boundary.
- Distinguish Proxy from Decorator in code review by *intent*: access control vs feature addition.

Architect Perspective

Proxies place cross-cutting access concerns (lazy init, caching, auth, remoting) at object boundaries without polluting callers or the real subject. They're foundational to performance (deferral/caching), security (protection), and distributed designs (remote stubs), often working alongside Repository and DI (Modules 15, 16, 37).

Summary

- Proxy is a same-interface stand-in that controls access (virtual/caching/protection/remote).
- Keep it transparent and access-focused; create the real subject lazily and cache it.
- Structurally like Decorator; the intent (access vs behavior) distinguishes them.

Revision Notes

- Proxy = surrogate sharing interface, controls access.
- Types: virtual (lazy), caching, protection (auth), remote.
- Proxy controls access; Decorator adds behavior (same structure).
- Flutter: image cache, lazy list items, network wrappers.

Practice Questions

1. Why is a virtual proxy a performance optimization?
2. How do you tell Proxy from Decorator in code?
3. What risk does a remote proxy introduce?

Coding Questions

1. Build a caching proxy over an expensive `PrimeChecker`.
2. Implement a protection proxy gating a `BankAccount` behind an auth check.
3. Write a virtual proxy that lazily loads a config file on first access.

Mini Project

Image loading proxy (pure Dart): Implement an `Image` interface with an expensive `HighResImage`, a `LazyImageProxy` (defer + cache), and a `ProtectedImage` proxy (auth gate). Compose them and test that loading is deferred, cached, and gated. Acceptance: interface transparent; single expensive load; access control enforced; `dart analyze` clean.

Strategy Pattern

> Strategy defines a family of interchangeable algorithms behind a common interface, letting you swap behavior at runtime without changing the code that uses it.

Introduction

Strategy encapsulates each algorithm (sorting, pricing, validation, routing) as an object implementing a shared interface, so the context can switch algorithms at runtime. It's the canonical way to satisfy the Open/Closed Principle for *behavior*.

Why this concept exists

Hard-coding an algorithm (or `if/else` selecting one) couples the caller to every variant and forces edits to add new ones. Strategy externalizes the algorithm so variants are added/swapped independently — and injected for testing.

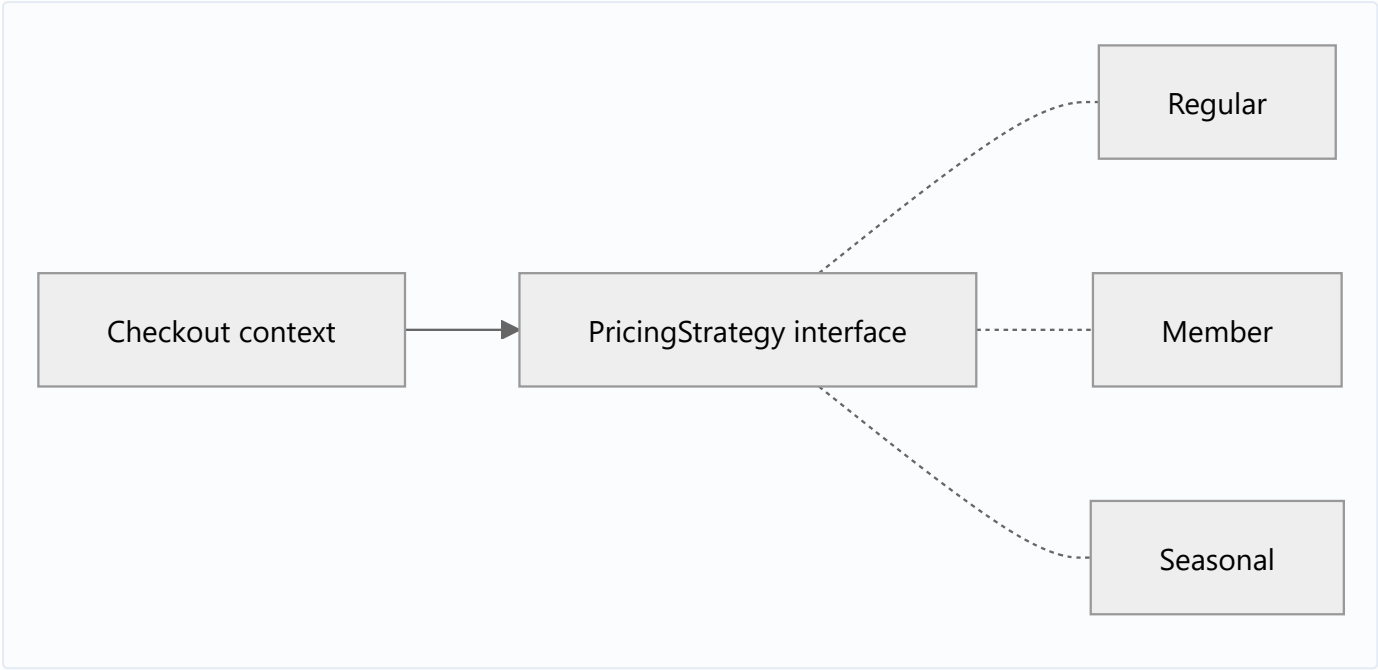
Real-world analogy

Navigation modes in a maps app: "drive," "walk," "cycle," "transit" are interchangeable route-finding strategies. You pick one; the app computes the route the same way regardless — and adding "scooter" doesn't change the app's routing call.

Problem Statement

A checkout must apply different pricing rules (regular, member, seasonal) chosen at runtime. `if (type == ...)` scatters and grows. You'll inject a `PricingStrategy` the context uses.

Internal Working



- **Strategy interface** declares the algorithm (`double price(double base)`).
- **Concrete strategies** implement variants.
- **Context** holds a strategy (injected/swappable) and delegates to it.
- In Dart, a strategy can be a **function** (`typedef`) instead of a class — often simpler.

Memory Representation

The context holds one strategy reference; strategies are usually stateless (can be `const` /singletons).

Compiler Behavior / Runtime Behavior

Not special; the context calls the strategy polymorphically; swapping changes behavior at runtime.

Flutter Engine Behavior

Not applicable. (Flutter uses Strategy widely: `ScrollPhysics` , `PageTransitionsBuilder` , comparator functions in `sort` .)

Dart VM Behavior

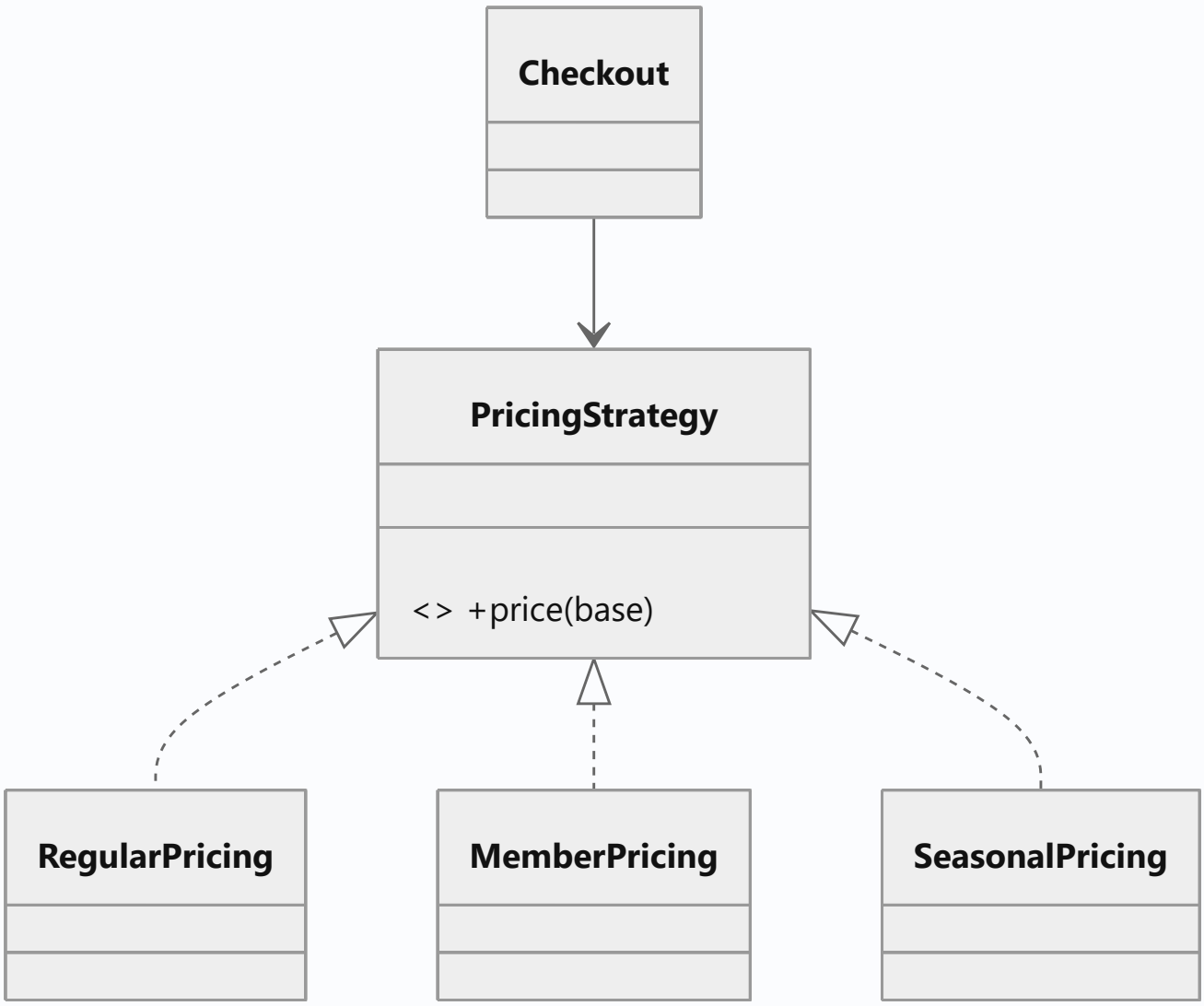
Not applicable.

Examples

```
// Class-based strategy
abstract interface class PricingStrategy {
  double price(double base);
}
class RegularPricing implements PricingStrategy {
  @override
  double price(double base) => base;
}
class MemberPricing implements PricingStrategy {
  @override
  double price(double base) => base * 0.9;
}
class SeasonalPricing implements PricingStrategy {
  final double off;
  const SeasonalPricing(this.off);
  @override
  double price(double base) => base * (1 - off);
}

class Checkout {
  PricingStrategy strategy; // swappable at runtime
  Checkout(this.strategy);
  double total(double base) => strategy.price(base);
}

// Function-based strategy (idiomatic Dart)
typedef Pricing = double Function(double base);
double memberPricing(double base) => base * 0.9;

void main() {
  final cart = Checkout(RegularPricing());
  print(cart.total(100)); // 100.0
  cart.strategy = MemberPricing(); // swap behavior at runtime
  print(cart.total(100)); // 90.0
  cart.strategy = const SeasonalPricing(0.25);
  print(cart.total(100)); // 75.0

  // function strategy:
```

```
  Pricing p = memberPricing;
  print(p(200)); // 180.0
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`if/switch` selecting behavior inline	OCP violation, coupling	Extract strategies + inject
Stateful strategies with hidden state	Surprising behavior	Keep strategies stateless/immutable
Class strategy where a function suffices	Boilerplate	Use a `typedef` /closure
Context knowing concrete strategies	Coupling	Depend on the strategy interface

Best Practices

- Depend on the **strategy interface**; inject the concrete choice.
- Prefer **function-type strategies** (`typedef` + closures) for simple algorithms.
- Keep strategies **stateless/immutable** (make them `const`).
- Combine with Factory/DI to select strategies by config.

Performance

Negligible (one indirection); stateless strategies can be shared singletons.

Advantages / Disadvantages

- **+** Swap algorithms at runtime, OCP-compliant, testable (inject fakes), removes conditionals.
- **−** More types (unless using functions); context must expose a way to set the strategy.

Interview Questions

1. 🟢 **What is Strategy?** — A family of interchangeable algorithms behind one interface, selectable/swappable at runtime.
2. 🟢 **How does Strategy support OCP?** — New algorithms are new implementations; the context using the interface never changes.
3. 🟡 **Class vs function strategy in Dart?** — Functions (`typedef` /closures) are idiomatic for simple, stateless algorithms; classes suit stateful/config-carrying strategies.
4. 🟡 **Strategy vs State pattern?** — Strategy swaps an algorithm chosen by the client; State changes behavior based on the object's *internal state*, often transitioning itself.
5. 🟡 **Where does Flutter use Strategy?** — `ScrollPhysics` , sort comparators, transition builders.
6. 🔴 **How do Strategy and DIP relate?** — The context depends on the strategy abstraction and receives the concrete strategy via injection.
7. 🔴 **Why keep strategies stateless?** — So they're reusable, shareable (`const` /singleton), and free of surprising cross-call state.

Senior Engineer Tips

- In Dart, reach for a function-typed strategy first; escalate to a class only when the strategy needs state/config or multiple methods.
- Pair Strategy with a registry/factory to pick strategies from config or feature flags.
- Strategy is the refactor target when you see a growing behavior-selecting `switch` (OCP file, Module 04).

Architect Perspective

Strategy is the workhorse for pluggable behavior — pricing, ranking, validation, sync policies — enabling A/B tests, feature flags, and per-tenant customization without editing core flows. It's OCP + DIP in miniature and appears throughout well-factored codebases.

Summary

- Strategy = interchangeable algorithms behind one interface, swapped at runtime.
- Use functions for simple cases, classes for stateful ones; keep them stateless; inject them.
- Distinct from State (internal-state-driven, self-transitioning).

Revision Notes

- Strategy = swappable algorithm via interface (or `typedef`/closure).
- Inject the strategy; keep stateless (`const`); OCP + DIP.
- Strategy (client-chosen algorithm) vs State (internal-state behavior).
- Flutter: ScrollPhysics, sort comparators.

Practice Questions

1. When is a function strategy better than a class?
2. How does Strategy differ from State?
3. Why keep strategies stateless?

Coding Questions

1. Implement `SortStrategy` (asc/desc/byLength) injected into a `Sorter`.
2. Build a function-based `Validator` strategy set (email/phone/nonEmpty).
3. Add a new pricing strategy and prove `Checkout` is unchanged.

Mini Project

Pricing engine (pure Dart): Implement `PricingStrategy` (regular/member/seasonal/coupon) both as classes and function strategies, a `Checkout` context that swaps them at runtime, and a factory selecting by config. Test each and the swap. Acceptance: no behavior-selecting `switch` in `Checkout`; strategies stateless; new strategy added without editing context; `dart analyze` clean.

Observer Pattern

> Observer lets an object (subject) notify a list of dependents (observers) automatically when its state changes — the foundation of reactive UI.

Introduction

Observer establishes a one-to-many dependency: when the subject changes, all registered observers are notified. It's the mechanism behind `ChangeNotifier` / `ValueNotifier`, streams, and every reactive state-management solution in Flutter.

Why this concept exists

UIs must react to data changes without the data knowing about the UI. Observer decouples the source of change (subject) from the reactors (observers): the subject just announces "I changed," and interested parties respond — enabling loose coupling and multiple independent listeners.

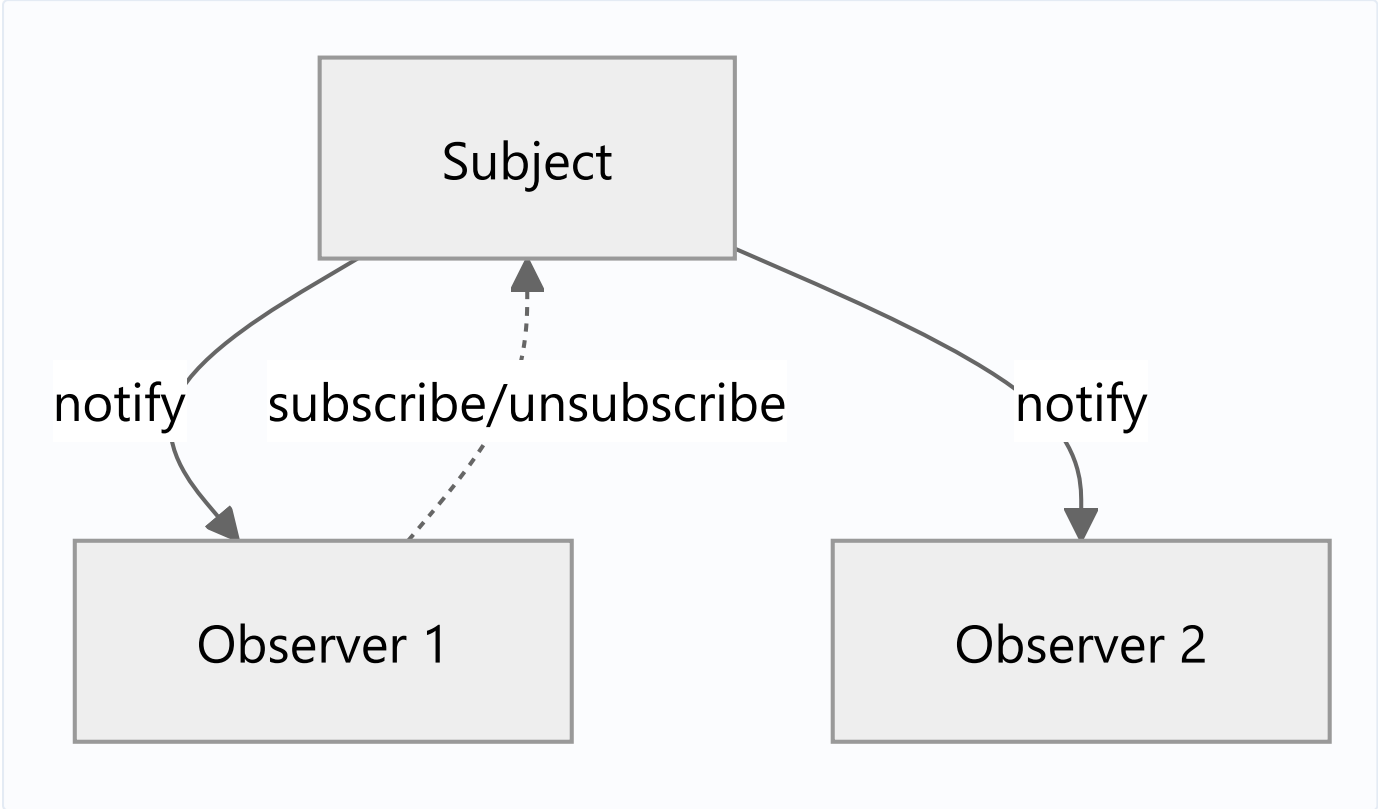
Real-world analogy

A **newsletter**: subscribers (observers) register with the publisher (subject). When a new issue is published, everyone subscribed gets it automatically. The publisher doesn't know or care who subscribers are; subscribers can join/leave anytime.

Problem Statement

A `Stock` price changes and multiple widgets/loggers must update. Hard-wiring the stock to each consumer couples them. You'll implement subscribe/notify so observers react without the subject knowing their concrete types.

Internal Working



- **Subject** maintains a list of observers and `subscribe` / `unsubscribe` / `notify` methods.
- **Observers** implement an `update(...)` callback (or are just closures).
- On state change, the subject calls each observer.
- Dart-native forms: `Stream` / `StreamController` (broadcast), `ChangeNotifier` / `Listenable`, `ValueNotifier<T>`.

Memory Representation

The subject holds references to observers → **observers won't be GC'd while subscribed**; forgetting to unsubscribe leaks them (02 · memory_and_gc).

Compiler Behavior

Not applicable.

Runtime Behavior

Notification iterates observers synchronously (or via stream events). Modifying the observer list during notification needs care (iterate a copy).

Flutter Engine Behavior

Not applicable, but core to Flutter: `setState`, `ChangeNotifier` + `ListenableBuilder`, `ValueListenableBuilder`, and `StreamBuilder` all implement/consume Observer (Module 11).

Dart VM Behavior

Not applicable.

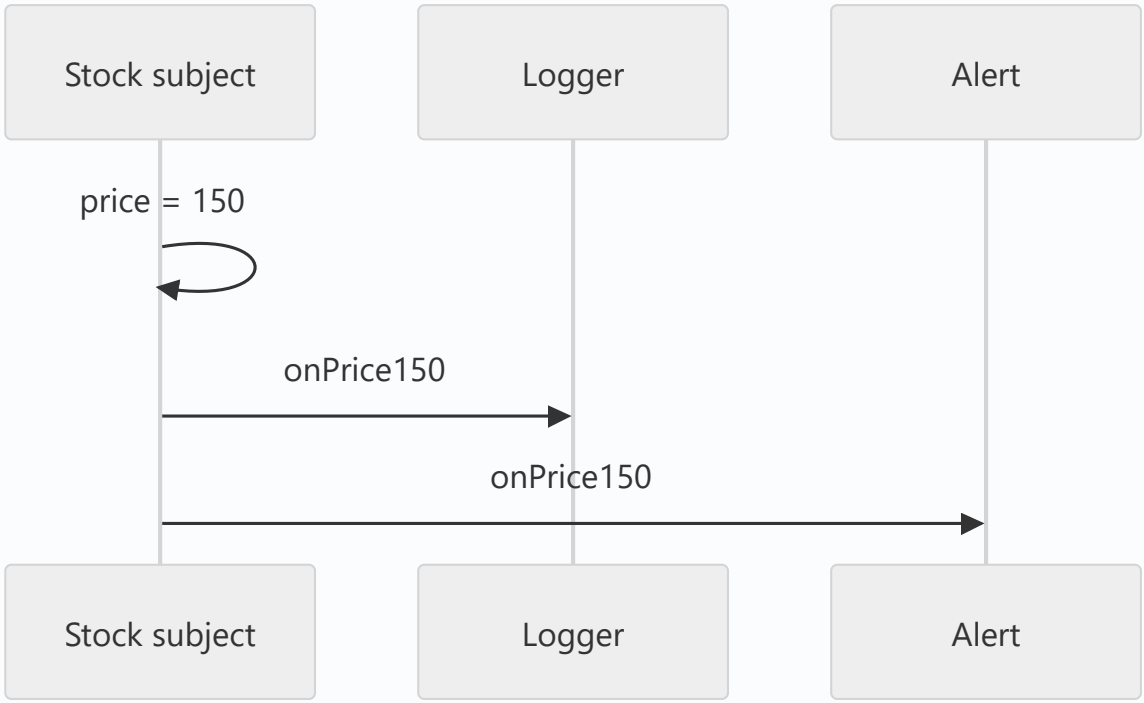
Examples

```
// Hand-rolled Observer
abstract interface class PriceObserver {
  void onPrice(double price);
}

class Stock {
  final _observers = <PriceObserver>[];
  double _price = 0;

  void subscribe(PriceObserver o) => _observers.add(o);
  void unsubscribe(PriceObserver o) => _observers.remove(o); // avoid leaks!

  set price(double p) {
    _price = p;
    for (final o in List.of(_observers)) {
      o.onPrice(p); // notify (copy to survive concurrent unsubscribe)
    }
  }
}

class Logger implements PriceObserver {
  @override
  void onPrice(double price) => print('log: $price');
}
class Alert implements PriceObserver {
  @override
  void onPrice(double price) {
    if (price > 100) print('ALERT: $price');
  }
```

```
}

void main() {
  final stock = Stock();
  final logger = Logger();
  stock.subscribe(logger);
  stock.subscribe(Alert());

  stock.price = 90;  // log: 90.0
  stock.price = 150; // log: 150.0  +  ALERT: 150.0

  stock.unsubscribe(logger); // stop receiving (and allow GC)
  stock.price = 200; // ALERT: 200.0 only
}
```

```
// Idiomatic Dart: broadcast Stream is Observer built-in
// final controller = StreamController<double>.broadcast();
// controller.stream.listen(print); // observer; remember to cancel the subscription
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Never unsubscribing	Memory leak; callbacks after dispose	Unsubscribe/cancel in `dispose`
Mutating observer list during notify	ConcurrentModification	Iterate a copy
Heavy work in observer callbacks	Blocks/janks	Keep handlers light; offload
Subject knowing concrete observer types	Coupling	Depend on the observer interface/callback

Best Practices

- Always provide (and use) **unsubscribe**; cancel in `dispose` to prevent leaks.
- Prefer Dart-native mechanisms (`Stream.broadcast`, `ValueNotifier`, `ChangeNotifier`) over hand-rolled lists.
- Keep observer callbacks fast; iterate a copy when notifying.
- Consider push (send data) vs pull (observer queries subject) notification styles.

Performance

Notification is O(observers); keep callbacks light. Broadcast streams are efficient but each listener still runs on the loop.

Advantages / Disadvantages

- + Loose coupling, multiple independent listeners, reactive UI foundation.
- − Leak-prone (must unsubscribe), notification order/reentrancy pitfalls, debugging "who updated?" can be hard.

Interview Questions

1. 🟢 **What is Observer?** — A one-to-many dependency where a subject notifies registered observers automatically on state change.
2. 🟢 **Dart/Flutter built-ins that implement Observer?** — `Stream` / `StreamController.broadcast`, `ChangeNotifier`, `ValueNotifier`, and the builders that listen to them.
3. 🟡 **Why is Observer leak-prone?** — The subject holds observer references; unsubscribed observers (and their captures like `State` / `context`) stay reachable — leak. Always unsubscribe/cancel.
4. 🟡 **Push vs pull notification?** — Push sends the changed data to observers; pull just signals change and observers query the subject.

5. 🟡 **Why iterate a copy of observers when notifying?** — An observer may unsubscribe during notification, mutating the list mid-iteration.
6. 🔴 **Observer vs Pub/Sub?** — Observer usually has direct subject↔observer references; Pub/Sub adds a broker/event bus decoupling publishers and subscribers entirely.
7. 🔴 **How does Observer underpin state management?** — Reactive solutions expose a listenable/stream subject; widgets observe and rebuild on change.

Senior Engineer Tips

- Prefer `Stream` / `ValueNotifier` / `ChangeNotifier` over hand-rolled observer lists — they handle the mechanics and integrate with Flutter builders.
- Treat every subscription as owned; cancel in `dispose` (a top Flutter leak source — 02 · memory_and_gc).
- Beware reentrancy: an observer that mutates the subject during notification can cause surprising cascades.

Architect Perspective

Observer is the backbone of reactive architecture: it decouples state producers from consumers, enabling the entire spectrum of Flutter state management (Provider/Riverpod/BLoC) (Module 11). Discipline around subscription lifecycle is a system-wide reliability concern.

Summary

- Observer: subject notifies many observers on change; decouples producer from consumers.
- Use Dart-native streams/notifiers; always unsubscribe (leak risk); iterate a copy on notify.
- Foundation of reactive UI and state management.

Revision Notes

- Observer = 1-to-many notify on change; subscribe/unsubscribe/notify.
- Dart built-ins: `Stream.broadcast`, `ChangeNotifier`, `ValueNotifier`.
- Must unsubscribe/cancel (leak!); iterate copy on notify.
- Underpins state management; Pub/Sub adds a broker.

Practice Questions

1. Why does forgetting to unsubscribe leak memory?
2. Push vs pull notification — tradeoffs?
3. Why notify over a copy of the observer list?

Coding Questions

1. Implement a generic `Observable<T>` with subscribe/unsubscribe/notify.
2. Reimplement the stock example with a broadcast `Stream` and cancellable subscriptions.
3. Build a `ValueNotifier`-like class from scratch with listeners.

Mini Project

Reactive stock ticker (pure Dart): Implement `Observable<double>` with subscribe/unsubscribe, plus `Logger` and `Alert` observers; simulate price changes; prove unsubscribed observers stop receiving and are collectible. Then reimplement using a broadcast `Stream`. Acceptance: no leaks (all cancelled); notify-over-copy; both implementations tested; `dart analyze` clean.

Command Pattern

> Command turns a request into a standalone object — capturing the action, its parameters, and (often) how to undo it — enabling queues, logs, undo/redo, and decoupling of invoker from receiver.

Introduction

Command encapsulates "do X" as an object with an `execute()` (and often `undo()`). The invoker triggers commands without knowing what they do; the command knows its receiver and parameters. This enables undo/redo, macro recording, queuing, and transactional operations.

Why this concept exists

Sometimes an action must be **reified** — stored, queued, logged, undone, or replayed. A plain method call can't be any of those. Wrapping it as a Command object makes actions first-class: schedule them, keep a history, reverse them.

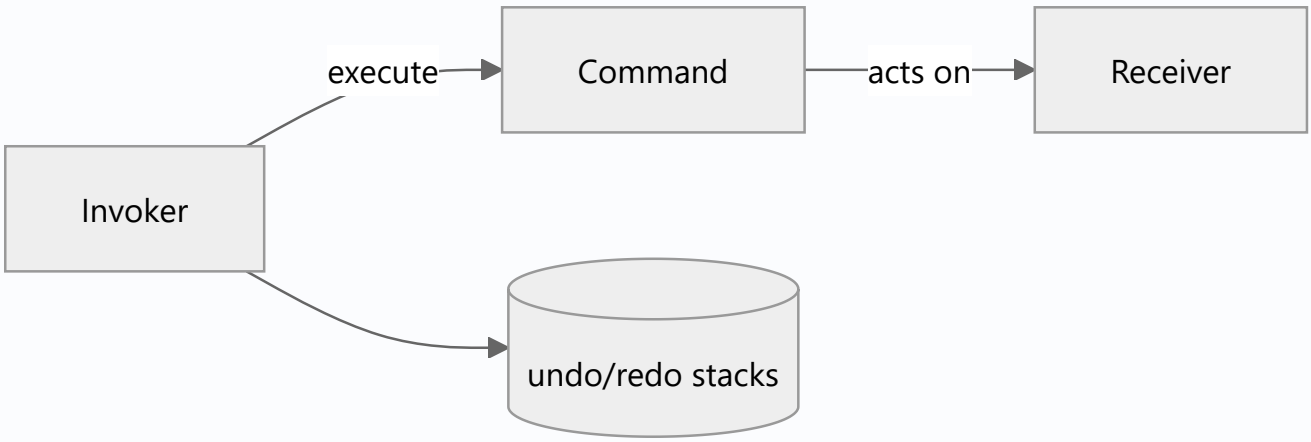
Real-world analogy

A **restaurant order ticket**: the waiter (invoker) writes an order (command) and hands it to the kitchen (receiver). The ticket can be queued, re-fired, or voided (undo). The waiter doesn't cook; the ticket carries all the info.

Problem Statement

A text editor needs undo/redo for actions like insert and delete. You'll model each action as a Command with `execute` / `undo`, and keep an undo/redo stack in an invoker.

Internal Working



- **Command interface:** `execute()` and optionally `undo()`.
- **Concrete command:** holds the receiver + params; implements do/undo.
- **Invoker:** runs commands and maintains history (stacks) for undo/redo.
- **Receiver:** the object the command operates on.
- Dart-friendly: a command can be a pair of closures (`do` , `undo`).

Memory Representation

Command objects + a history stack retain state needed to undo; unbounded history grows memory — cap it.

Compiler Behavior / Runtime Behavior

Not special; `execute` / `undo` run at invocation; history push/pop manages reversibility.

Flutter Engine Behavior

Not applicable. (Flutter `Intent` / `Action` / `Shortcuts` is a Command-like system; undo/redo stacks in editors use Command.)

Dart VM Behavior

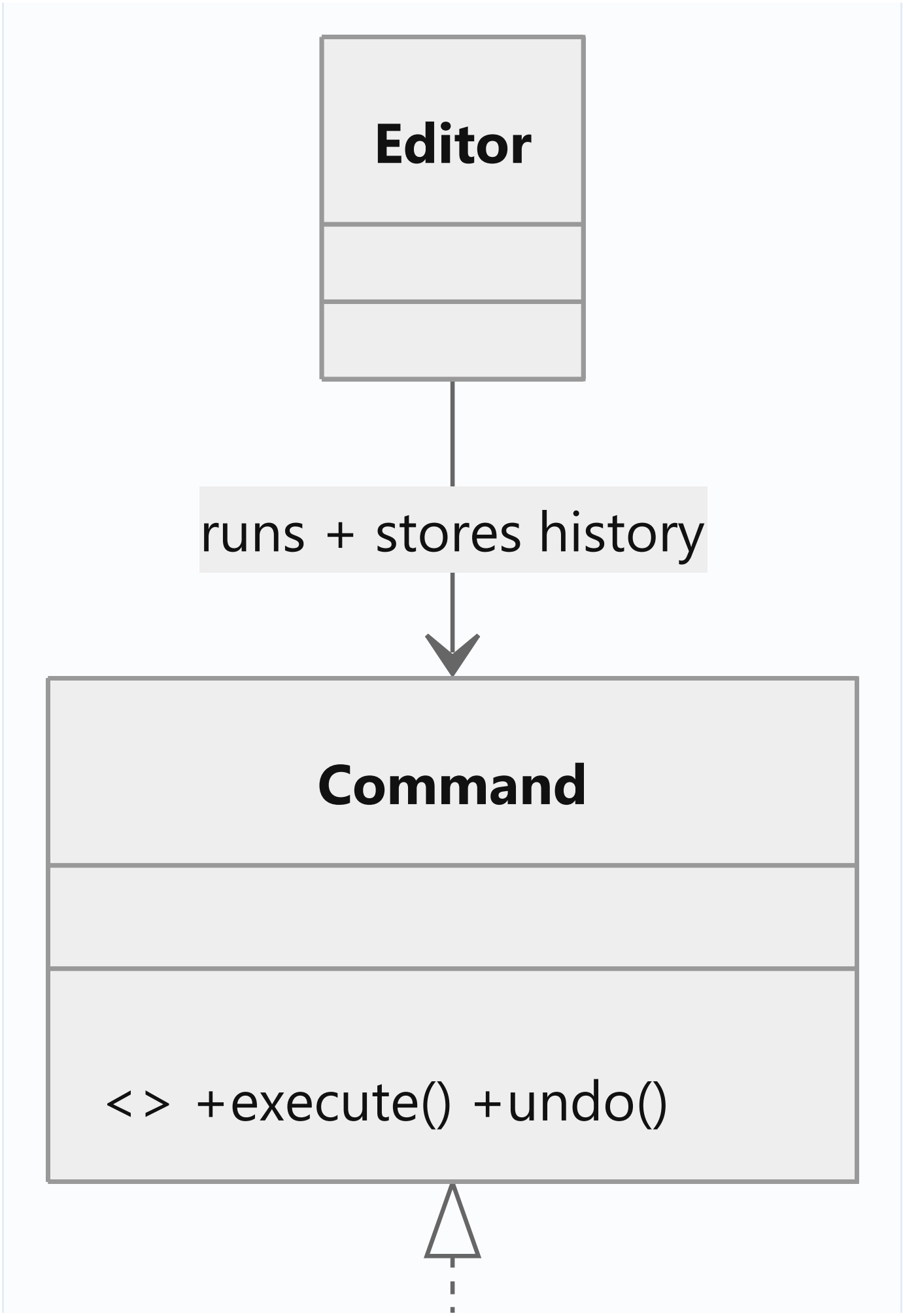
Not applicable.

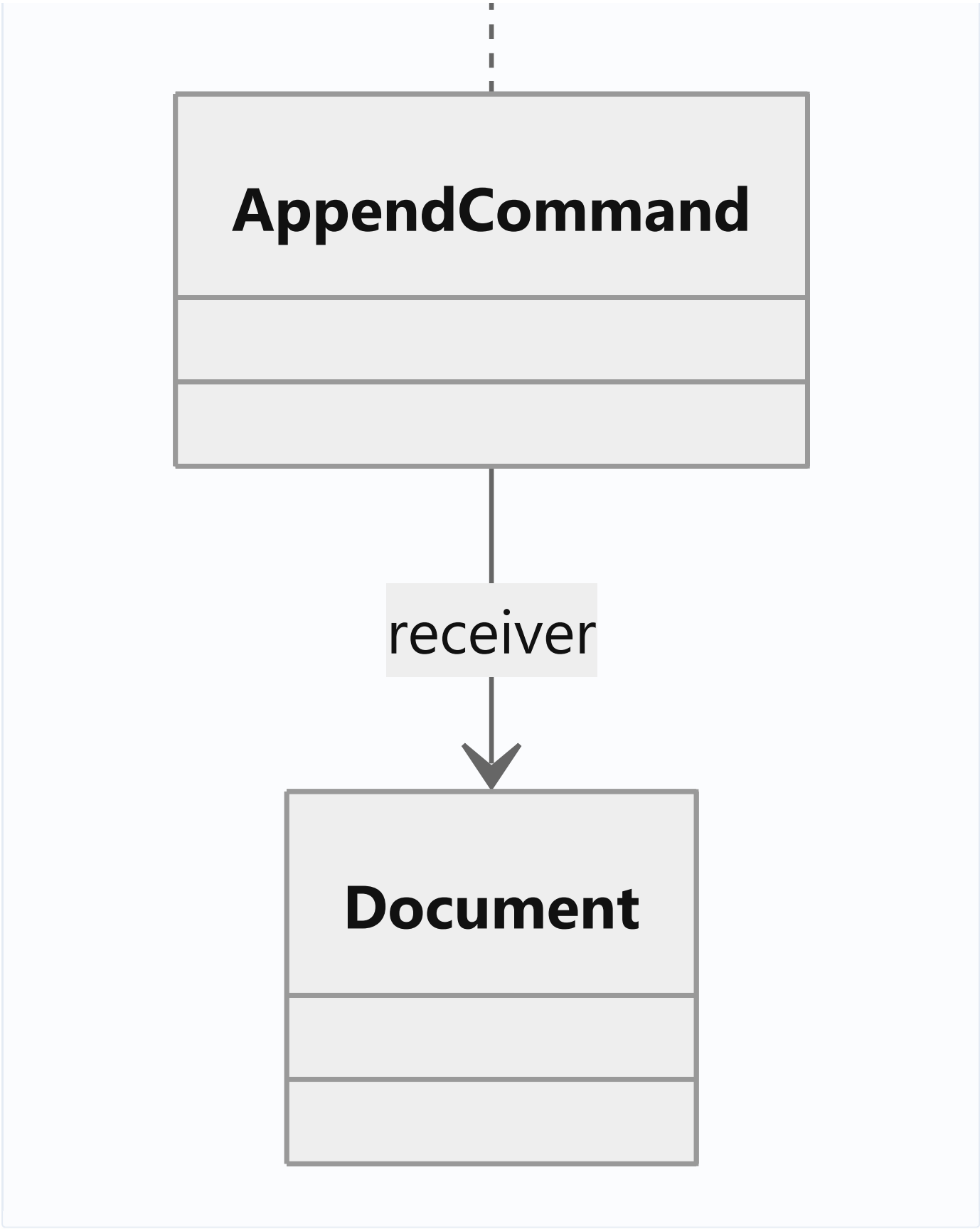
Examples

```
abstract interface class Command {
  void execute();
  void undo();
}

class Document {
  final StringBuffer _buf = StringBuffer();
  String get text => _buf.toString();
  void append(String s) => _buf.write(s);
  void removeLast(int n) {
    final t = text;
    _buf
      ..clear()
      ..write(t.substring(0, t.length - n));
  }
}

class AppendCommand implements Command {
  final Document doc;
  final String textToAdd;
  AppendCommand(this.doc, this.textToAdd);
  @override
  void execute() => doc.append(textToAdd);
  @override
  void undo() => doc.removeLast(textToAdd.length); // reverse
}

class Editor {
  final _undo = <Command>[];
  final _redo = <Command>[];

  void run(Command c) {
    c.execute();
    _undo.add(c);
    _redo.clear();
  }
  void undo() {
    if (_undo.isEmpty) return;
```

```
    final c = _undo.removeLast()..undo();
    _redo.add(c);
  }
  void redo() {
    if (_redo.isEmpty) return;
    final c = _redo.removeLast()..execute();
    _undo.add(c);
  }
}

void main() {
  final doc = Document();
  final editor = Editor();
  editor.run(AppendCommand(doc, 'Hello '));
  editor.run(AppendCommand(doc, 'World'));
  print(doc.text); // Hello World
  editor.undo();
  print(doc.text); // Hello
  editor.redo();
  print(doc.text); // Hello World
}
```

Diagrams


Editor
runs + stores history
Command
<> +execute() +undo()


AppendCommand
receiver
Document

Common Mistakes

Mistake	Why	Fix
Command without enough state to undo	Undo impossible/incorrect	Capture what's needed to reverse
Unbounded history	Memory growth	Cap history size
Business logic in the invoker	Coupling	Keep invoker generic; logic in commands/receivers
Non-idempotent redo	Corrupts state	Ensure execute/undo are exact inverses

Best Practices

- Make `execute` / `undo` **exact inverses**; capture the minimal state to reverse.
- Keep the **invoker generic** (runs any command + manages history).
- Cap history to bound memory.
- Consider closure-pair commands for lightweight cases.

Performance

History retains state; cap it. Command dispatch itself is cheap.

Advantages / Disadvantages

- + Undo/redo, queuing, logging, macros, decoupled invoker/receiver, transactional actions.
- − Many command classes; undo state management; memory from history.

Interview Questions

1. 🟢 **What does Command encapsulate?** — A request as an object: the action, its parameters, and (often) how to undo it.
2. 🟢 **What capabilities does Command enable?** — Undo/redo, queuing, logging/replay, macros, and decoupling invoker from receiver.
3. 🟡 **What's needed to support undo?** — The command must capture enough state to reverse its effect (`undo()` is the inverse of `execute()`).
4. 🟡 **Who are the invoker and receiver?** — Invoker triggers commands and manages history; receiver is the object the command acts on.
5. 🟡 **How is Command done idiomatically in Dart?** — Often as a pair of closures (`execute` , `undo`) rather than a full class per action.
6. 🔴 **How do undo/redo stacks interact?** — Executing pushes to undo (clears redo); undo pops to redo; redo pops back to undo.

7. 🔴 **Where does Flutter use Command-like structures?** — The `Actions` / `Intent` / `Shortcuts` system maps intents to actions (commands).

Senior Engineer Tips

- For editors/drawing/forms, model each user action as a command from day one — retrofitting undo later is painful.
- Prefer capturing inverse *data* (e.g., removed text) over recomputing it; recomputation is error-prone.
- Bound and possibly serialize history for crash recovery in production editors.

Architect Perspective

Command reifies actions, enabling undo/redo, offline action queues (record commands, replay on reconnect — Module 19), audit logs, and macro systems. It cleanly separates "what to do" from "when/whether to do it," which is powerful for resilient, user-forgiving apps.

Summary

- Command wraps an action (+undo) as an object; invoker runs and records it.
- Enables undo/redo, queuing, logging, macros; keep execute/undo exact inverses; cap history.
- Use closure pairs in Dart for lightweight commands.

Revision Notes

- Command = request-as-object (`execute` / `undo`); invoker + receiver.
- Enables undo/redo, queue, log, macros; capture inverse state; cap history.
- Dart: closure-pair commands. Flutter: Actions/Intents.

Practice Questions

1. What state must an `AppendCommand` capture to undo?
2. How do the undo and redo stacks stay consistent?
3. When is a closure-pair command better than a class?

Coding Questions

1. Add a `DeleteCommand` (with undo) to the editor.
2. Implement a `MacroCommand` that groups several commands into one undoable unit.
3. Build a queued command runner that records and replays commands.

Mini Project

Undoable text editor (pure Dart): Implement `Command` with `Append` / `Delete` / `Replace`, an `Editor` invoker with bounded undo/redo, and a macro command. Test undo/redo correctness across sequences. Acceptance: execute/undo are exact inverses; history bounded; invoker generic; `dart analyze` clean.

State Pattern

> State lets an object change its behavior when its internal state changes — as if it changed class — replacing sprawling state-checking conditionals with polymorphic state objects.

Introduction

State encapsulates each state of an object as its own class implementing a shared interface; the context delegates behavior to its current state object, which can also trigger transitions. It turns a tangled state machine of `if (state == ...)` into clean, self-contained states.

Why this concept exists

Objects with modal behavior (a document that's Draft/Moderated/Published, a media player that's Stopped/Playing/Paused) accumulate conditionals in every method (`if (state == playing) ... else if ...`). State replaces that with polymorphism: each state class knows its own behavior and valid transitions.

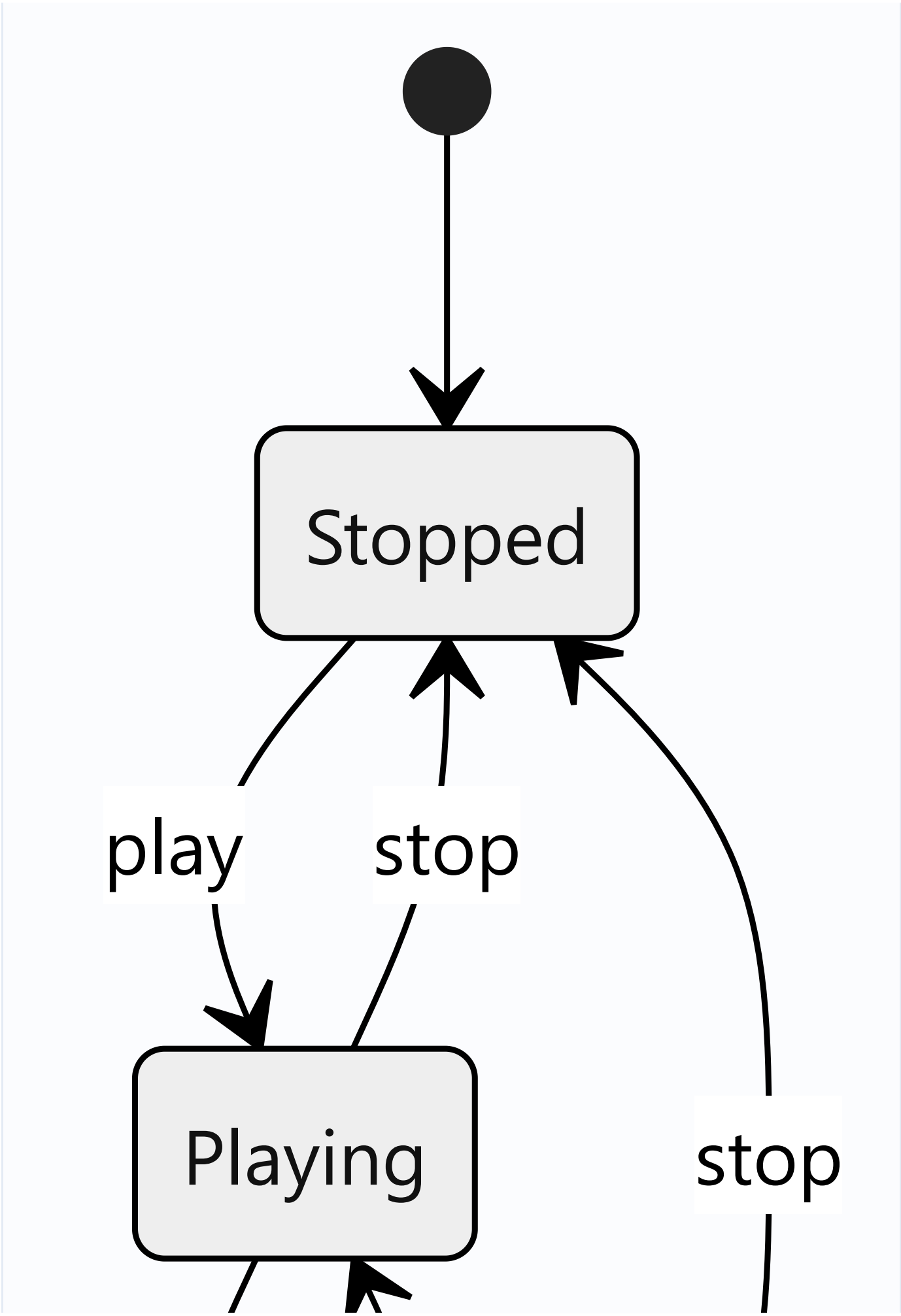
Real-world analogy

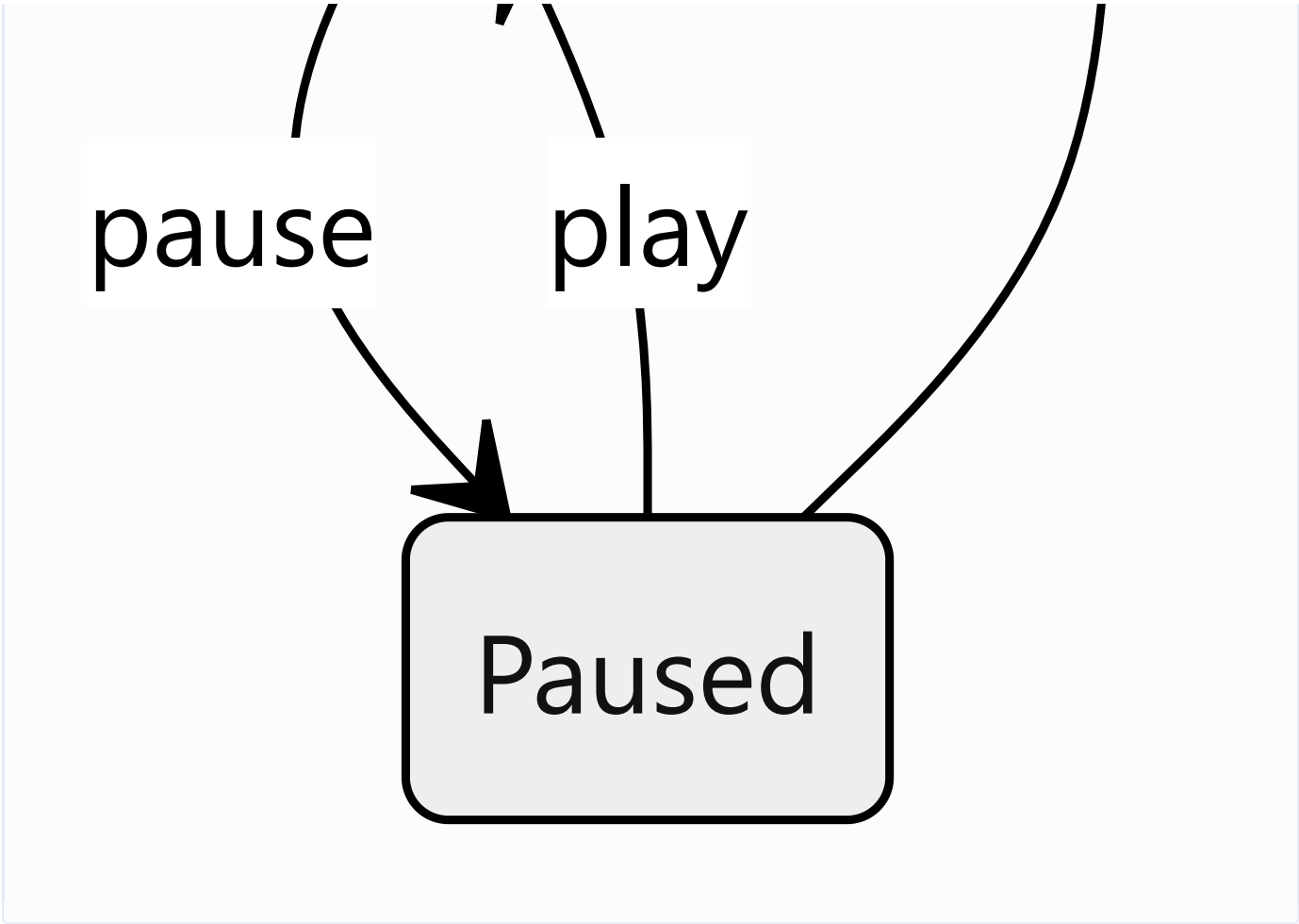
A **traffic light**: in "Green" it behaves one way and transitions to "Yellow"; "Yellow" → "Red"; "Red" → "Green." Each color (state) defines what happens next. The intersection (context) just delegates to the current light.

Problem Statement

A media player behaves differently for `play` / `pause` / `stop` depending on whether it's Stopped, Playing, or Paused, and only certain transitions are valid. You'll model each state as a class handling those actions and transitions.

Internal Working


Stopped
play
stop
Playing
stop



- **State interface** declares the actions (`play` , `pause` , `stop`).
- **Concrete states** implement behavior + set the context's next state.
- **Context** holds the current state and delegates; states call back to switch it.
- Dart 3 alternative: a **sealed class** of states + exhaustive `switch` in the context (data-oriented state machine).

Memory Representation

The context holds one current-state reference; states are usually stateless (shareable/ `const`).

Compiler Behavior / Runtime Behavior

Delegation + transition at runtime. With sealed states, the compiler enforces exhaustive handling.

Flutter Engine Behavior

Not applicable, but modeling UI/BLoC state as sealed states + exhaustive `switch` is idiomatic Flutter (Module 11).

Dart VM Behavior

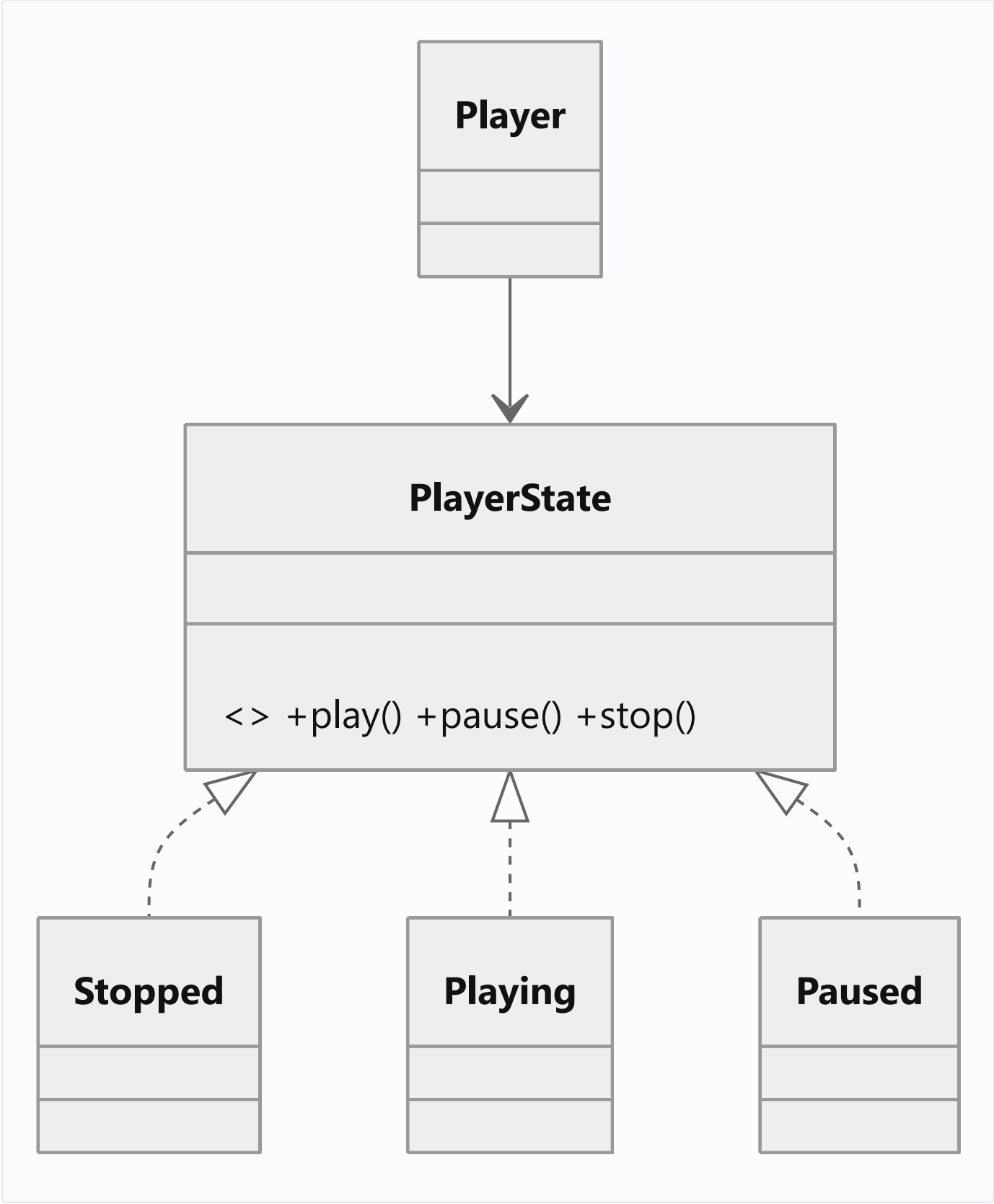
Not applicable.

Examples

```
abstract interface class PlayerState {
  void play(Player p);
  void pause(Player p);
  void stop(Player p);
  String get name;
}

class Stopped implements PlayerState {
  const Stopped();
  @override
  String get name => 'Stopped';
  @override
  void play(Player p) { print('start'); p.state = const Playing(); }
  @override
  void pause(Player p) => print('cannot pause when stopped');
  @override
  void stop(Player p) => print('already stopped');
}
class Playing implements PlayerState {
  const Playing();
  @override
  String get name => 'Playing';
  @override
  void play(Player p) => print('already playing');
  @override
  void pause(Player p) { print('pause'); p.state = const Paused(); }
  @override
  void stop(Player p) { print('stop'); p.state = const Stopped(); }
}
class Paused implements PlayerState {
  const Paused();
  @override
  String get name => 'Paused';
  @override
```

```
  void play(Player p) { print('resume'); p.state = const Playing(); }
  @override
  void pause(Player p) => print('already paused');
  @override
  void stop(Player p) { print('stop'); p.state = const Stopped(); }
}

class Player {
  PlayerState state = const Stopped();
  void play() => state.play(this);
  void pause() => state.pause(this);
  void stop() => state.stop(this);
}

void main() {
  final p = Player();
  p.play();  // start   (Stopped -> Playing)
  p.pause(); // pause   (Playing -> Paused)
  p.play();  // resume  (Paused -> Playing)
  p.stop();  // stop    (Playing -> Stopped)
  p.pause(); // cannot pause when stopped
}
```

Diagrams


Player
PlayerState
<> +play() +pause() +stop()
Stopped
Playing
Paused

Common Mistakes

Mistake	Why	Fix
`if (state == ...)` in every method	The problem State solves	Delegate to state objects
States mutating shared context data unsafely	Hidden coupling	Keep transitions explicit and localized
Illegal transitions unhandled	Invalid states reachable	Each state defines/blocks its transitions
Stateful state objects	Surprising reuse bugs	Keep states stateless (`const`)

Best Practices

- One class per state; each defines behavior + valid transitions.
- Keep states **stateless/** `const` (shareable); put mutable data in the context.
- Handle/deny invalid transitions explicitly.
- Consider **sealed classes + exhaustive** `switch` for a data-oriented alternative (great with Dart 3 and BLoC).

Performance

Negligible; stateless states are shared singletons.

Advantages / Disadvantages

- **+** Removes conditional sprawl, localizes per-state behavior + transitions, easy to add states.
- **−** More classes; transition logic spread across states can be harder to see at a glance (a state diagram helps).

Interview Questions

1. 🟢 **What is the State pattern?** — Encapsulating each state as an object so an object's behavior changes with its internal state, replacing conditionals with polymorphism.
2. 🟢 **State vs Strategy?** — Same structure; Strategy's algorithm is chosen by the client and independent of state; State's behavior depends on internal state and states often trigger transitions.
3. 🟡 **How are transitions handled?** — The current state performs the action and sets the context's next state.
4. 🟡 **Dart 3 alternative to classic State?** — A `sealed` class of states + exhaustive `switch` in the context (data-oriented state machine).
5. 🟡 **Why keep states stateless?** — So they're reusable/shareable (`const`) and free of cross-use state bugs.

6. 🔴 **How does State prevent invalid transitions?** — Each state only implements the transitions valid from it (others are no-ops or errors), making illegal moves explicit.
7. 🔴 **How does State relate to BLoC?** — BLoC often models states as a sealed family and transitions via events — the data-oriented cousin of the State pattern.

Senior Engineer Tips

- Draw the state diagram first; it maps 1:1 to states/transitions and reveals illegal moves.
- For UI, prefer sealed-class states + exhaustive `switch` (compiler-enforced completeness) over classic State classes.
- Keep transition rules with the state that owns them; centralizing them re-creates the conditional sprawl.

Architect Perspective

State turns implicit, bug-prone modal logic into explicit, testable state machines — critical for flows like auth, checkout, onboarding, and media. The Dart 3 sealed-class variant integrates directly with modern state management, making the full state space explicit and exhaustively handled (Modules 11, 01).

Summary

- State encapsulates per-state behavior + transitions as objects; context delegates.
- Removes conditional sprawl; keep states stateless; deny invalid transitions.
- Distinct from Strategy (internal-state vs client-chosen); Dart 3 sealed classes are the modern variant.

Revision Notes

- State = behavior varies by internal state; states define transitions.
- Context delegates to current state; keep states stateless (`const`).
- State (internal, self-transitioning) vs Strategy (client-chosen algorithm).
- Dart 3: sealed states + exhaustive `switch` ; ties to BLoC.

Practice Questions

1. How does State differ from Strategy?
2. Why keep state objects stateless?
3. When would you use sealed classes instead of State classes?

Coding Questions

1. Add a `Buffering` state to the media player with valid transitions.
2. Model a `Turnstile` (Locked/Unlocked) with coin/push events.
3. Reimplement the player as a sealed-class state machine with exhaustive `switch`.

Mini Project

Order lifecycle state machine (pure Dart): Model `Draft → Placed → Shipped → Delivered` (+ `Cancelled`) as State classes with valid transitions and denied illegal ones; then reimplement with sealed classes + exhaustive `switch`. Test all valid/invalid transitions. Acceptance: no conditional sprawl; illegal transitions blocked; both implementations tested; `dart analyze` clean.

Template Method Pattern

> Template Method defines the skeleton of an algorithm in a base class and lets subclasses fill in specific steps — the overall structure is fixed; the details vary.

Introduction

Template Method puts the invariant algorithm structure in one place (a `final`/non-overridable method) that calls overridable "hook" steps. Subclasses customize the steps without changing the sequence.

Why this concept exists

Many algorithms share a fixed sequence but differ in a few steps (parse → validate → transform → save, with format-specific parse/transform). Duplicating the whole sequence per variant is wasteful and drift-prone. Template Method centralizes the sequence and isolates the variation.

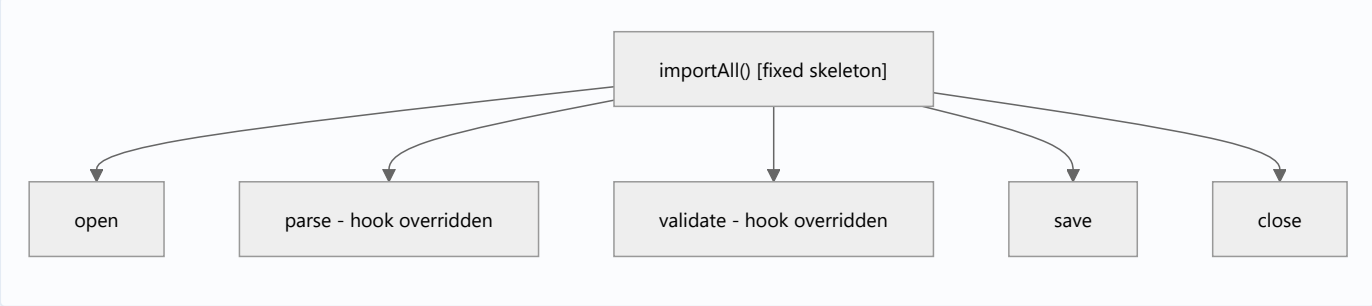
Real-world analogy

A **recipe template** for beverages: boil water → add main ingredient → pour → add condiments. Tea and coffee follow the same steps but "add main ingredient" and "condiments" differ. The overall procedure is fixed; two steps vary.

Problem Statement

A data-import pipeline always does open → read → parse → validate → save → close, but parse/validate differ per format (CSV vs JSON). You'll fix the sequence in a base class and override only the varying steps.

Internal Working



- A base class defines the **template method** (the fixed sequence) — ideally non-overridable.

- **Primitive/hook operations** (abstract or overridable) are the variable steps subclasses implement.
- Optional **hooks** with default no-op bodies let subclasses opt in.
- Contrast: Template Method uses **inheritance**; Strategy achieves similar variation via **composition** (often preferred).

Memory Representation

Not applicable — ordinary inheritance.

Compiler Behavior / Runtime Behavior

Subclass hook implementations are dispatched during the template method's fixed flow (dynamic dispatch).

Flutter Engine Behavior

Not applicable, but the `State` lifecycle is Template-Method-like: the framework calls `initState` → `build` → `dispose` in a fixed order; you override the steps (Module 08).

Dart VM Behavior

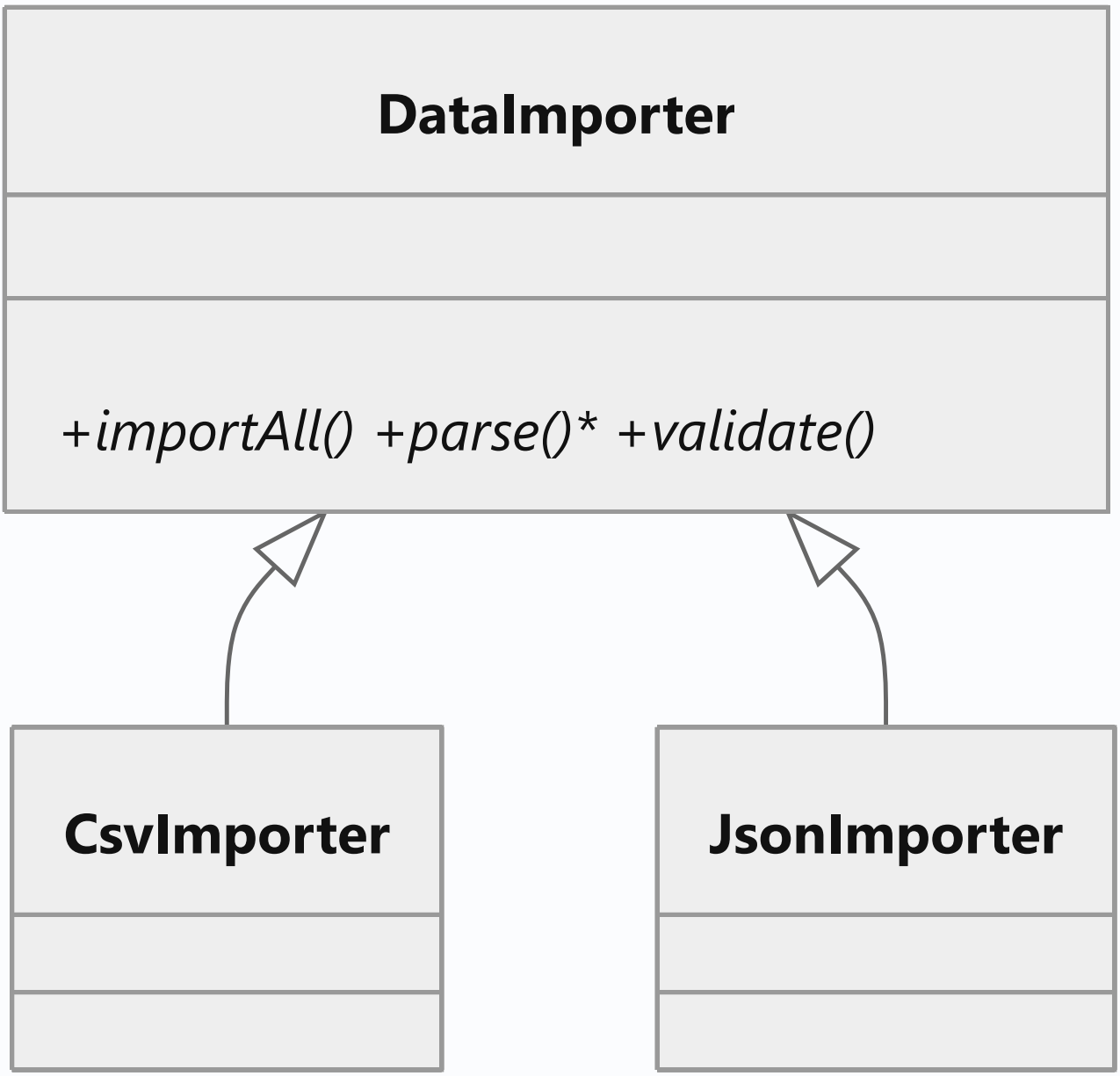
Not applicable.

Examples

```
abstract class DataImporter {
  // Template method: fixed skeleton (don't override)
  Future<int> importAll(String source) async {
    final raw = await _open(source);
    final rows = parse(raw);        // hook (varies)
    final valid = rows.where(validate).toList(); // hook (varies)
    await _save(valid);
    _close();
    return valid.length;
  }

  Future<String> _open(String s) async => s;      // shared
  Future<void> _save(List<Map<String, String>> rows) async =>
      print('saved ${rows.length}');              // shared
```

```
  void _close() => print('closed');               // shared

  // steps subclasses customize:
  List<Map<String, String>> parse(String raw);
  bool validate(Map<String, String> row);
}

class CsvImporter extends DataImporter {
  @override
  List<Map<String, String>> parse(String raw) =>
      raw.split('\n').where((l) => l.isNotEmpty).map((l) {
        final c = l.split(',');
        return {'name': c[0], 'age': c[1]};
      }).toList();
  @override
  bool validate(Map<String, String> row) => (int.tryParse(row['age']!) ?? -1) >= 0;
}

Future<void> main() async {
  final n = await CsvImporter().importAll('Ada,36\nBob,-1\nEve,29');
  print('imported $n'); // saves 2 valid rows -> imported 2
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Allowing the template method to be overridden	Breaks the fixed structure	Make it non-overridable (`base` /documented)
Too many abstract hooks	Subclass burden	Provide sensible default hooks
Using inheritance where composition fits	Rigid coupling	Prefer Strategy for swappable steps
Hidden step ordering assumptions	Fragile subclasses	Document the contract of each hook

Best Practices

- Keep the **template method fixed**; expose only the intended hook steps.
- Provide **default hook implementations** where reasonable to reduce subclass burden.
- Prefer **Strategy/composition** when steps should be swappable at runtime or reused across hierarchies.
- Document each hook's contract (inputs, when called, expected side effects).

Performance

Negligible; ordinary virtual calls.

Advantages / Disadvantages

- + Eliminates duplicated algorithm structure, centralizes control flow, isolates variation.
- − Inheritance coupling (fragile base class), less flexible than composition, single-inheritance limit.

Interview Questions

1. 🟢 **What is Template Method?** — A base class defines an algorithm's fixed skeleton and delegates variable steps to overridable methods.
2. 🟢 **Template Method vs Strategy?** — Template Method varies steps via **inheritance** (fixed skeleton, overridden hooks); Strategy varies the whole algorithm via **composition** (swappable at runtime).
3. 🟡 **Why should the template method not be overridable?** — To guarantee the invariant sequence; subclasses customize only the designated steps.
4. 🟡 **What are hooks?** — Overridable steps, sometimes with default (often no-op) implementations subclasses can opt into.
5. 🟡 **Where does Flutter exhibit Template Method?** — The `State` lifecycle: the framework calls fixed lifecycle methods in order; you override the steps.
6. 🔴 **When prefer Strategy over Template Method?** — When steps should be swapped at runtime, reused across unrelated types, or to avoid inheritance coupling.
7. 🔴 **What's the main risk of Template Method?** — Fragile base class: changes to the skeleton ripple to all subclasses.

Senior Engineer Tips

- If you need to vary steps at runtime or across hierarchies, refactor Template Method → Strategy (inject step functions).

- Provide default hooks so common subclasses stay tiny; reserve abstract hooks for truly required variation.
- Use Dart's `base` / `final` modifiers to lock the skeleton and communicate intent.

Architect Perspective

Template Method encodes a canonical process (import pipelines, request lifecycles, build flows) once, ensuring consistency while allowing controlled variation. For flexibility-sensitive code, its composition-based sibling (Strategy) usually scales better — a recurring inheritance-vs-composition decision (Module 03).

Summary

- Template Method fixes an algorithm's skeleton and lets subclasses fill in steps.
- Keep the skeleton non-overridable; provide default hooks; document hook contracts.
- Prefer Strategy/composition when runtime swapping or cross-hierarchy reuse is needed.

Revision Notes

- Template Method = fixed skeleton + overridable steps (inheritance).
- Skeleton non-overridable; hooks = variable steps (with defaults).
- Template Method (inheritance) vs Strategy (composition, runtime swap).
- Flutter `State` lifecycle is Template-Method-like.

Practice Questions

1. How does Template Method differ from Strategy?
2. Why lock the template method?
3. When would you refactor to Strategy?

Coding Questions

1. Add a `JsonImporter` subclass overriding parse/validate.
2. Build a `ReportGenerator` template (`header` / `body` / `footer` hooks).
3. Refactor a Template Method pipeline into an injectable Strategy-based one.

Mini Project

Import pipeline (pure Dart): Implement `DataImporter` with a fixed `importAll` skeleton and `CsvImporter` / `JsonImporter` overriding parse/validate; add a default hook (`onSkippedRow`). Then provide a Strategy-based variant and compare. Acceptance: skeleton fixed; hooks documented; both approaches tested; `dart analyze` clean.

Chain of Responsibility Pattern

> Chain of Responsibility passes a request along a chain of handlers until one handles it — decoupling sender from receiver and letting you add/reorder handlers freely.

Introduction

Chain of Responsibility (CoR) links handlers so each either processes a request or passes it to the next. The sender doesn't know which handler will act. It's the pattern behind middleware pipelines, event bubbling, and validation chains.

Why this concept exists

When multiple objects might handle a request and the handler isn't known in advance (or several should get a chance), hard-coding the selection couples the sender to all of them. CoR lets each handler decide independently, and lets you insert/remove/reorder handlers without touching the sender.

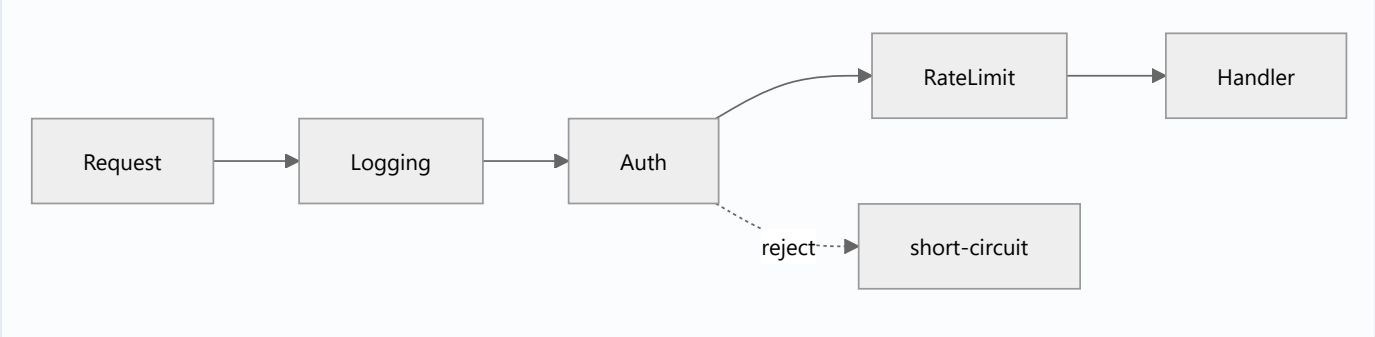
Real-world analogy

Escalating support tiers: your ticket goes to L1; if they can't solve it, it passes to L2, then L3. Each tier either resolves it or forwards it. You (sender) don't pick the tier; the chain routes it.

Problem Statement

An HTTP server must run a request through middleware: logging → auth → rate-limit → handler. Each stage may handle/short-circuit or pass along. You'll build a handler chain that's easy to reorder and extend.

Internal Working



- Each **handler** has a reference to the **next** and a `handle(request)` that either processes and/or delegates to `next`.
- A handler can **short-circuit** (stop the chain) or **pass through**.
- Dart-friendly: a list of handler **functions** (middleware) composed into a pipeline is often cleaner than a linked-object chain.

Memory Representation

A linked list of handlers (or a list of functions); negligible.

Compiler Behavior / Runtime Behavior

Requests traverse the chain at runtime until handled or exhausted.

Flutter Engine Behavior

Conceptually related: **gesture/event propagation** and `Notification` bubbling travel up the tree until handled (Module 08).

Dart VM Behavior

Not applicable.

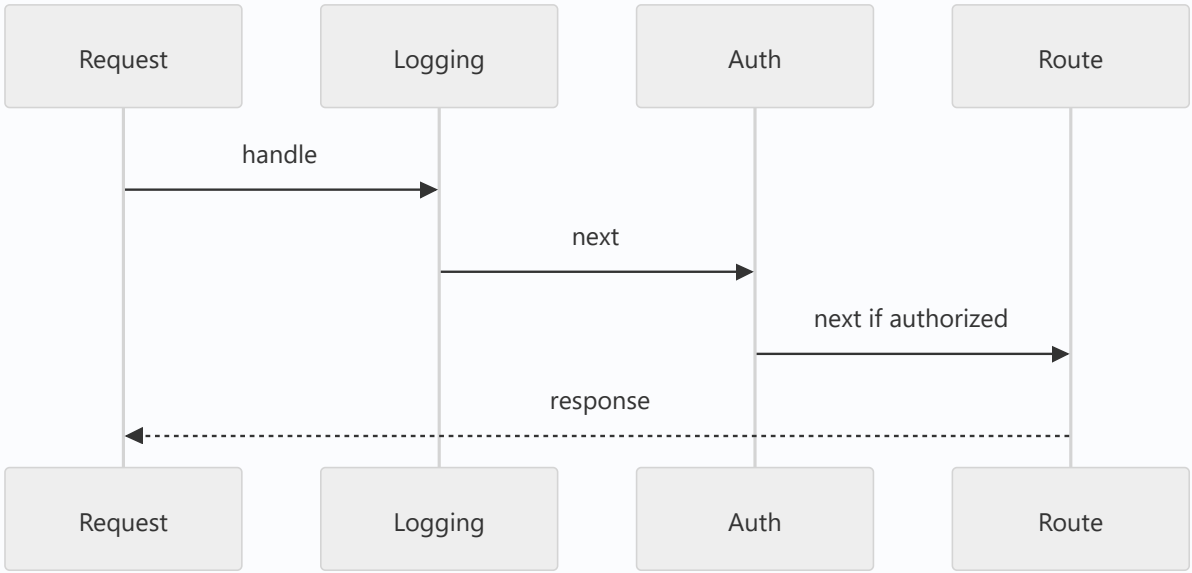
Examples

```
class Request {
  final String path;
  final Map<String, String> headers;
  Request(this.path, this.headers);
}

abstract class Handler {
  Handler? _next;
  Handler setNext(Handler next) { _next = next; return next; } // chain builder
  String handle(Request req) => _next?.handle(req) ?? 'no handler';
}

class LoggingHandler extends Handler {
  @override
  String handle(Request req) {
```

```
    print('LOG ${req.path}');
    return super.handle(req); // pass along
  }
}
class AuthHandler extends Handler {
  @override
  String handle(Request req) {
    if (!req.headers.containsKey('Authorization')) {
      return '401 Unauthorized'; // short-circuit
    }
    return super.handle(req);
  }
}
class RouteHandler extends Handler {
  @override
  String handle(Request req) =>
      req.path == '/home' ? '200 OK home' : '404 Not Found';
}

void main() {
  final chain = LoggingHandler();
  chain.setNext(AuthHandler()).setNext(RouteHandler()); // build the chain

  print(chain.handle(Request('/home', {'Authorization': 'x'}))); // 200 OK home
  print(chain.handle(Request('/home', {})));                     // 401 Unauthorized
}
```

```
// Idiomatic Dart: middleware as a list of functions
typedef Next = String Function();
typedef Middleware = String Function(Request req, Next next);
// compose a List<Middleware> into a pipeline — easy to reorder/insert.
```

Diagrams



Common Mistakes

Mistake	Why	Fix
No terminal handler	Request falls off the end unhandled	Provide a default/terminal handler
Handlers with hidden dependencies on order	Fragile	Make ordering explicit and documented
Handler doing too much	SRP violation	One concern per handler
Silent drop (neither handle nor pass)	Lost requests	Ensure every handler passes or handles

Best Practices

- Ensure a **terminal handler** (default response) so requests are always resolved.
- Keep each handler **single-purpose**; compose the pipeline explicitly.
- Prefer a **list-of-functions middleware** pipeline in Dart for readability/reordering.
- Allow handlers to short-circuit clearly (return a result) vs pass (`next`).

Performance

O(chain length) per request; keep chains reasonable and handlers light.

Advantages / Disadvantages

- + Decouples sender from handlers, easy to add/reorder/remove, single-purpose stages.

- − Request may go unhandled if misconfigured; harder to trace flow; ordering matters.

Interview Questions

1. 🟢 **What does Chain of Responsibility do?** — Passes a request along a chain of handlers until one handles it, decoupling sender from receiver.
2. 🟢 **Give a real example.** — HTTP middleware (logging/auth/rate-limit), support escalation, event bubbling.
3. 🟡 **How does a handler short-circuit?** — By handling the request and not calling `next` (returning a result immediately).
4. 🟡 **What's the risk if there's no terminal handler?** — Requests fall off the end unhandled; always provide a default.
5. 🟡 **Dart-idiomatic form?** — A composed list of middleware functions rather than a linked chain of objects.
6. 🔴 **CoR vs Decorator?** — Both form chains; Decorator always delegates and augments (all layers run), CoR may stop at the first handler that acts.
7. 🔴 **Where is CoR-like behavior in Flutter?** — Gesture/notification propagation bubbling up the widget tree until handled.

Senior Engineer Tips

- Model cross-cutting request processing (auth, logging, validation, rate-limiting) as ordered middleware — trivially testable per stage.
- Make the pipeline data (a list) so ordering/feature-flagging is configuration, not code edits (OCP).
- Always include a catch-all terminal handler.

Architect Perspective

CoR structures request/event processing into composable, reorderable stages — the model for interceptors, middleware, and validation pipelines across networking, auth, and event systems (Modules 16, 17). It keeps cross-cutting concerns modular and independently testable.

Summary

- CoR passes a request through handlers until one handles it; sender is decoupled.
- Ensure a terminal handler; keep handlers single-purpose; prefer function-based middleware in Dart.
- Related to Decorator (chain) but may short-circuit; mirrors event bubbling.

Revision Notes

- CoR = request travels handlers until handled; may short-circuit.
- Terminal/default handler required; one concern per handler.
- Dart: middleware as list of functions; ordering = config (OCP).
- Vs Decorator (all run + augment); Flutter event bubbling is CoR-like.

Practice Questions

1. How does a handler short-circuit the chain?
2. Why is a terminal handler necessary?
3. How does CoR differ from Decorator?

Coding Questions

1. Add a `RateLimitHandler` to the middleware chain.
2. Reimplement the chain as a composed list of middleware functions.
3. Build a validation chain (`nonEmpty` → `email` → `maxLength`) returning the first failure.

Mini Project

Request middleware pipeline (pure Dart): Implement a CoR pipeline (logging/auth/rate-limit/route) both as linked handlers and as composed middleware functions, with a terminal 404 handler. Test short-circuit and pass-through paths, and reordering. Acceptance: no unhandled requests; handlers single-purpose; ordering configurable; `dart analyze` clean.

Mediator Pattern

> Mediator centralizes complex communication between objects so they talk through a hub instead of referencing each other directly — turning a tangled many-to-many web into a clean hub-and-spoke.

Introduction

Mediator introduces a central object that coordinates interactions among "colleague" objects. Colleagues notify the mediator of events; the mediator decides how others should react. This removes direct references between colleagues.

Why this concept exists

When many objects interact directly, you get an `N×N` web of references — a change to one ripples everywhere and reuse becomes impossible. Mediator collapses that into `N` relationships (each colleague ↔ mediator), centralizing the interaction logic in one place.

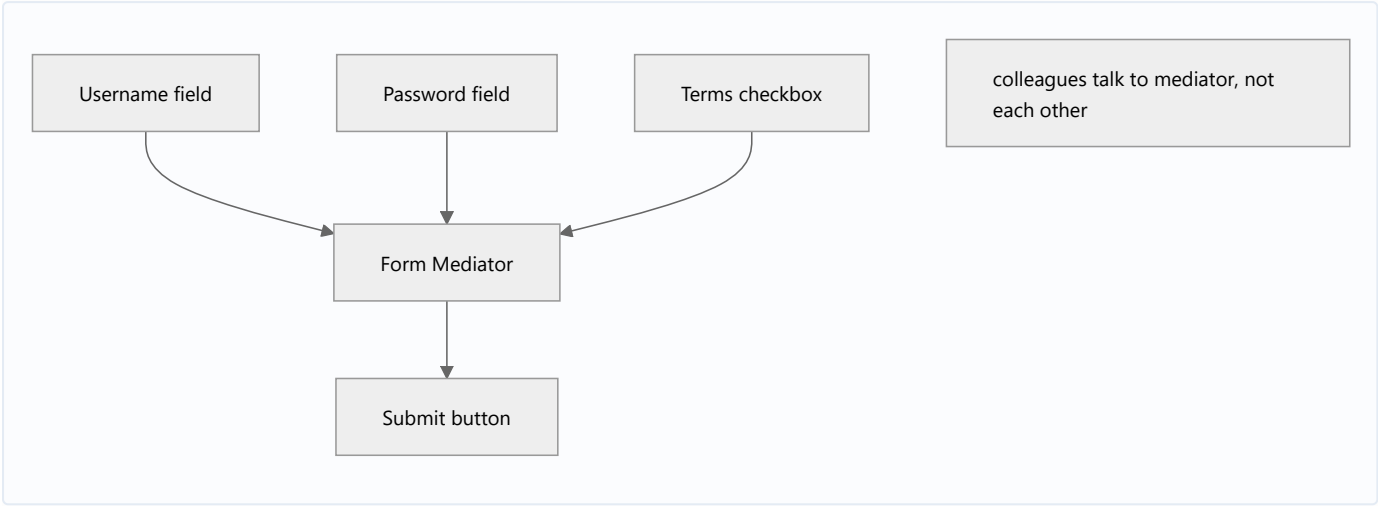
Real-world analogy

Air traffic control: planes (colleagues) don't coordinate landings by talking to each other directly (chaos); they all talk to the control tower (mediator), which sequences everyone. One hub, clear rules.

Problem Statement

A form has fields whose interactions are tangled: enabling "Submit" depends on username + password + terms; changing one field affects others. Wiring each field to every other field is a mess. You'll route all changes through a mediator (the form controller).

Internal Working



- **Mediator interface** with a `notify(sender, event)` method.
- **Colleagues** hold a reference to the mediator (not to each other) and call `notify` on changes.
- The **concrete mediator** encapsulates the interaction rules and updates colleagues.
- In Flutter, a state controller/BLoC/ViewModel often *is* the mediator for a screen's widgets.

Memory Representation

Colleagues reference the mediator; the mediator references colleagues — a hub, not a mesh.

Compiler Behavior / Runtime Behavior

Events flow to the mediator, which orchestrates responses at runtime.

Flutter Engine Behavior

Not applicable, but a screen's **controller/ViewModel/BLoC** acts as a mediator between its widgets, centralizing cross-widget interaction (Modules 11, 43).

Dart VM Behavior

Not applicable.

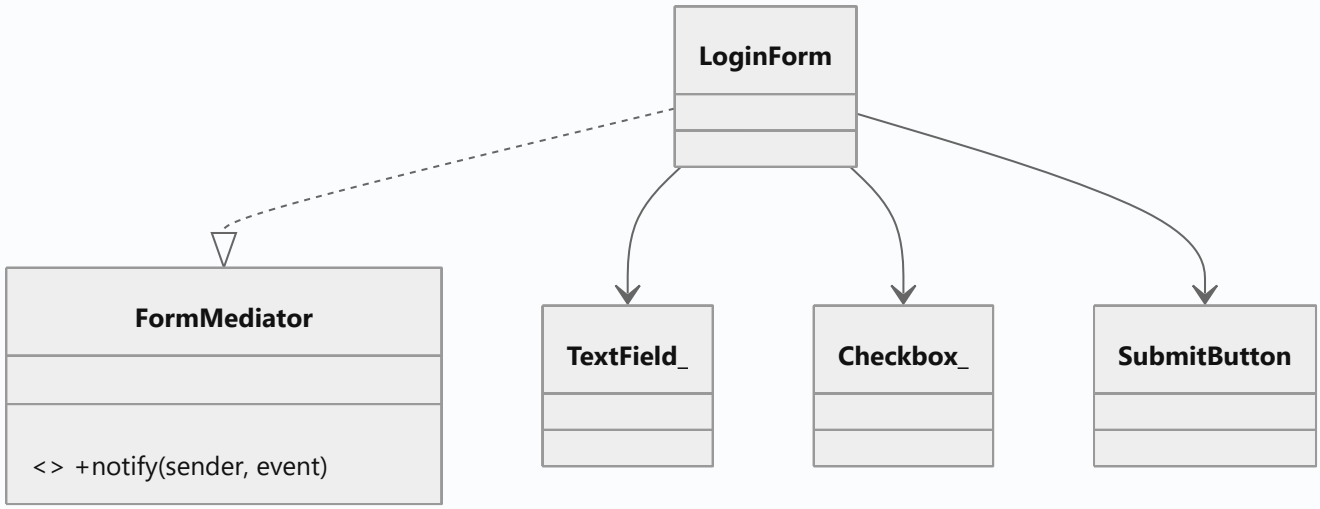
Examples

```
abstract interface class FormMediator {
  void notify(String sender, String event);
}
```

```
class TextField_ {
  final String name;
  final FormMediator mediator;
  String value = '';
  TextField_(this.name, this.mediator);
  void setValue(String v) {
    value = v;
    mediator.notify(name, 'changed'); // tell the hub, not other fields
  }
}

class Checkbox_ {
  final FormMediator mediator;
  bool checked = false;
  Checkbox_(this.mediator);
  void toggle() {
    checked = !checked;
    mediator.notify('terms', 'changed');
  }
}

class SubmitButton {
  bool enabled = false;
}

class LoginForm implements FormMediator {
  late final TextField_ username = TextField_('username', this);
  late final TextField_ password = TextField_('password', this);
  late final Checkbox_ terms = Checkbox_(this);
  final SubmitButton submit = SubmitButton();

  @override
  void notify(String sender, String event) {
    // centralized interaction rules:
    submit.enabled = username.value.isNotEmpty &&
        password.value.length >= 6 &&
        terms.checked;
    print('submit enabled: ${submit.enabled}');
  }
}
```

```
void main() {
  final form = LoginForm();
  form.username.setValue('ada');   // submit enabled: false
  form.password.setValue('secret'); // submit enabled: false
  form.terms.toggle();              // submit enabled: true
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Colleagues still referencing each other	Defeats the pattern	Route all interaction through the mediator
Mediator becoming a God object	Absorbs all logic	Split by screen/area; keep colleagues cohesive
Overusing for simple interactions	Needless indirection	Use direct calls for trivial cases
Mediator with hidden global state	Coupling	Scope mediators; inject dependencies

Best Practices

- Route colleague interactions through the mediator; colleagues stay ignorant of each other.
- Keep mediators **scoped** (per screen/feature) to avoid God objects.
- Recognize that a **ViewModel/BLoC/controller** is your Flutter mediator — put cross-widget rules there.
- Keep colleagues cohesive (SRP); mediator owns only the *coordination*.

Performance

Negligible; centralized dispatch.

Advantages / Disadvantages

- + Reduces coupling (N×N → N), centralizes interaction logic, reusable colleagues.
- − Mediator can bloat into a God object; centralization can hide flow.

Interview Questions

1. 🟢 **What does Mediator do?** — Centralizes communication between objects so they interact via a hub rather than direct references.
2. 🟢 **What problem does it solve?** — The tangled many-to-many (`N×N`) coupling of objects that reference each other.
3. 🟡 **Mediator vs Observer?** — Observer is one-to-many notification (subject→observers); Mediator coordinates many-to-many interactions with centralized *logic*.
4. 🟡 **Mediator vs Facade?** — Facade is a one-way simplifier over a subsystem; Mediator manages bidirectional interaction among peers.
5. 🟡 **What's the Flutter analogue?** — A screen's ViewModel/BLoC/controller mediating its widgets.
6. 🔴 **What's the main risk?** — The mediator becoming a God object; mitigate by scoping per feature and keeping colleagues cohesive.
7. 🔴 **When is Mediator overkill?** — For a couple of objects with simple, stable interactions — direct calls are clearer.

Senior Engineer Tips

- In Flutter, don't wire widgets to each other — let the controller/BLoC mediate; that's the pattern in action and keeps widgets dumb.
- Scope mediators tightly (one per screen/flow) to prevent God-object drift.
- Keep coordination rules in the mediator and domain rules in the domain — don't conflate them.

Architect Perspective

Mediator localizes interaction complexity, which is exactly the role of presentation-layer controllers (ViewModel/BLoC) coordinating UI components. It keeps components reusable and decoupled, and confines cross-component logic to one testable place — a cornerstone of MVVM/BLoC screen design (Modules 43, 11).

Summary

- Mediator centralizes many-to-many interaction into a hub; colleagues don't reference each other.
- Reduces coupling but risks God-object bloat — scope it per feature.
- Your Flutter ViewModel/BLoC is a mediator; distinct from Observer (notify) and Facade (simplify).

Revision Notes

- Mediator = hub for object interaction (N×N → N); colleagues ↔ mediator only.
- Risk: God object → scope per screen/feature.
- Mediator (bidirectional coordination) vs Observer (notify) vs Facade (simplify).
- Flutter: ViewModel/BLoC/controller = mediator.

Practice Questions

1. How does Mediator differ from Observer and Facade?
2. Why can a mediator become a God object, and how do you prevent it?
3. Why is a BLoC/ViewModel a mediator?

Coding Questions

1. Build a `WizardMediator` coordinating next/back/step-validity across steps.
2. Model a chat-room mediator routing messages between users.
3. Refactor mutually-referencing UI components to communicate via a mediator.

Mini Project

Login form mediator (pure Dart): Implement a `LoginForm` mediator coordinating username/password/terms/submit with centralized enable rules; colleagues never reference each other. Test that field changes correctly toggle submit. Acceptance: no colleague-to-colleague references; rules centralized; mediator scoped to the form; `dart analyze` clean.

Visitor Pattern

> Visitor lets you add new operations to a set of object types **without modifying those types** — you move the operation into a separate "visitor" object that each element accepts.

Introduction

Visitor separates an algorithm from the object structure it operates on. Elements expose an `accept(visitor)` method; the visitor has a `visitXxx` method per element type. Adding a new operation = a new visitor, not edits to every element class.

Why this concept exists

When you have a stable set of types and *many* operations over them (a compiler AST: type-check, optimize, print, evaluate), putting every operation as a method on every type bloats them and violates SRP/OCP. Visitor externalizes operations so new ones don't touch the types. The tradeoff: adding a new *type* touches every visitor.

Real-world analogy

A **tax inspector** (visitor) visits different businesses (elements) — a restaurant, a factory, a shop — applying the right assessment for each. To add a new kind of inspection (safety audit), you send a new inspector; the businesses don't change.

Problem Statement

You have a stable `Shape` hierarchy (`Circle` , `Square`) and keep needing new operations (area, perimeter, export-to-SVG, describe). Adding each as a method on every shape bloats them. You'll add operations as visitors instead.

Internal Working

- **Element interface:** `accept(Visitor v)` → calls `v.visitConcrete(this)` (double dispatch).
- **Visitor interface:** one `visitX(X)` per concrete element.
- **Concrete visitors** implement an operation across all element types.

- **Double dispatch:** the element picks the visit method by its type; the visitor picks the operation — together selecting the right code.
- Dart alternative: **sealed classes + exhaustive** `switch` achieves the same "operations outside the types" with less ceremony.

Memory Representation

Not applicable — ordinary objects; visitors are usually stateless (or accumulate a result).

Compiler Behavior / Runtime Behavior

Double dispatch resolves at runtime via `accept` → `visitX`. With sealed classes, the compiler enforces handling all types in the `switch`.

Flutter Engine Behavior

Not applicable directly. (The element tree is walked by visitor-like mechanisms; render tree operations traverse nodes.)

Dart VM Behavior

Not applicable.

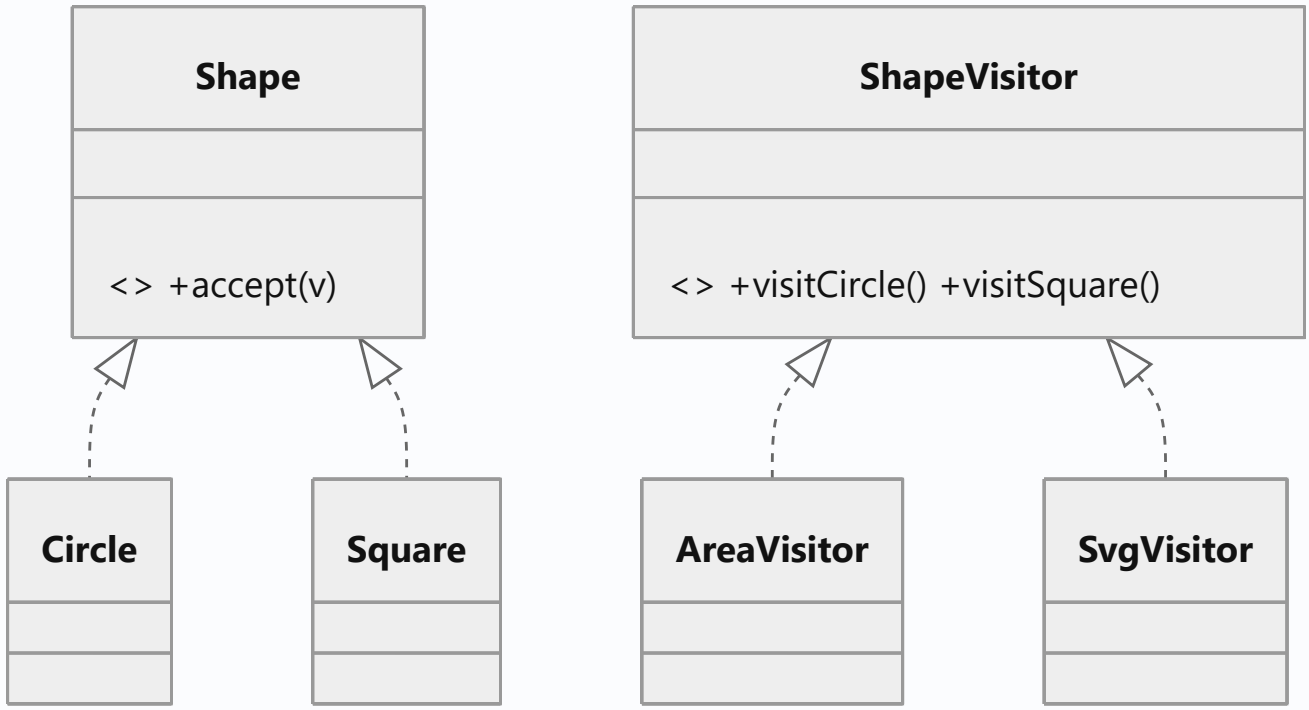
Examples

```
// Classic Visitor
abstract interface class ShapeVisitor<R> {
  R visitCircle(Circle c);
  R visitSquare(Square s);
}

abstract interface class Shape {
  R accept<R>(ShapeVisitor<R> v);
}

class Circle implements Shape {
  final double r;
  Circle(this.r);
  @override
  R accept<R>(ShapeVisitor<R> v) => v.visitCircle(this);
}
```

```
class Square implements Shape {
  final double side;
  Square(this.side);
  @override
  R accept<R>(ShapeVisitor<R> v) => v.visitSquare(this);
}

// New operation = new visitor (no edits to Circle/Square)
class AreaVisitor implements ShapeVisitor<double> {
  @override
  double visitCircle(Circle c) => 3.14159 * c.r * c.r;
  @override
  double visitSquare(Square s) => s.side * s.side;
}
class SvgVisitor implements ShapeVisitor<String> {
  @override
  String visitCircle(Circle c) => '<circle r="${c.r}"/>';
  @override
  String visitSquare(Square s) => '<rect w="${s.side}"/>';
}

void main() {
  final shapes = <Shape>[Circle(2), Square(3)];
  final area = AreaVisitor();
  final svg = SvgVisitor();
  for (final s in shapes) {
    print('area=${s.accept(area)} svg=${s.accept(svg)}');
  }
}
```

```
// Dart 3 alternative: sealed classes + switch (no accept/visit boilerplate)
// sealed class Shape2 {}  ... final area = switch (shape) { Circle2() => ..., Square2() => ... };
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Visitor over an unstable type set	Every new type edits all visitors	Use only when types are stable
Visitor boilerplate where a `switch` suffices	Overhead	Prefer sealed classes + `switch` in Dart
Visitors mutating elements unexpectedly	Hidden coupling	Keep visitors focused; document side effects
Forgetting double dispatch (type-checking inside visitor)	Defeats the pattern	Use `accept` / `visitX`

Best Practices

- Use Visitor when the **type set is stable** but **operations grow**.
- In Dart, strongly consider **sealed classes + exhaustive** `switch` — same benefit, far less boilerplate, compiler-enforced completeness.
- Keep visitors single-operation (SRP); parameterize result type (`Visitor<R>`).

Performance

Negligible (two virtual calls per element). Sealed `switch` may be faster/clearer.

Advantages / Disadvantages

- **+** Add operations without editing types (OCP for operations), gathers a related operation in one place.
- **−** Adding a *type* touches every visitor; boilerplate; awkward vs Dart's sealed `switch`.

Interview Questions

1. 🟢 **What does Visitor enable?** — Adding new operations over a set of types without modifying those types.
2. 🟢 **What's the tradeoff?** — Easy to add operations, hard to add types (every visitor must handle the new type).
3. 🟡 **What is double dispatch?** — Two virtual calls (`accept` picks element type, `visitX` picks operation) select the correct code — Dart lacks multi-methods, so Visitor simulates it.
4. 🟡 **Dart 3 alternative to Visitor?** — Sealed classes + exhaustive `switch`: operations live outside the types with compiler-enforced completeness and no `accept` boilerplate.
5. 🟡 **When is Visitor appropriate?** — Stable type hierarchies with many/growing operations (ASTs, document models).
6. 🔴 **Why does Visitor violate OCP for types but satisfy it for operations?** — New operations are new visitors (closed types, open operations); new types force edits across all visitors.
7. 🔴 **Where is Visitor common?** — Compilers/interpreters (AST traversal), serialization over object graphs.

Senior Engineer Tips

- In modern Dart, reach for **sealed classes + `switch`** first; use classic Visitor mainly when integrating with code that expects the `accept`/`visit` shape.
- If your types change more often than your operations, Visitor is the wrong pattern — flip to methods-on-types or polymorphism.
- Parameterize the visitor's return type (`Visitor<R>`) for reusable traversal.

Architect Perspective

Visitor (or its sealed-class equivalent) is the tool for operation-rich, type-stable structures like ASTs, rule engines, and document models — keeping the many operations organized and the types clean. In Dart, sealed classes make this ergonomic and compiler-safe, aligning with pattern-matching state modeling (Module 02).

Summary

- Visitor adds operations to a stable type set without editing the types (via double dispatch).
- Easy to add operations, hard to add types; boilerplate-heavy.
- In Dart, prefer sealed classes + exhaustive `switch` for the same benefit.

Revision Notes

- Visitor = operations outside types via `accept` / `visitX` (double dispatch).
- Open for operations, closed for types (opposite of usual OCP).
- Dart alt: sealed classes + exhaustive `switch` (less boilerplate, compiler-checked).
- Use for stable types + many operations (ASTs).

Practice Questions

1. Why is adding a new type expensive with Visitor?
2. What is double dispatch and why is it needed?
3. When would you use sealed `switch` instead?

Coding Questions

1. Add a `PerimeterVisitor` without editing shape classes.
2. Reimplement the shapes example with sealed classes + `switch`.
3. Build an AST (`Num` / `Add` / `Mul`) with `EvalVisitor` and `PrintVisitor`.

Mini Project

Expression evaluator (pure Dart): Model an AST (`Num` , `Add` , `Mul`) and implement `EvalVisitor` and `PrintVisitor` via Visitor; then reimplement with sealed classes + exhaustive `switch` and compare. Acceptance: new operation added without editing node types; sealed variant compiler-exhaustive; `dart analyze` clean.

Iterator Pattern

> Iterator provides a way to traverse a collection's elements sequentially without exposing its internal representation — the abstraction behind every `for-in` loop.

Introduction

Iterator decouples traversal from the collection. A collection exposes an iterator that yields elements one at a time; clients traverse via a uniform interface regardless of the underlying structure (array, tree, linked list). Dart bakes this in via `Iterable` / `Iterator` and `for-in`.

Why this concept exists

Different collections store data differently, but clients want a uniform way to walk them without knowing (or coupling to) the internals. Iterator provides that uniform traversal contract and allows multiple simultaneous/independent traversals and custom iteration orders.

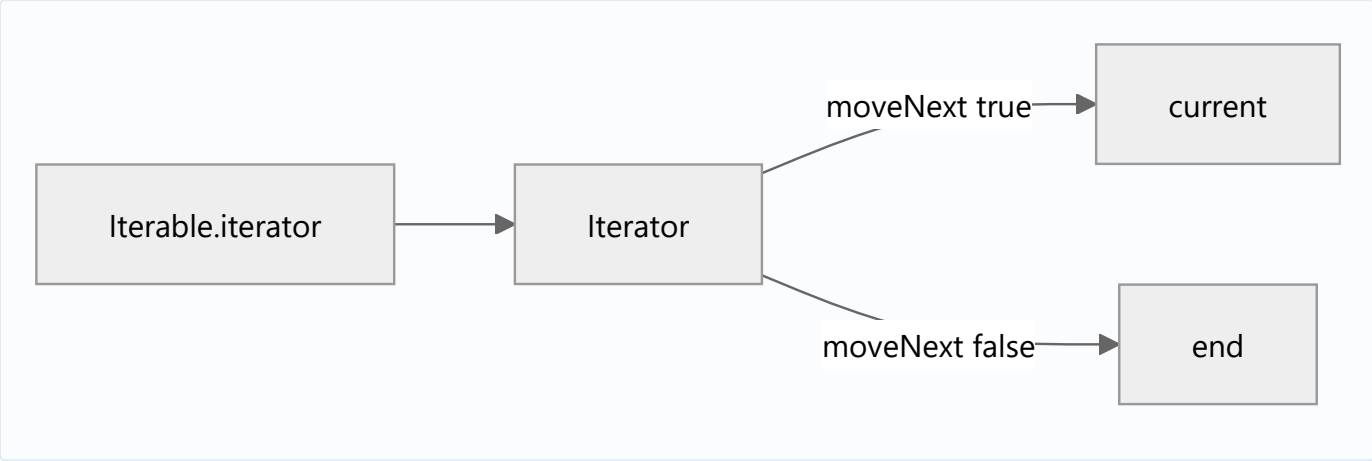
Real-world analogy

A **TV remote's channel-up button**: you move through channels one at a time without knowing how channels are stored internally. The remote (iterator) gives sequential access; the TV's storage stays hidden.

Problem Statement

You have a custom tree/collection and want clients to loop with `for-in` and use `map` / `where` without exposing its internal nodes. You'll implement `Iterable` / `Iterator` (and the far simpler `sync*` generator).

Internal Working



- **Iterator interface:** `moveNext()` advances and returns whether an element exists; `current` gives it.
- **Iterable interface:** exposes `iterator` (a fresh iterator each call → supports multiple traversals).
- Implementing `Iterable` / `Iterator` unlocks `for-in`, `map`, `where`, `fold`, etc., for free.
- `sync*` **generators** are the idiomatic Dart way to produce lazy iterables without hand-writing an iterator (01 · collections).

Memory Representation

An iterator holds a cursor/position into the collection; lazy iterables compute elements on demand (minimal memory).

Compiler Behavior

`for-in` desugars to `iterator` + `moveNext` / `current`. `sync*` compiles to a lazy iterator state machine.

Runtime Behavior

Traversal is lazy for `sync*` /lazy `Iterable`s; modifying a collection during iteration throws `ConcurrentModificationError`.

Flutter Engine Behavior

Not applicable, but lazy iteration underpins `ListView.builder`-style on-demand construction (compute items as needed).

Dart VM Behavior

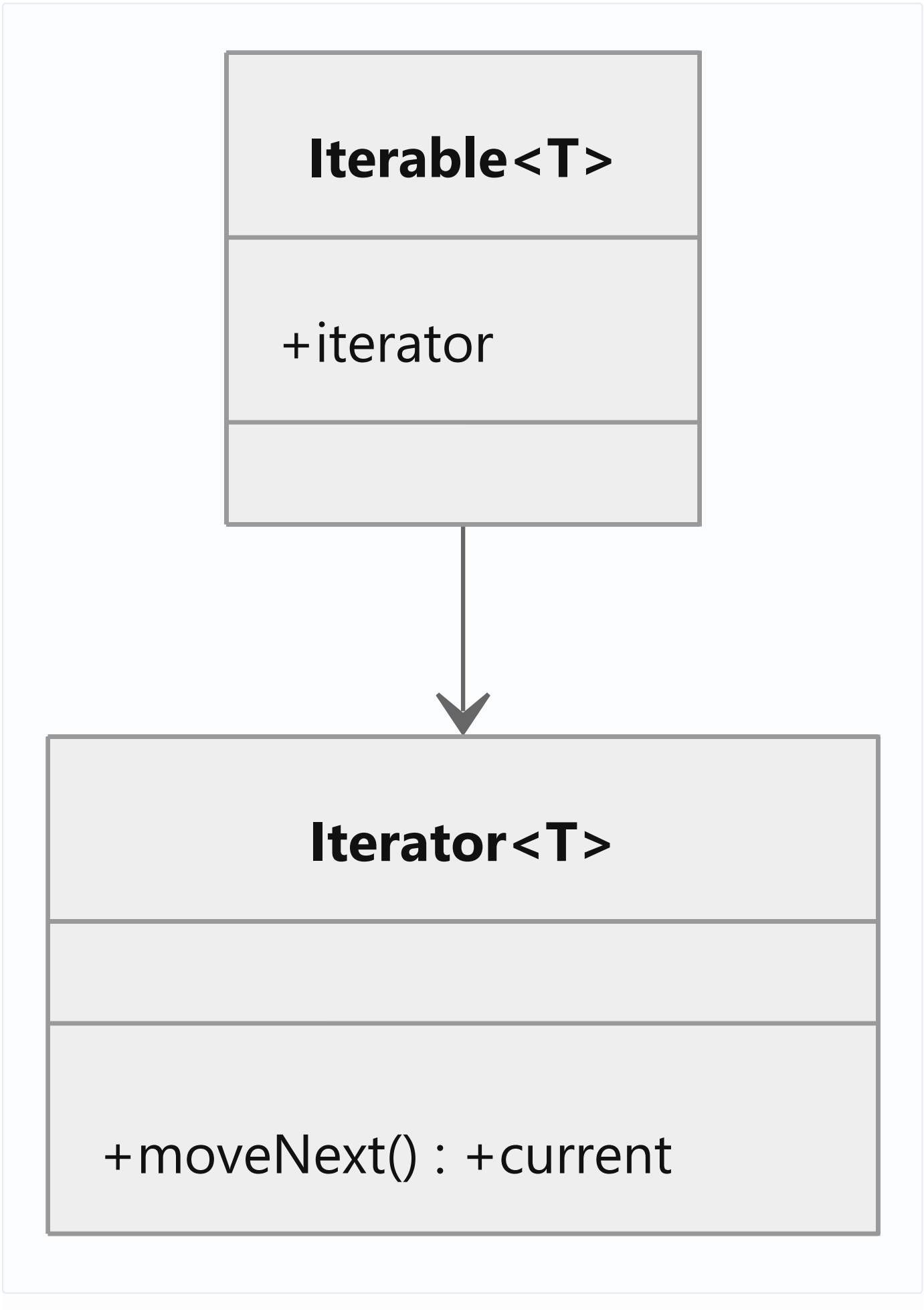
Not applicable.

Examples

```
// Custom collection made iterable via sync* (idiomatic Dart)
class Ring<T> extends Iterable<T> {
  final List<T> _items;
  Ring(this._items);

  @override
  Iterator<T> get iterator => _gen().iterator;

  Iterable<T> _gen() sync* {
    for (final x in _items) {
      yield x; // lazy, one at a time
    }
  }
}

// Hand-written Iterator (what for-in uses under the hood)
class Countdown extends Iterable<int> {
  final int from;
  Countdown(this.from);
  @override
  Iterator<int> get iterator => _CountdownIterator(from);
}
class _CountdownIterator implements Iterator<int> {
  int _n;
  _CountdownIterator(this._n) : _n = _n + 1; // will pre-decrement
  @override
  int current = 0;
  @override
  bool moveNext() {
    if (_n <= 0) return false;
    _n--;
    current = _n;
    return _n >= 0;
  }
}

void main() {
  final ring = Ring([1, 2, 3]);
```

```
  for (final x in ring) {
    print(x); // 1,2,3 — for-in works
  }
  print(ring.map((x) => x * 10).toList()); // [10,20,30] — Iterable methods free


  print(Countdown(3).toList()); // [3,2,1,0]


  // lazy generator:
  Iterable<int> naturals() sync* {
    var i = 0;
    while (true) {
      yield i++;
    }
  }
  print(naturals().take(4).toList()); // [0,1,2,3] — infinite, but lazy
}
```

Diagrams


Iterable<T>
+iterator
Iterator<T>
+moveNext() : +current

Common Mistakes

Mistake	Why	Fix
Hand-writing iterators when `sync*` works	Boilerplate/bugs	Use `sync*` generators
Mutating a collection while iterating	`ConcurrentModificationError`	Iterate a copy or collect changes
Sharing one iterator across loops	Iterators are single-pass	Get a fresh `iterator` per traversal
Infinite generator without a bound	Hangs	Use `take` /conditions

Best Practices

- Prefer `sync*` **generators** and extending `Iterable` over hand-rolled iterators.
- Return a **fresh iterator** each time (supports multiple independent traversals).
- Keep iteration **lazy** for large/infinite sequences; bound with `take`.
- Don't mutate during iteration.

Performance

Lazy iteration avoids materializing whole collections; `sync*` has small per-yield overhead — fine for typical use, avoid in ultra-hot inner loops if profiling shows cost.

Advantages / Disadvantages

- **+** Uniform traversal, hides internals, supports lazy/infinite sequences and multiple traversals, unlocks all `Iterable` methods.
- **−** Hand-written iterators are fiddly (use `sync*`); single-pass; mutation-during-iteration hazard.

Interview Questions

1. 🟢 **What does Iterator provide?** — Sequential access to a collection's elements without exposing its internal structure.
2. 🟢 **What's the Dart idiom for it?** — Extend `Iterable` and/or use `sync*` generators; `for-in` uses `iterator` / `moveNext` / `current`.
3. 🟡 **Why return a fresh iterator each call?** — Iterators are single-pass/stateful; fresh ones enable multiple independent traversals.
4. 🟡 **How do you make a lazy/infinite sequence?** — A `sync*` generator that `yield`s on demand, bounded by `take` /conditions.
5. 🟡 **Why does mutating during iteration throw?** — It invalidates the iterator's cursor; Dart raises `ConcurrentModificationError`.

6. 🔴 **`sync*` vs `async*`?** — `sync*` produces a lazy `Iterable` (pull); `async*` produces a `Stream` (push, async) (02 · streams).
7. 🔴 **How does implementing `Iterable` benefit you for free?** — You get `map` / `where` / `fold` / `take` / `for-in` etc. without extra code.

Senior Engineer Tips

- Almost never hand-write an `Iterator` in Dart — `sync*` or extending `Iterable` is cleaner and correct.
- Expose custom collections as `Iterable` to hide internals yet give clients the full functional toolkit.
- For async sequences, reach for `Stream` / `async*`, not Iterator.

Architect Perspective

Iterator standardizes traversal so client code is decoupled from data-structure internals — the basis of Dart's rich `Iterable` API and lazy pipelines. Lazy iteration underlies memory-efficient processing of large/streamed datasets, complementing streams for async and `ListView.builder` for on-demand UI (Modules 01, 21).

Summary

- Iterator gives uniform, internals-hiding sequential traversal; Dart provides it via `Iterable` / `Iterator` + `for-in`.
- Use `sync*` generators over hand-written iterators; keep iteration lazy; return fresh iterators.
- `sync*` = lazy `Iterable` (pull); `async*` = `Stream` (push).

Revision Notes

- Iterator = sequential access, hides internals; `moveNext` / `current`.
- Dart: extend `Iterable` / `sync*` generator → `for-in` + all Iterable methods.
- Fresh iterator per traversal; no mutation during iteration.
- `sync*` (Iterable) vs `async*` (Stream).

Practice Questions

1. Why prefer `sync*` over a hand-written `Iterator`?
2. How do you make a lazy infinite sequence safely?
3. `sync*` vs `async*`?

Coding Questions

1. Make a custom `Matrix` iterable in row-major order via `sync*`.
2. Implement a lazy `fibonacci()` generator and take the first 10.
3. Extend `Iterable` on a tree to yield nodes in-order.

Mini Project

Custom iterable collection (pure Dart): Implement a `LinkedList<T>` (or tree) that extends `Iterable<T>` using `sync*`, supporting `for-in`, `map`, `where`, and multiple independent traversals, plus a lazy generator utility. Add tests. Acceptance: internals hidden; fresh iterator per traversal; lazy where appropriate; `dart analyze` clean.

Repository Pattern

> A repository is an abstraction over data access that gives the rest of the app a clean, domain-oriented API — hiding whether data comes from network, database, or cache.

Introduction

The Repository pattern mediates between the domain and data sources. Callers use methods like `getUser(id)` / `saveOrder(order)` against an **interface**, unaware of the underlying REST API, local DB, or cache. It's arguably the single most important application pattern in Flutter apps.

Why this concept exists

Without a repository, UI/logic code calls HTTP clients and databases directly, coupling business logic to infrastructure, making testing require real servers/DBs, and scattering caching/error-mapping everywhere. Repository centralizes data access behind a domain interface, enabling swap, cache, and test in one place (DIP applied — Module 04).

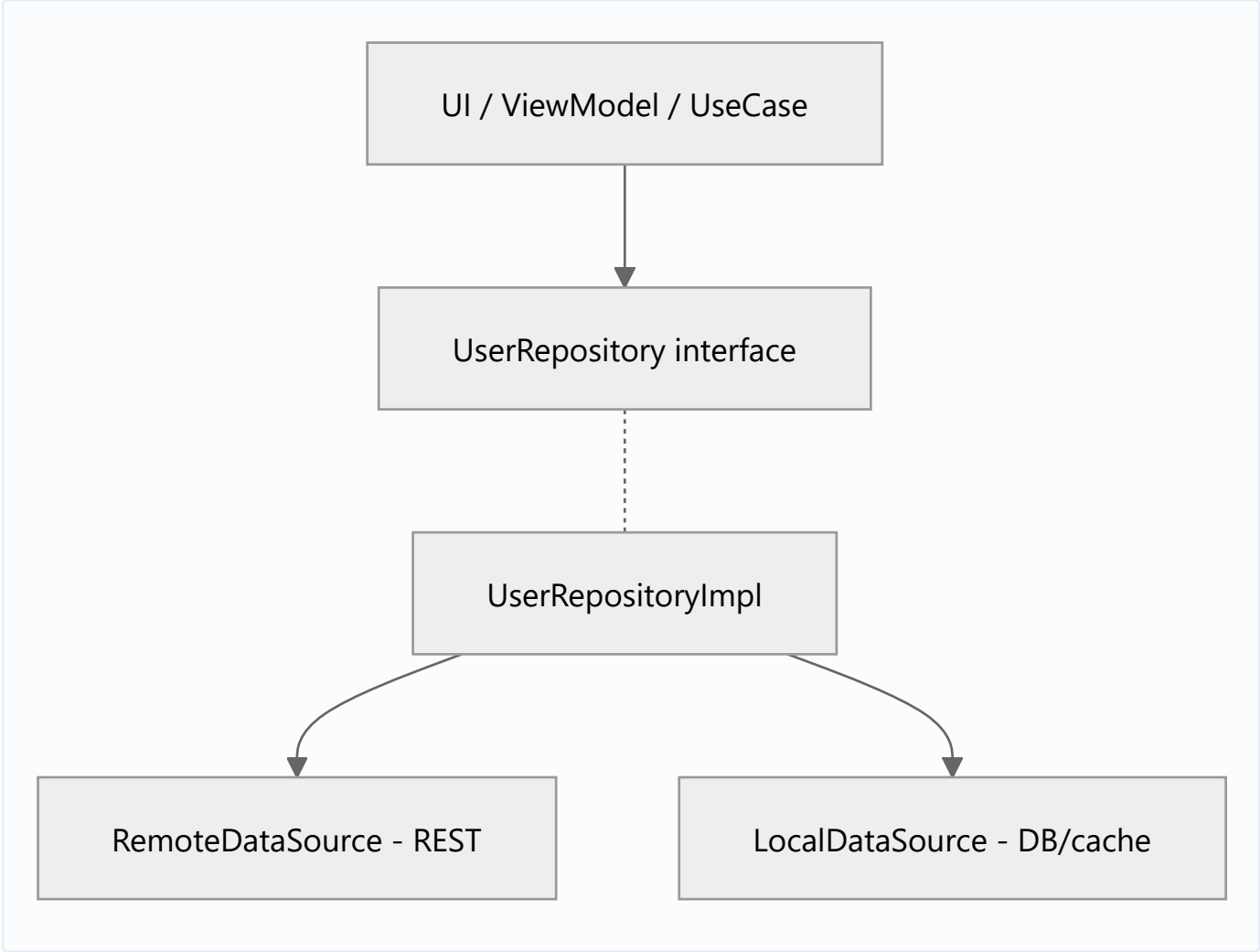
Real-world analogy

A **librarian**: you ask for a book by title; the librarian figures out whether it's on the shelf (cache), in storage (DB), or must be ordered (network). You don't care how it's fetched — you get the book through one simple request.

Problem Statement

Your app needs users from a REST API, cached locally, falling back to the DB offline. UI shouldn't know any of that. You'll define a `UserRepository` interface and an implementation coordinating remote + local sources.

Internal Working



- **Repository interface** in the domain, expressed in **domain types** (entities), returning `Future` / `Stream` (often `Result` — Module 38).
- **Implementation** orchestrates data sources, maps DTOs↔entities, applies caching/offline strategy, and translates errors.
- Depends on **data source abstractions** (also injected) — remote (HTTP/GraphQL) and local (DB/cache).

Memory Representation

The repo may hold an in-memory cache; bound it to avoid leaks (02 · memory_and_gc).

Compiler Behavior / Runtime Behavior

Callers depend on the interface (compile-time decoupling); at runtime the impl chooses source, maps data, and handles errors.

Flutter Engine Behavior

Not applicable. (Repositories sit in the data layer; ViewModels/BLoCs/use cases depend on them — Modules 11, 40.)

Dart VM Behavior

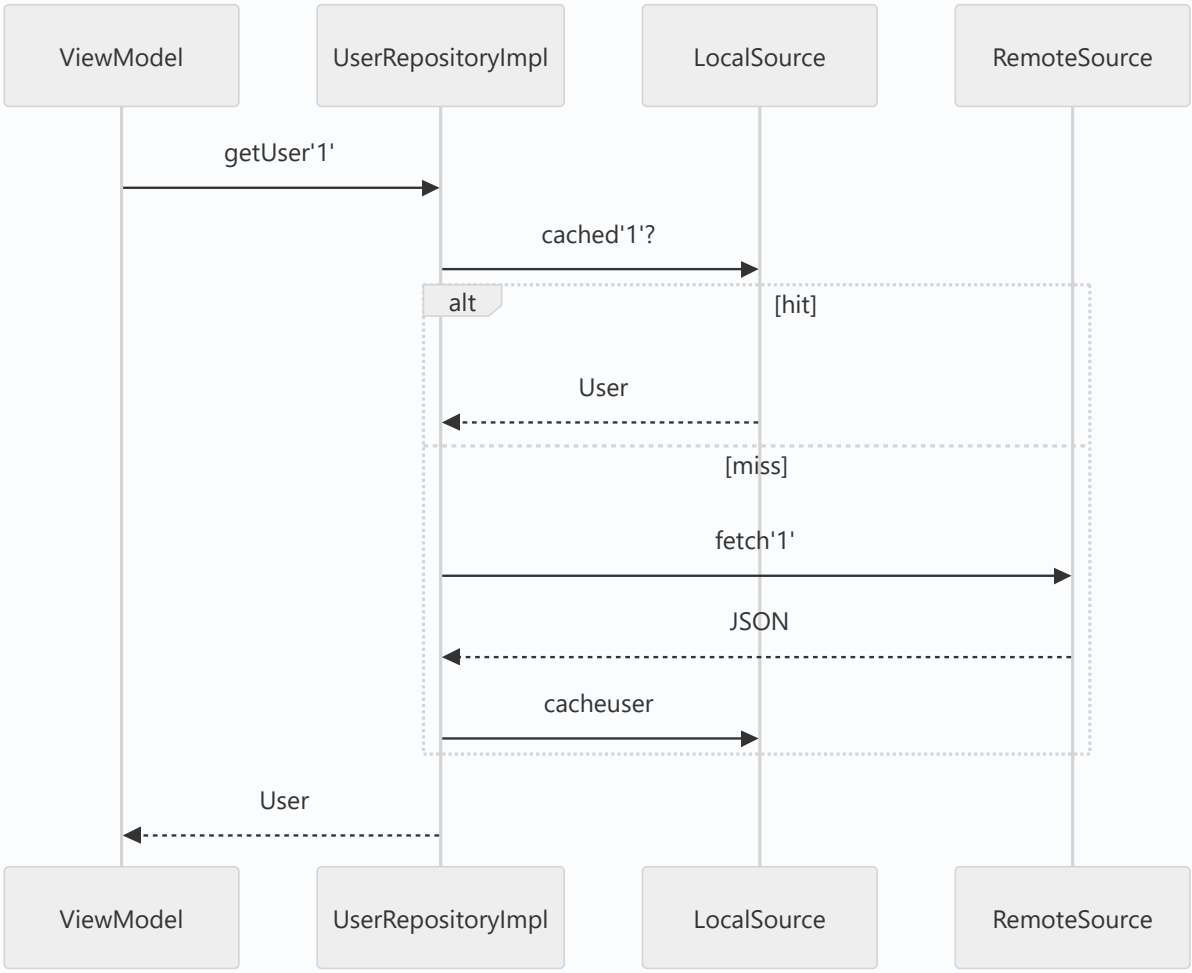
Not applicable.

Examples

```
// Domain entity (not the wire DTO)
class User {
  final String id, name;
  const User(this.id, this.name);
}

// Repository interface (domain-facing, domain types)
abstract interface class UserRepository {
  Future<User> getUser(String id);
  Future<List<User>> getAll();
}

// Data source abstractions
abstract interface class UserRemoteSource {
  Future<Map<String, dynamic>> fetch(String id);
}
abstract interface class UserLocalSource {
  User? cached(String id);
  void cache(User u);
}

// Implementation: orchestrates sources, maps, caches, handles errors
class UserRepositoryImpl implements UserRepository {
  final UserRemoteSource remote;
  final UserLocalSource local;
  UserRepositoryImpl(this.remote, this.local);

  @override
  Future<User> getUser(String id) async {
```

```
    final cachedUser = local.cached(id);
    if (cachedUser != null) return cachedUser;        // cache hit
    final json = await remote.fetch(id);              // network
    final user = User(json['id'] as String, json['name'] as String); // map DTO->entity
    local.cache(user);                                // update cache
    return user;
  }

  @override
  Future<List<User>> getAll() async => [await getUser('1')];
}

// Fake for tests — no network/DB
class FakeUserRepository implements UserRepository {
  @override
  Future<User> getUser(String id) async => User(id, 'Test');
  @override
  Future<List<User>> getAll() async => [const User('1', 'Test')];
}

Future<void> main() async {
  final UserRepository repo = FakeUserRepository(); // swap real impl in prod
  print((await repo.getUser('1')).name); // Test
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
Returning DTOs/ `Map` /HTTP types from the repo	Leaks infra into domain	Return domain entities
Calling HTTP/DB directly from UI	Coupling, untestable	Go through the repository
Business logic in the repository	SRP blur	Keep repo about data access; logic in domain/use cases
Repo interface depending on Dio/sqflite types	Recouples to infra	Interface in domain types only
Unbounded in-repo cache	Memory leak	Bound/evict cache

Best Practices

- Define the interface in **domain terms** (entities, not DTOs); return `Future` / `Stream` / `Result`.
- Keep the **interface in the domain**, implementations in the data layer (DIP).

- Map DTO↔entity and translate errors **inside** the repository (02 · json, Module 38).
- Inject data sources; make the whole thing fakeable for tests.
- Put caching/offline strategy here, bounded.

Performance

Centralized caching improves latency; bound caches; offload heavy parsing to isolates (02 · isolates).

Advantages / Disadvantages

- **+** Decouples domain from infra, single place for caching/offline/error-mapping, highly testable, swappable sources.
- **−** Extra layer/boilerplate; risk of anemic pass-through repos or logic creep.

Interview Questions

1. 🟢 **What is the Repository pattern?** — An abstraction over data access exposing a domain-oriented API and hiding the data source(s).
2. 🟢 **Why not call HTTP/DB directly from the UI?** — It couples business logic to infrastructure, scatters caching/error handling, and makes testing require real backends.
3. 🟡 **What should a repository return?** — Domain entities (not DTOs/HTTP types), via `Future` / `Stream` / `Result`.
4. 🟡 **Where do the interface and implementation live (Clean Architecture)?** — Interface in the domain layer; implementation in the data layer (DIP).
5. 🟡 **What belongs inside a repository?** — Source orchestration, DTO↔entity mapping, caching/offline strategy, error translation — not business rules.
6. 🔴 **How does Repository enable testing?** — UI/use cases depend on the interface; you inject a fake repository, testing logic without network/DB.
7. 🔴 **Repository vs DAO?** — DAO is a lower-level, table/query-oriented data access object; Repository is higher-level and domain-oriented (may combine multiple DAOs/sources).

Senior Engineer Tips

- Keep repositories **thin orchestrators**: mapping + caching + source coordination, no domain rules (those go in use cases/entities).
- Separate **DTOs** (data layer) from **entities** (domain); map at the repo boundary so API changes don't ripple inward.
- Return `Result`/typed failures rather than throwing for expected errors — makes callers explicit and testable.

Architect Perspective

Repository is the boundary between domain and data in layered/clean architecture — the linchpin of testability, offline-first, and multi-source strategies. It localizes all the messy realities of data (network flakiness, caching, formats) so the domain stays pure and the UI stays simple (Modules 40, 19, 16).

Summary

- Repository exposes a domain-oriented data API and hides sources (network/DB/cache).
- Interface in domain (entities), impl in data layer; map DTOs, cache, translate errors inside.
- Enables testing (inject fakes), offline-first, and source swaps; keep it thin.

Revision Notes

- Repository = domain-facing data-access abstraction; hides sources.
- Return entities (not DTOs); interface in domain, impl in data (DIP).
- Map DTO↔entity + cache + error-translate inside; inject sources; fakeable.
- Repo (domain-level) vs DAO (query-level).

Practice Questions

1. Why should a repository return entities, not DTOs?
2. Where do the interface and implementation live in Clean Architecture?
3. What belongs in a repository vs a use case?

Coding Questions

1. Add offline fallback (local DB) to `UserRepositoryImpl` when remote fails.
2. Introduce DTO↔entity mapping and a `Result` return type.
3. Write tests for a use case using a `FakeUserRepository`.

Mini Project

Users data layer (pure Dart): Define `UserRepository` (domain), `UserRepositoryImpl` orchestrating remote + local fakes with caching and `Result` returns + DTO↔entity mapping, and a `GetUser` use case depending only on the interface. Test with fakes (no real IO). Acceptance: no DTO/infra leakage; caching bounded; swappable/testable; `dart analyze` clean.

Dependency Injection Pattern

> Dependency Injection (DI) supplies an object's dependencies from the outside instead of having it create them — the practical technique that realizes the Dependency Inversion Principle.

Introduction

DI means a class **receives** its collaborators (via constructor, setter, or a container) rather than constructing them internally. This file covers the DI *pattern* and forms; the full Flutter tooling (get_it, Provider, Riverpod, InheritedWidget) is Module 14.

Why this concept exists

Hard-wired dependencies (`final repo = FirebaseRepo()`) make code untestable and rigid (Module 04 · DIP). DI decouples "what a class needs" from "who provides it," enabling substitution (real vs fake), single wiring point, and clear dependency graphs.

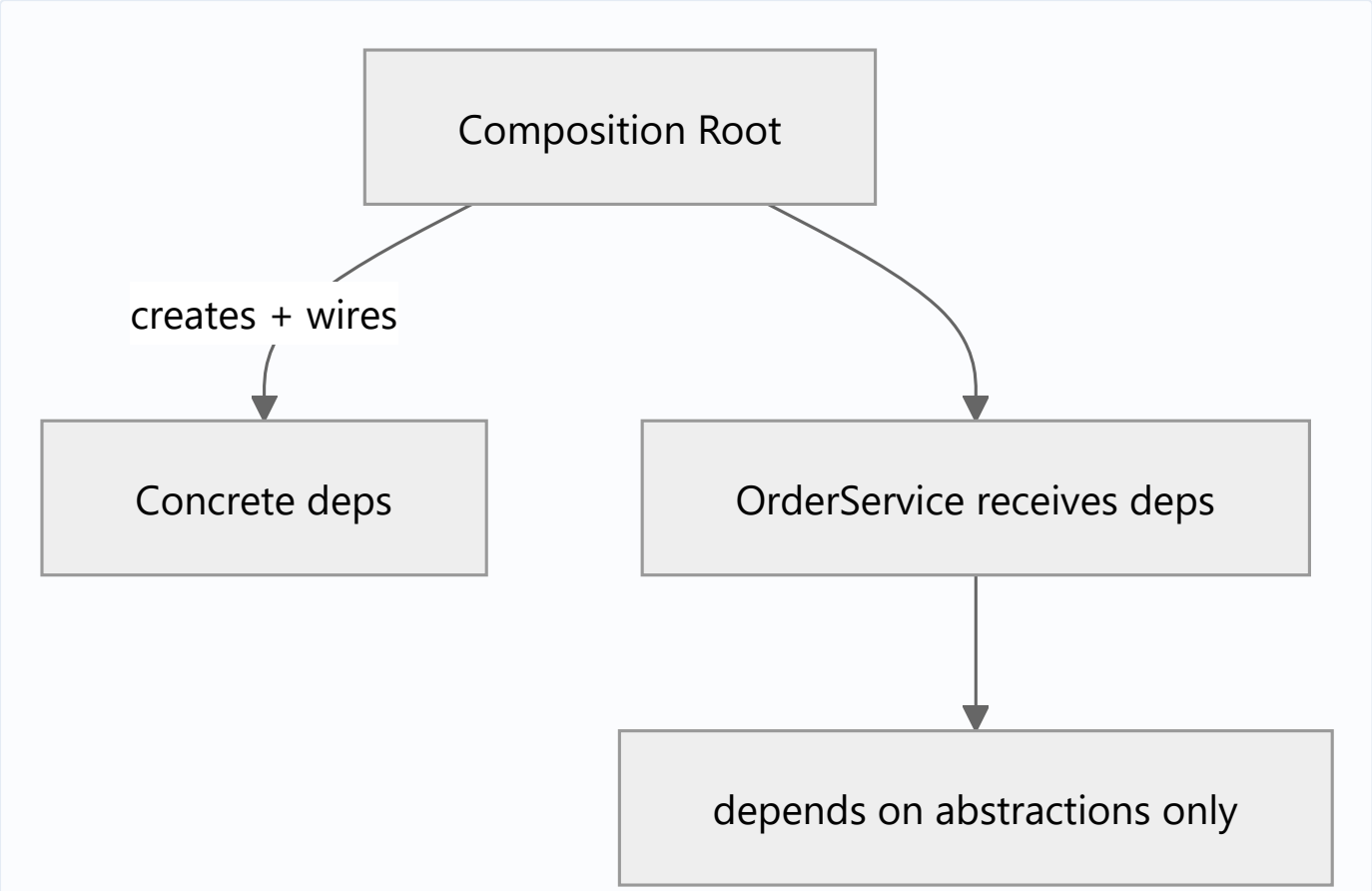
Real-world analogy

A **coffee machine that takes water from a plumbed-in supply** vs one you must manually fill from a specific well. Injected water (dependency) can come from any source (tap, bottle, test supply). The machine doesn't care where water comes from — it just receives it.

Problem Statement

An `OrderService` needs a repository, an email sender, and a logger. Constructing them inside makes it untestable. You'll inject abstractions via the constructor and wire concretes at a single composition root.

Internal Working



- **Forms of DI:**
  - **Constructor injection** (preferred): dependencies passed to the constructor — explicit, immutable, testable.
  - **Setter/property injection:** set after construction (for optional deps).
  - **Interface/method injection:** pass per call.
- **Composition root:** the single place (app startup) where the object graph is assembled.
- **DI container / service locator:** automates registration/resolution (get_it), but overuse hides dependencies — prefer constructor injection.

Memory Representation

Container-managed singletons live for app lifetime; factory registrations create per-request instances. Mind retained singletons (02 · memory_and_gc).

Compiler Behavior / Runtime Behavior

Depending on abstractions gives compile-time-safe substitution; the container resolves concretes at runtime.

Flutter Engine Behavior

Not applicable directly. (Flutter provides DI via `InheritedWidget` / `Provider` / `Riverpod` down the widget tree — Module 14.)

Dart VM Behavior

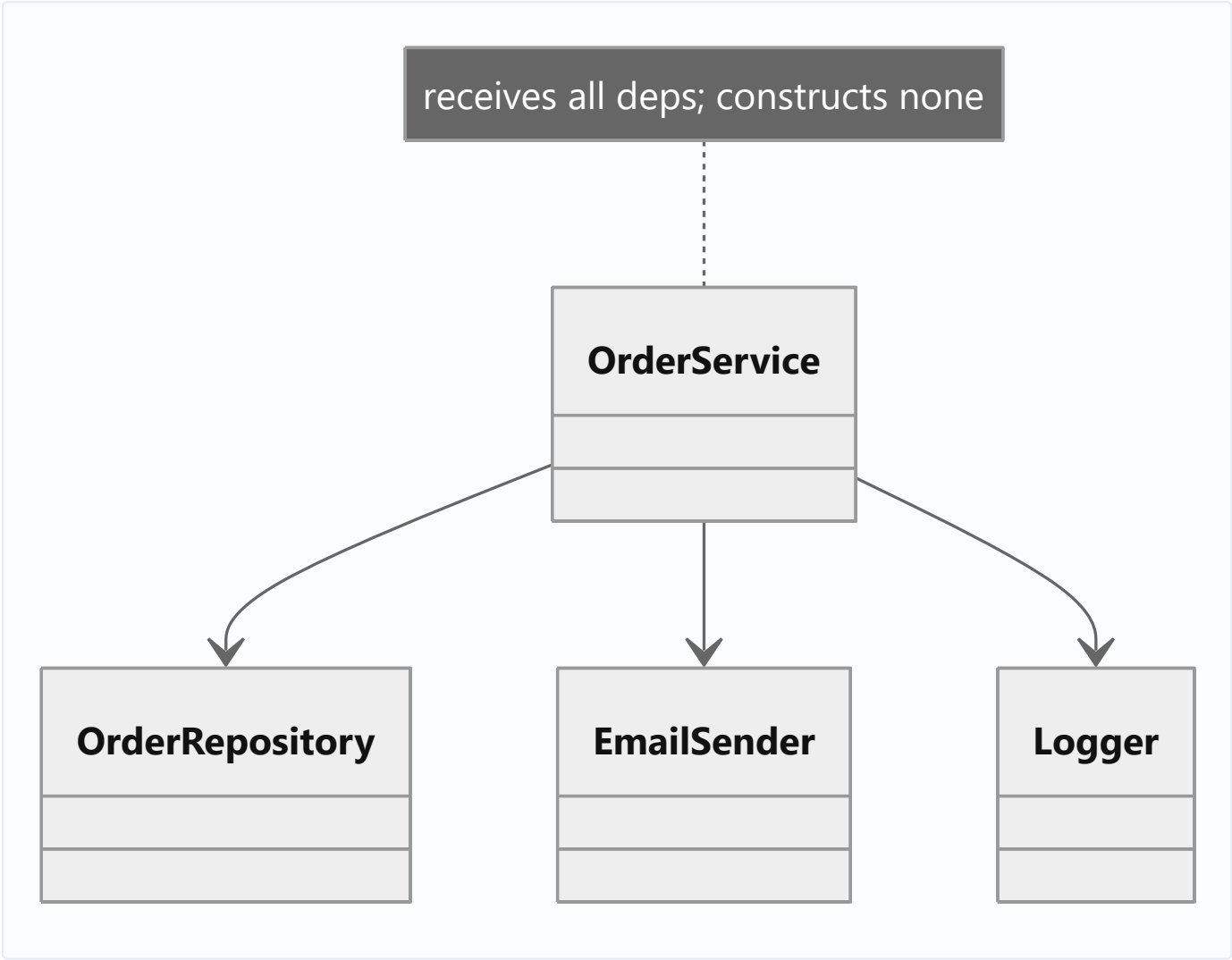
Not applicable.

Examples

```
// Abstractions
abstract interface class OrderRepository { Future<void> save(String o); }
abstract interface class EmailSender { Future<void> send(String to); }
abstract interface class Logger { void log(String m); }

// Consumer receives dependencies (constructor injection)
class OrderService {
  final OrderRepository repo;
  final EmailSender email;
  final Logger logger;
  OrderService({required this.repo, required this.email, required this.logger});

  Future<void> place(String order, String customer) async {
    logger.log('placing $order');
    await repo.save(order);
    await email.send(customer);
  }
}

// Concrete implementations
class SqlRepo implements OrderRepository {
  @override
  Future<void> save(String o) async => print('SQL save $o');
}
class SmtpEmail implements EmailSender {
  @override
  Future<void> send(String to) async => print('email $to');
}
class ConsoleLogger implements Logger {
```

```
  @override
  void log(String m) => print('[LOG] $m');
}

// Tiny manual DI container (composition root)
class Injector {
  final _singletons = <Type, Object>{};
  void register<T extends Object>(T instance) => _singletons[T] = instance;
  T get<T>() => _singletons[T] as T;
}

Future<void> main() async {
  // Composition root: wire the graph once
  final injector = Injector()
    ..register<OrderRepository>(SqlRepo())
    ..register<EmailSender>(SmtpEmail())
    ..register<Logger>(ConsoleLogger());

  final service = OrderService(
    repo: injector.get<OrderRepository>(),
    email: injector.get<EmailSender>(),
    logger: injector.get<Logger>(),
  );
  await service.place('o1', 'ada@x.com');

  // In tests: inject fakes directly, no container needed
  // OrderService(repo: FakeRepo(), email: FakeEmail(), logger: FakeLogger());
}
```

Diagrams



Common Mistakes

Mistake	Why	Fix
`new` ing dependencies inside a class	Untestable, coupled	Inject via constructor
Service locator everywhere	Hidden deps, global state	Prefer constructor injection; locate at edges
Injecting concrete types	Recouples to impl	Depend on abstractions
Over-injecting trivial values	Ceremony	Inject volatile/external deps, not pure helpers
Wiring scattered across the app	Hard to reason about	One composition root

Best Practices

- Prefer **constructor injection** of **abstractions**; make deps `final`.
- Wire the graph at a **single composition root**.
- Use a DI container to *reduce wiring boilerplate*, not to hide dependencies.

- Inject **volatile/external** dependencies (repos, clients, clock); don't inject trivial pure values.
- Register single instances via the container instead of hard singletons (03_singleton.md).

Performance

Negligible; container lookups are cheap. Singleton lifetimes save re-construction.

Advantages / Disadvantages

- **+** Testable (inject fakes), swappable, explicit dependency graph, single wiring point, enables DIP/OCP.
- **−** Wiring/boilerplate; service-locator overuse hides deps; lifetime management needs care.

Interview Questions

1. 🟢 **What is Dependency Injection?** — Supplying a class's dependencies from outside (constructor/setter/container) instead of creating them internally.
2. 🟢 **DI vs DIP?** — DIP is the principle (depend on abstractions); DI is the technique that provides those abstractions.
3. 🟡 **Which DI form is preferred and why?** — Constructor injection: explicit, immutable, guarantees deps at construction, easiest to test.
4. 🟡 **Constructor injection vs service locator?** — Constructor injection makes dependencies explicit; a service locator is a global lookup that can hide them — prefer the former, use locators at edges.
5. 🟡 **What is a composition root?** — The single place (startup) where concrete implementations are wired to abstractions.
6. 🔴 **What should you inject vs not?** — Inject volatile/external dependencies (repos, HTTP clients, clock, config); don't inject trivial pure values/helpers.
7. 🔴 **How does DI enable testing and swapping?** — You pass fakes/alternate implementations without changing the consumer, since it depends only on abstractions.

Senior Engineer Tips

- Default to constructor injection; reach for a container (get_it/Riverpod) to cut boilerplate, not to smuggle globals.
- Keep the composition root thin and explicit — it's the one place allowed to know concretes.
- Inject a `Clock` / `Random` / `Uuid` abstraction so time/randomness are testable (a frequent oversight).

Architect Perspective

DI is the practical backbone of testable, modular architecture: it realizes DIP, enables Clean Architecture's wiring at the edge, supports multi-flavor builds (dev/prod/mock), and keeps the dependency graph explicit. The choice of DI mechanism (manual, get_it, Riverpod) is a key architectural decision covered fully in Module 14.

Summary

- DI supplies dependencies from outside (prefer constructor injection of abstractions).
- Wire at a single composition root; use containers to reduce boilerplate, not hide deps.
- Realizes DIP; enables testing, swapping, and modularity; inject volatile deps only.

Revision Notes

- DI = provide deps externally; DIP = depend on abstractions (DI implements it).
- Prefer constructor injection (`final` abstractions); single composition root.
- Container reduces boilerplate; service locator can hide deps (use sparingly).
- Inject volatile/external deps (repo/client/clock), not trivial values.

Practice Questions

1. Why is `OrderService` with injected deps testable while a `new` -ing version isn't?
2. Constructor injection vs service locator — tradeoffs?
3. What belongs in the composition root?

Coding Questions

1. Refactor a class that `new` s a `Clock` / `HttpClient` to inject abstractions.
2. Build a tiny DI container with singleton + factory registrations.
3. Inject a fake `Clock` to make a time-based function deterministic in tests.

Mini Project

DI-wired order service (pure Dart): Build `OrderService` depending on `OrderRepository` / `EmailSender` / `Logger` abstractions, a small `Injector` container, and a composition root wiring real impls; then a test wiring fakes. Acceptance: no `new` inside `OrderService` ; all deps abstract + injected; single composition root; tests use fakes; `dart analyze` clean. (Full Flutter DI tooling continues in Module 14.)