🧰 Design Patterns – Quick Book

A fast-access guide to the most essential software design patterns and when to use them.

🔨 What Are Design Patterns?

Design Patterns are proven, reusable solutions to common problems in software design.
They are not code, but **blueprints** for solving recurring architecture challenges in a maintainable and elegant way.

🏗️ 1. Creational Patterns

How objects are created, instantiated, and configured

Pattern	Purpose	Analogy / When to Use
Singleton	One instance globally available	Database connection, config loader
Factory	Create objects without exposing logic	`.createUser()` that returns different types
Abstract Factory	Group of factories producing related objects	UI themes: buttons, inputs, etc.
Builder	Build complex object step-by-step	Pizza builder with toppings & crust
Prototype	Clone an object	Copy-paste existing settings/configs

---

🧱 2. Structural Patterns

How classes and objects are composed to form larger structures

Pattern	Purpose	Analogy / When to Use
Adapter	Convert one interface to another	Plug adapter: fit incompatible APIs

Decorator	Add responsibilities without altering	Add toppings to a base pizza
Facade	Simplified interface over complex system	`VideoPlayer.play()` hiding 10 inner steps
Proxy	Control access with a wrapper	Virtual proxy for image loading
Composite	Treat individual & groups uniformly	File → Folder → Folder (same interface)
Bridge	Separate abstraction from implementation	Remote → works with TV, Radio, etc.
Flyweight	Share common state between many objects	Characters in a text editor

---

🔁 3. Behavioral Patterns

How objects interact and communicate with each other

Pattern	Purpose	Analogy / When to Use
Observer	Notify multiple objects on state change	Event listeners, subscriptions (e.g. Redux)
Strategy	Switch between algorithms dynamically	Sorting with different comparators
Command	Encapsulate request as object	Undo/Redo system, command queue
State	Change behavior based on internal state	Media player: play/pause/stop states
Chain of Responsibility	Pass request through handlers	Middleware, logging chain
Iterator	Sequential access to collection	`.next()` on a generator
Mediator	Central controller for object interaction	Chatroom → users talk via a mediator
Memento	Capture & restore object state	Save/restore game state
Template Method	Define skeleton, let subclasses fill steps	Abstract game loop

Visitor	Separate logic from structure traversal	Tax calculator visiting financial records
Interpreter	Interpret a language or command syntax	Regex engine, calculators

---

🧠 How to Choose a Design Pattern

You Need To...	Use This Pattern
Ensure only one object exists	Singleton
Build complex objects step-by-step	Builder
Add behavior without modifying class	Decorator
Wrap access to something expensive	Proxy
Simplify a complex API	Facade
Notify multiple components of change	Observer
Switch between algorithms easily	Strategy
Queue and execute commands	Command
Traverse tree-like structure uniformly	Composite

📌 Real-World Pattern Mapping

Real World Concept	Design Pattern
React Hooks & Context	Observer, Mediator
Redux Middleware	Chain of Responsibility
Express Middleware Stack	Chain of Responsibility
React Component Wrapping	Decorator
Command Line Parsing	Interpreter
Axios Interceptors	Proxy + Chain of Responsibility
Singleton Logger	Singleton

✅ Final Checklist Before You Use a Pattern

- Am I solving a **real recurring problem**, or overengineering?
- Is this pattern **adding clarity**, not complexity?
- Can this **improve extensibility** or maintainability?
- Does this decouple logic in a **clean and reusable** way?
- Am I familiar with **when NOT to use it**?