OOP & Design Patterns

1. Inheritance, composition & mixins

composition over inheritance — inheritance couples a subclass to the parent's internals and its every future change, and it is single-slot — one hierarchy per class; composition is many slots and a narrow contract. A middle hierarchy level that exists only to share two helpers should be a plain function or a collaborator held as a field

the "is-a" test lies — Square extends Rectangle passes "is-a" in English and breaks in code — set the width, the height changes, and every caller written against Rectangle is now wrong (this is LSP's standard example)

mixins in TS — the (Base) => class extends Base {...} shape — multiple inheritance's use case without the hierarchy; the honest answer to "what does it buy over an interface plus a field" is: not much, most of the time

abstract class vs interface — an interface is erased and satisfied structurally by any object literal; an abstract class ships shared implementation and forces extends — pick the interface unless there is real shared code

protected is a coupling — protected is public API to every subclass forever — the mechanism by which inheritance hierarchies calcify

template method — a base class fixes the algorithm's steps and subclasses fill the holes — pass the holes in as functions and the base class becomes one higher-order function

2. SOLID

S — single responsibility — one REASON to change, not "does one thing" — one axis of change, one owner; the trap is splitting by noun (three anemic classes) instead of by reason-to-change

O — open/closed — extendable without editing — in TS the honest version is a registry, a strategy map or a union, not an abstract-class hierarchy; OCP is also the letter most often used to justify premature abstraction

L — Liskov substitution — a subtype must be usable everywhere the supertype is: no strengthened preconditions, no weakened postconditions, no new exceptions — and TS's method-parameter bivariance lets the compiler accept a violation

I — interface segregation — a caller should not depend on methods it does not use — structural typing makes it cheap: declare the two-method interface at the point of use and any bigger object satisfies it with no changes at the definition site

D — dependency inversion — both sides depend on an abstraction, and the abstraction is OWNED BY THE CONSUMER, not the implementation — that ownership rule is the part people miss; in TS the abstraction is often just a function type

SOLID's own limits — the letters were written for 1990s class-heavy OO; several collapse into "use a function" in a language with closures, and citing them as a reason to add layers is a smell — a senior answer includes this

3. Encapsulation & construction

private vs #field — private is compile-time only — erased, visible on the object, reachable from JS or through a cast; # is runtime-enforced hard private, and it breaks structuredClone-style copying and JSON round-trips. # when untrusted code shares the object, private otherwise

constructor vs static factory — a constructor cannot be named, cannot be async, cannot return a cached instance, and cannot fail cleanly; a named static (User.fromRow, Client.connect) can do all four — an object that must await I/O before it is valid takes a private constructor plus a static async factory

parameter properties — constructor(private readonly repo: Repo) {} — TS-only sugar that declares and assigns in one place

invariants belong in the constructor — an object that can exist in an invalid state pushes validation onto every caller — "make illegal states unrepresentable", from the OO side

a class with no state — a class whose methods never touch this is a namespace — it should be exported functions or a module

a module is already a singleton — ES modules are evaluated once and cached, so a module-level const is a singleton with no getInstance and no lazy-init dance — and the pattern itself is on most people's list of mistakes: global mutable state, hidden dependency, untestable

4. Unions vs class hierarchies — the expression problem

sum type vs product type — a product holds ALL of its fields at once (a record); a sum holds exactly ONE of several alternatives (a discriminated union)

the expression problem — a union + switch makes adding an OPERATION cheap (one new function) and adding a CASE expensive (every switch changes — though never-exhaustiveness makes the compiler list them); a class hierarchy with virtual methods is the exact mirror: a new subclass is cheap, a new method touches every subclass. Pick by which direction the shape will grow

state as a union, not booleans — {status:'loading'} | {status:'ok', data} | {status:'error', error} over {loading, data?, error?} — the union makes the impossible combinations unrepresentable

should the design be exhaustive at all — an open set of cases (plugin kinds arriving at runtime) argues for the class/registry shape; a closed set argues for the union

optional field vs union member — three optional fields = eight states, most of them nonsense — same illegal-states idea, from the API-design side

5. The collapses — patterns that are a language feature

Strategy — a function parameter, or a Record<Kind, (x: T) => R> map — the class-per-strategy version buys nothing once functions are first-class

Command — a closure — () => void — plus an array if you need a queue or an undo stack; the class exists in Java because it has no closures

Observer — a Set<(e: E) => void> and a for loop, or EventTarget/EventEmitter; what it does NOT give you is the interesting half — no backpressure, no error routing, and listeners that outlive their subject (the leak)

Template method — a higher-order function taking the varying steps as callbacks

Visitor — a discriminated union plus a switch with never-exhaustiveness — Visitor exists to get double dispatch in a language without sum types, and TS has sum types

Iterator — Symbol.iterator — it is in the language

Decorator (GoF) — a function that wraps a function and returns the same signature — a logging or retry wrapper; NOT the same thing as a TS @decorator, and the name collision is worth knowing

Adapter — a function or a small object literal mapping one shape to another — survives as a concept, never needs a class

Facade — a module with a few exported functions over a messy subsystem — survives, and is usually the right call at a package boundary

Proxy — either a wrapper object, or the actual Proxy built-in for the traps

Factory method / Abstract factory — a function that returns objects / a function that returns a set of related factories — Abstract Factory is nearly always over-engineering in TS

Builder — usually an options object plus defaults; survives when there is real step-order or validation between steps, and in the fluent query-builder case where the return type narrows as you chain

Singleton — a module-level value

Composite — a recursive type — type Node = Leaf | {children: Node[]} — and a recursive function; the type IS the pattern

State — a union of states plus a transition function — a state machine as data; the class-per-state version is the one to argue against

Chain of responsibility — an array of handlers and a loop, or the middleware shape (ctx, next) => ... that every Node framework uses

Mediator — a module the parties call instead of calling each other

Memento — a snapshot — an immutable copy of state plus a stack; structural sharing is what makes it cheap

Flyweight / Bridge / Prototype — interning shared values · splitting abstraction from implementation · Object.create and the prototype chain, which in JS is the object model rather than a pattern

the meta-question — "which do you actually reach for in TS?" — Strategy, Adapter, Facade, Builder-when-it-earns-it, State-as-data, middleware chains, and a discriminated union wherever the book says Visitor

6. Object-modelling vocabulary

value object vs entity — a value object has no identity and is compared field-by-field (Money, DateRange); an entity has an id and stays the same entity across changes — and since === is reference equality, value semantics need an explicit equals or a canonical string key (why a Map keyed by an object rarely does what people expect)

anemic domain model — data classes with getters/setters and all the behaviour in a "service" — the name for it; in TS an anemic model is often FINE, since plain data plus functions is a legitimate design, and the term is only a criticism inside a class-oriented one

tell, don't ask — account.withdraw(n) over reading the balance, deciding outside, and writing it back — keeps invariants inside the object that owns them

Law of Demeter — do not reach through a chain of objects you were not handed — a.b.c.d() couples you to three shapes; ?. makes the chain safe from crashing, which HIDES the coupling rather than fixing it

coupling vs cohesion — coupling: how much one part depends on another's internals; cohesion: how much one part's contents belong together — the two words that make every other point sayable in one sentence

primitive obsession — five string parameters where two are ids and one is an email — the cheap TS fix is a branded type

7. Dependency injection & seams

constructor injection, plainly — pass collaborators in; do not new them inside, and do not import a singleton for them — no container, no decorators, no framework: that is the whole pattern in TS, and it is D applied

the dependency is a function type — type Clock = () => number beats an interface Clock { now(): number } with a class behind it — and it makes the test double () => 0

what a seam is — the place where behaviour can be replaced without editing the code under test — naming the seam is how you decide what to inject, instead of injecting everything

when a module-level import is fine — pure utilities and stable stdlib have no reason to be parameters — inject what varies by environment or by test: I/O, clocks, randomness, config

service locator — a global registry asked for dependencies at call time — the dependency vanishes from the signature and reappears as a runtime failure; the anti-pattern of this section