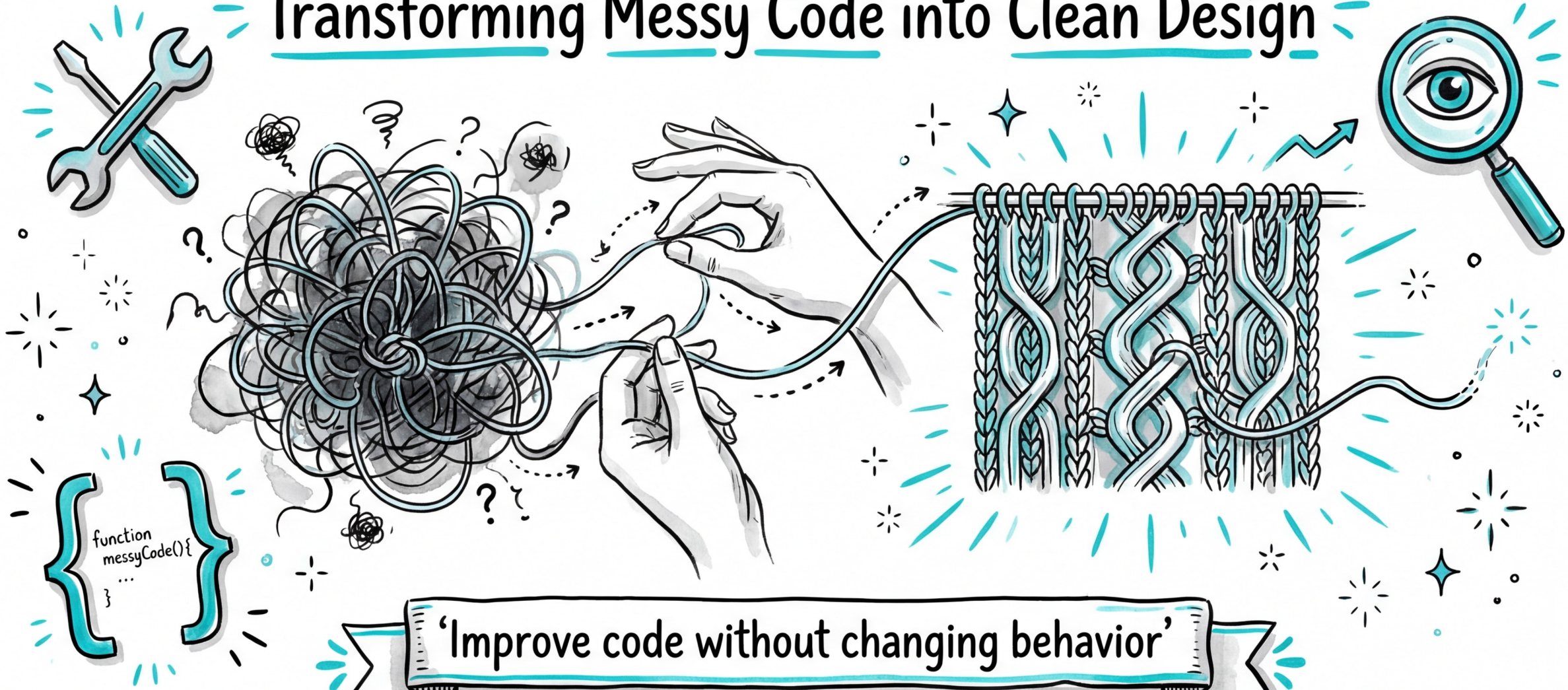
REFACTORING

Transforming Messy Code into Clean Design

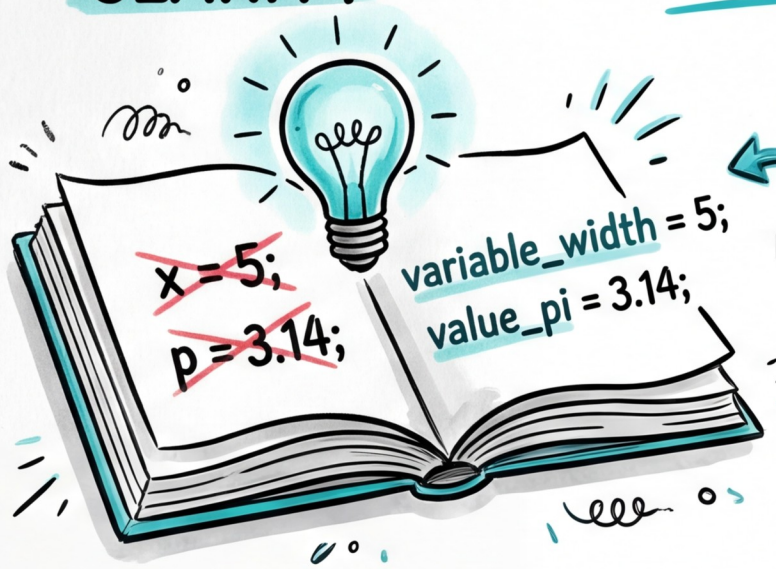


'Improve code without changing behavior'

WHAT IS REFACTORING?

Clean Code Principles

CLARITY



READABLE VARIABLE NAMES

IMPROVE STRUCTURE

EASIER MAINTENANCE

NO DUPLICATION

```
function processData(list) {
  for (let i=0; i < list.length; i++) {
    if (list[i] > 10) {
      print(list[i]);
      save(list[i]);
    }
  }
}
```

```
function processItem(item) {
  if (item > 10) {
    print(item);
    save(item);
  }
function processData(list) {
  list.forEach(processItem);
```

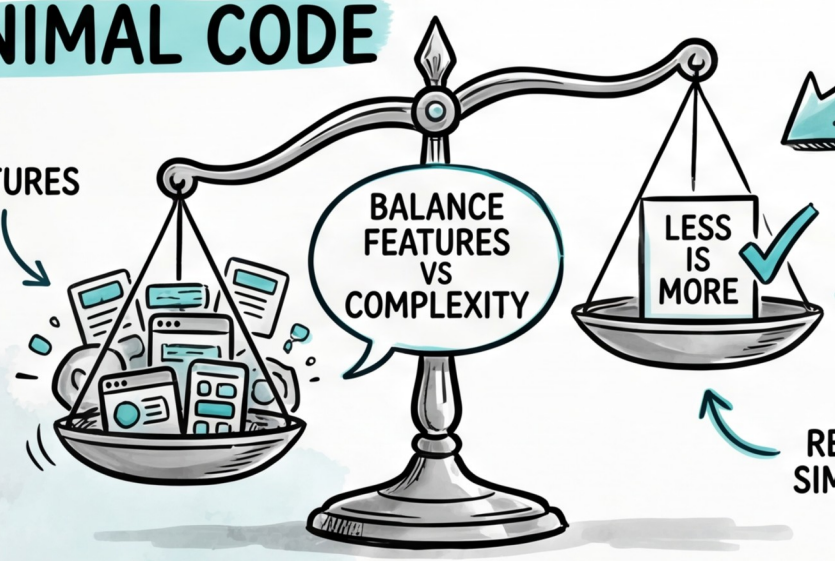
SINGLE BLOCK, DRY PRINCIPLE

BETTER COLLABORATION

Clean Code

MINIMAL CODE

FEATURES



REDUCE REDUNDANCY, SIMPLIFY LOGIC

TEST COVERAGE



PROGRESS & CONFIDENCE

TECHNICAL DEBT

The Hidden Cost of Quick Fixes

Quick & Dirty

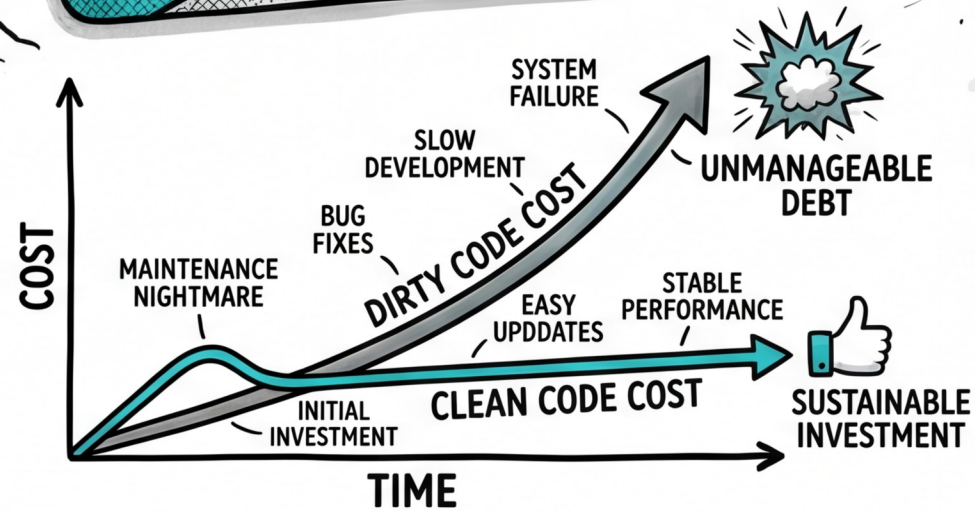
TODO
FIXME
HACK
LATER
SHORTCUTS
BYPASS
FUTURE VALUE
RISK OF COLLAPSE
CRACKING UNDER WEIGHT OF PATCHES

TECH DEBT

INTEREST RATE:
20% … 50% … 100% …
COMPOUNDING DAILY!
LIMIT REACHED!

Proper Design

CLEAN CODE
MODULAR
DOCUMENTED
TESTED
SCALABLE
REFACTORING



THE DEBT PILE-UP:

IGNORE AT YOUR OWN RISK!

TODO
FIXME
TODO
TODO
FIXME
HACK

LESSON: PAY IT DOWN! INVEST IN CLEAN CODE FOR LONG-TERM SUCCESS.

© 2024 Sketchnote Studio

WHEN TO REFACTOR

The Rule of Three

Clean code, happy life

```
function doSomething() {
  ...
}
```

SHIP IT

First time — just do it

Ugh... again?

```
function doSomethingElse() {
  // same logic...
}
```

PUSH (reluctantly)

Second time — wince but do it anyway

Improven debt

```
class Module {
  ...
}
function optimizedTask() {
  ...
}
```

Scalable future

CLEAN

DUPLICATE

MESSIER

TECHNICAL DEBT

Third time — REFACTOR!

REFACTOR & CELEBRATE!

Team efficiency

Improved maintainability

TRIGGER MOMENTS

Before review

Adding features

Fixing bugs

RECURRING REFACTORING SPRINTS

		1	2	3	4	5
6	7	8	9	REFACTORING SPRINT		
13	14	15	16	17	18	19
20	REFACTORING SPRINT					
27	28					

Schedule time for code health!

Recognize the patterns before they become problems!

Sketchnoted by *Artist*

'CODE SMELLS'

Recognizing Problems in Your Code

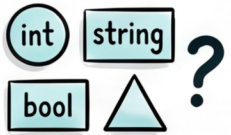
BLOATERS



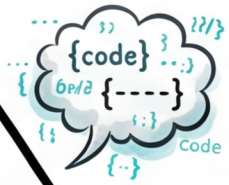
1) Long Method



2) Large Class

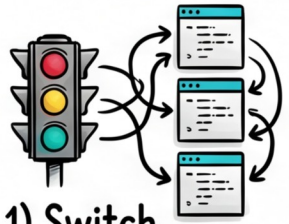


3) Primitive Obsession



OOP ABUSERS





1) Switch Statements



2) Temporary Field

CHANGE PREVENTERS

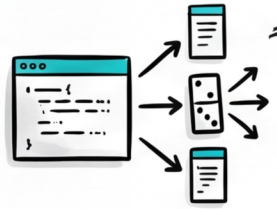


1)



Shotgun Surgery

2)

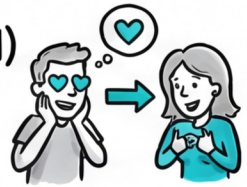


Divergent Change

COUPLERS

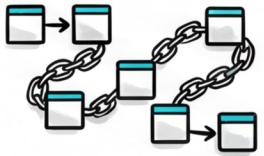


1)



Feature Envy

2)



Message Chains

3)



Middle Man

DISPENSABLES



1)



Dead Code

2)



Duplicate Code

3)



Lazy Class

2024

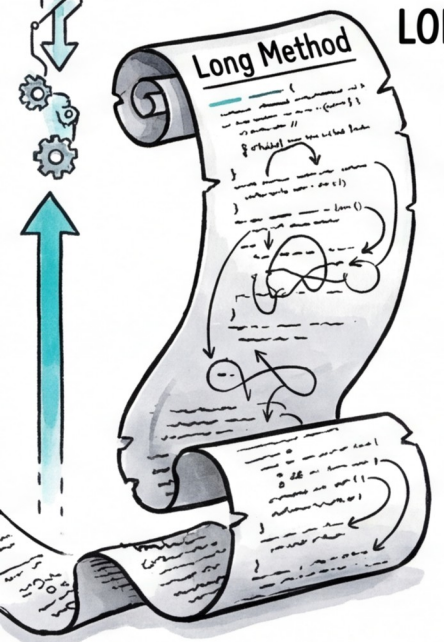
By

BLOATERS & DISPENSABLES

When Code Gets Too Big or Too Useless

BLOATERS

LONG METHOD



BAD

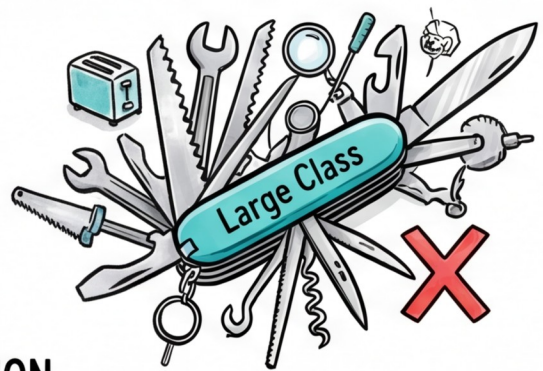
Refactored Method



GOOD!



LARGE CLASS



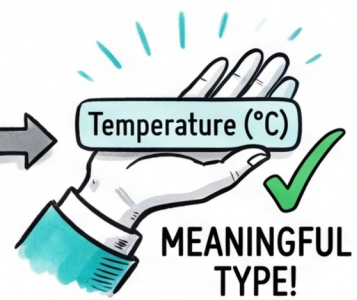


Focused Class

PRIMITIVE OBSESSION



BAD!



MEANINGFUL TYPE!

LONG PARAMETER LIST

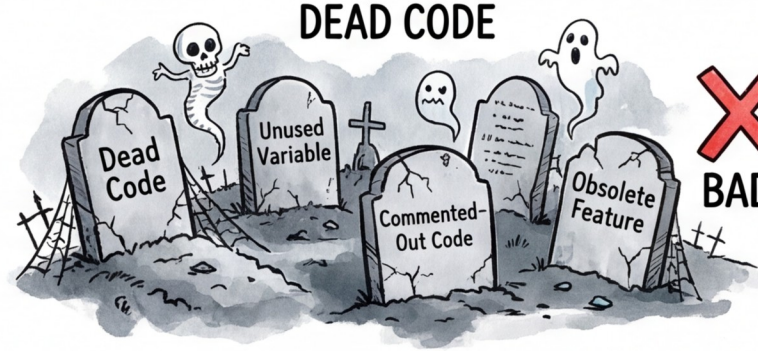




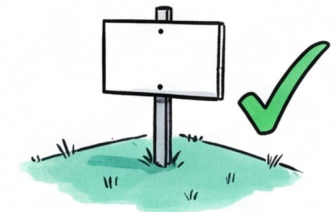
Parameter Object

DISPENSABLES

DEAD CODE



BAD!



CLEAN & REMOVED!

LAZY CLASS



BAD!



Lazy Class

Useful Class

DUPLICATE CODE

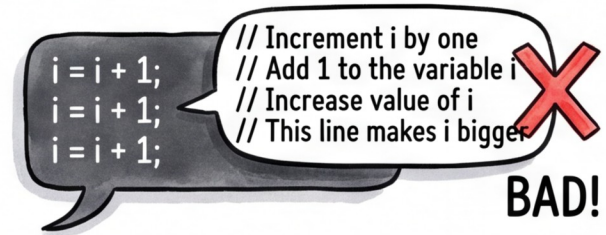


BAD!

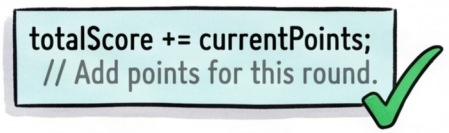


EXTRACT & REUSE!

COMMENTS EXPLAINING OBVIOUS CODE



BAD!



SELF-EXPLAINING CODE!

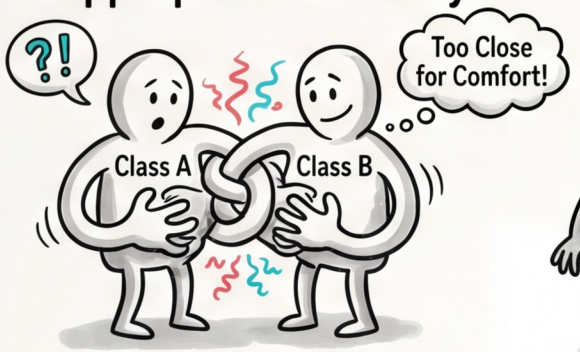
Sketchnote by Arit Nana, 2024

COUPLERS & CHANGE PREVENTERS

Dependencies That Slow You Down

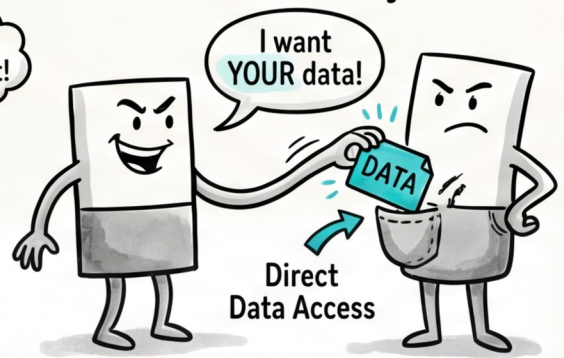
COUPLERS

Inappropriate Intimacy



Inappropriate Intimacy

Feature Envy

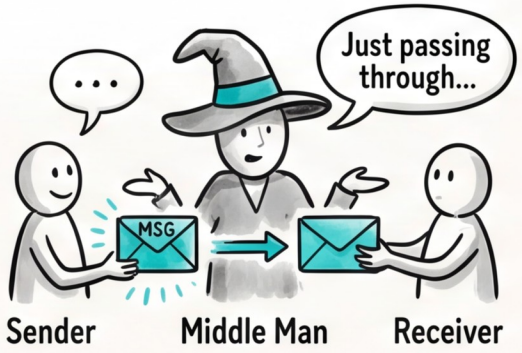


Message Chains



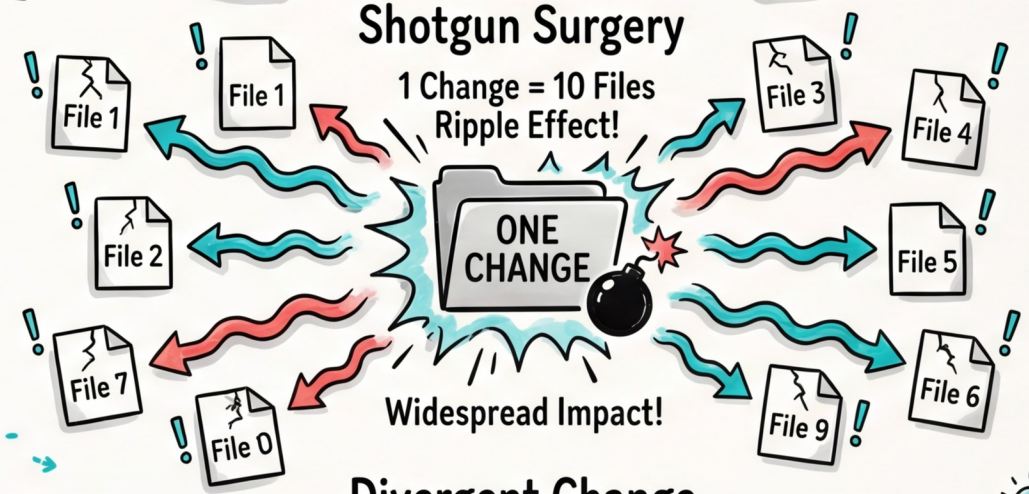
Long, Brittle Path!

Middle Man

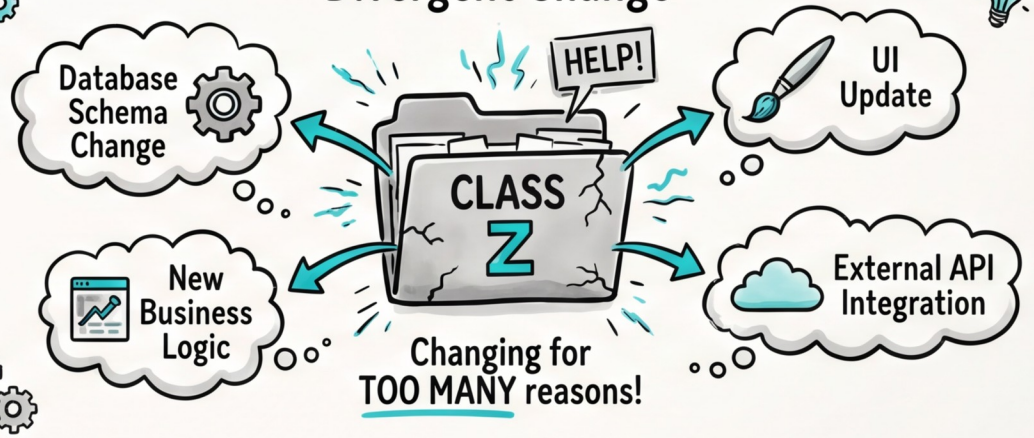


CHANGE PREVENTERS

Shotgun Surgery



Divergent Change

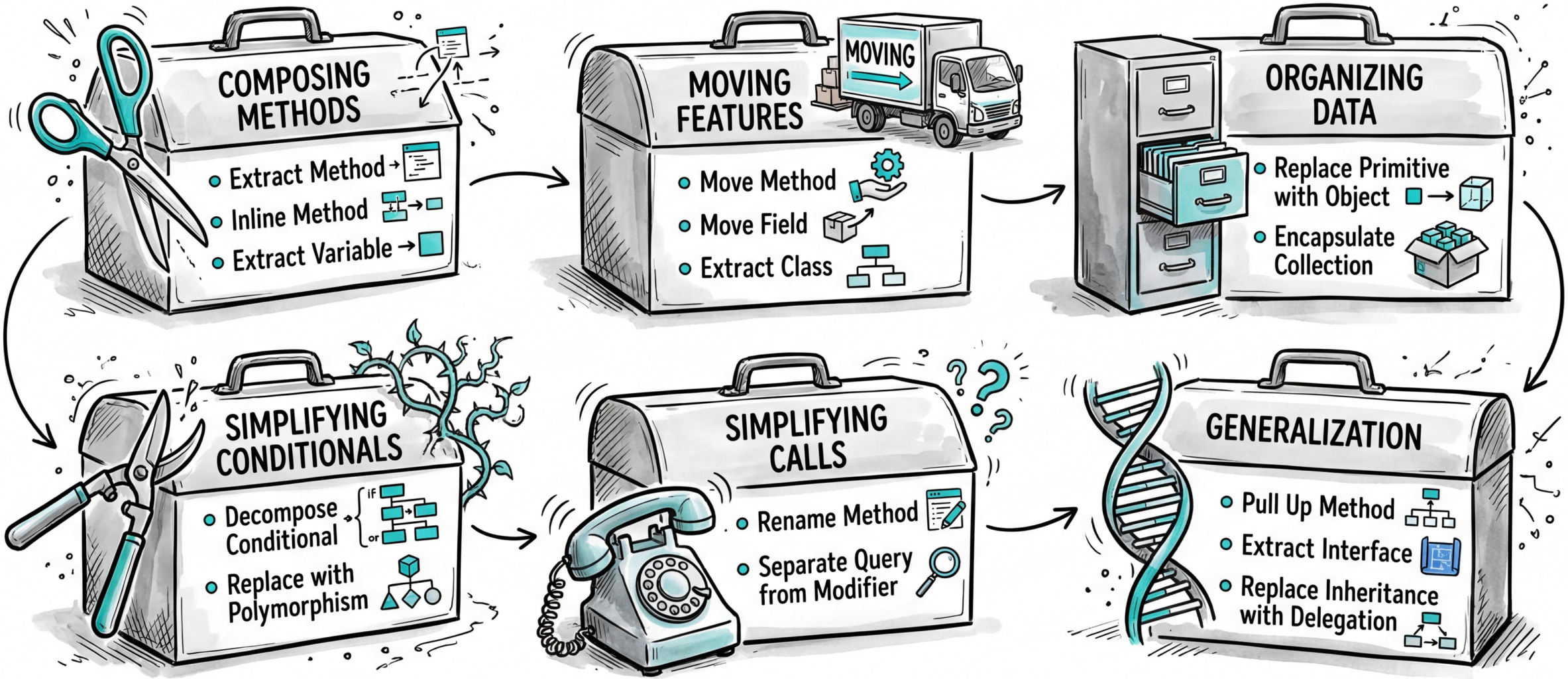


SOLUTION HINTS: Extract Class, Move Method, Hide Delegate

SOLUTION HINTS: Extract Class, Move Method, Inline Class

Refactoring Techniques

Your Toolkit for Clean Code



EXTRACT & COMPOSE

The Most Powerful Refactoring Moves

BEFORE

monolithic function

```
Function 'ProcessOrder', ~200 lines {
  Function Inpu(order) {
    // ~~~~~ ~~~~~~~~
    validate Input & Permissions);
  }
  Function calculateFinanis(order) {
    // ~~~~~ ~~~~~
    Calculate Discounts & Taxes + 10);
  }
  Function updateStock(order) {
    // ~~~~~ ~~~~
    // new update Inventory & Stock);
    return discantase;
  }
  generateInvoice(order) {
    // ~~~~~ ~~~~~
    generate Invoice & Payment ([);
  }
  send Notifications(order) {
    send Notifications & Log;
    ...
  }
}
```

Validate Input & Permissions

Calculate Discounts & Taxes

Update Inventory & Stock

Generate Invoice & Payment

Send Notifications & Log

REFACTORING

AFTER

```
Function 'ProcessOrder' (Main) {
  validateInput(order)
  calculateFinancials(order)
  updateStock(order)
  generateInvoice(order)
  sendNotifications(order)
}
```

```
Function validateInput(order)
  if (order)
  ...
}
```

```
Function calculateFinancials(order)
  if (order > 100) {
    calculatetor = " " + "
      + " " + &akd 12;
  } else {
    formal+ calculateFinancials(order);
    return total;
  }
}
```

```
Function updateStock(order)
  database updateStock(oal);
  database.updateStock();
}
```

```
Function generateInvoice(order)
  create document);
}
```

```
Function sendNotifications(order)
  send not&inations();
}
```

1) IDENTIFY code block

2) EXTRACT to new method

3) NAME it clearly

4) REPLACE original with call

EXTRACT VARIABLE

```
if (order.total > 1000 * 0.9) {
  ...
}
```

Magic Number

```
const highValueThreshold = 1000 * 0.9;
if (order.total > highValueThreshold) {
  ...
}
```

Named Constant

"Small functions that do one thing well"

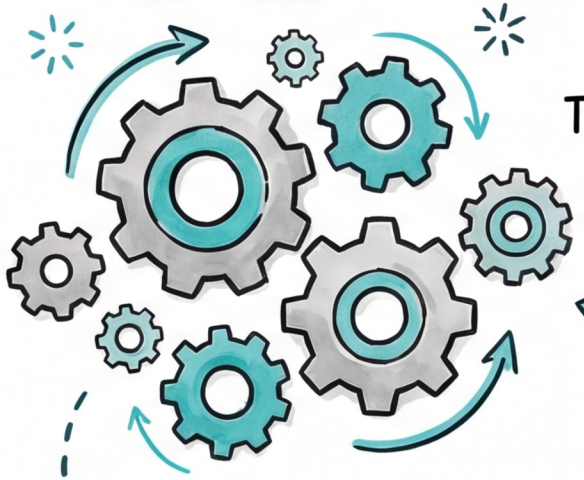
REFACTORING WORKFLOW

A Systematic Approach



DESIGN PATTERNS

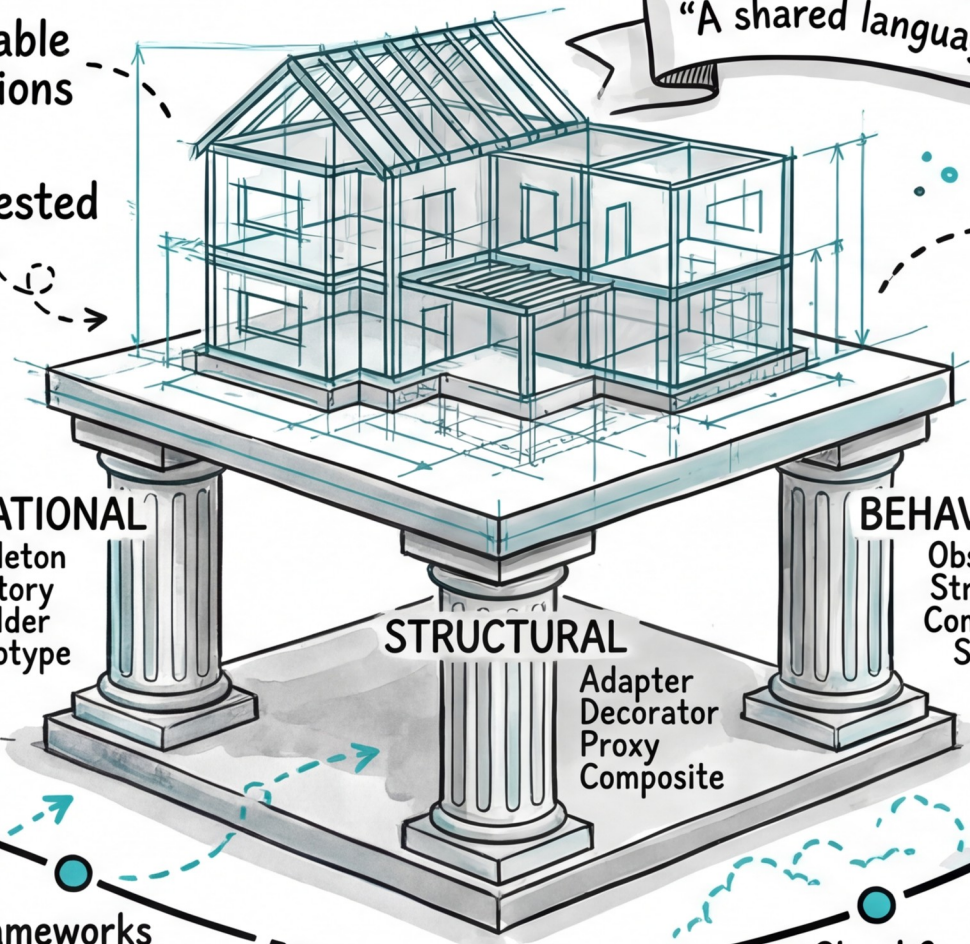
Proven Solutions to Common Problems



EFFICIENT COMPONENTS

Reusable Solutions

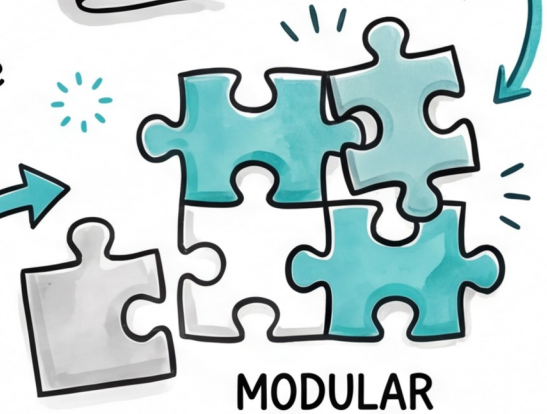
Time-Tested



"A shared language for your team"



Scalable Design

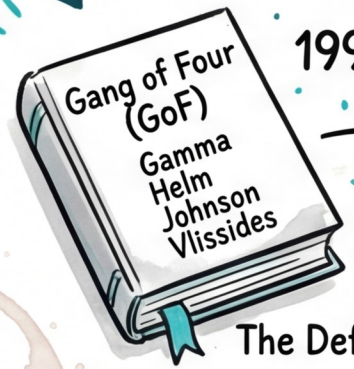


MODULAR INTEGRATION

CREATIONAL
Singleton
Factory
Builder
Prototype

STRUCTURAL
Adapter
Decorator
Proxy
Composite

BEHAVIORAL
Observer
Strategy
Command
State



1994

The Definitive Guide

Frameworks Era

Modern Web

Cloud & Microservices

TypeScript

Python

Java

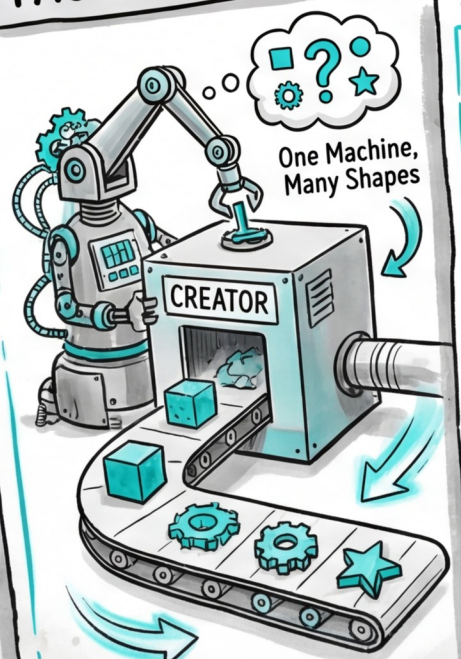
PATTERNS ARE UNIVERSAL:
Across Languages & Platforms

'CREATIONAL PATTERNS'

Smart Ways to Create Objects

CARD 1

FACTORY METHOD



Define an interface for creating objects, but let subclasses decide which class to instantiate.

CARD 2

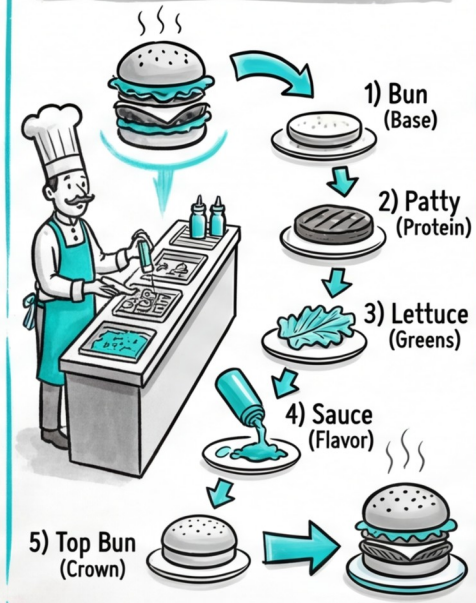
ABSTRACT FACTORY



Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

CARD 3

BUILDER



Separate the construction of a complex object from its representation so that the same construction process can create different representations.

CARD 4

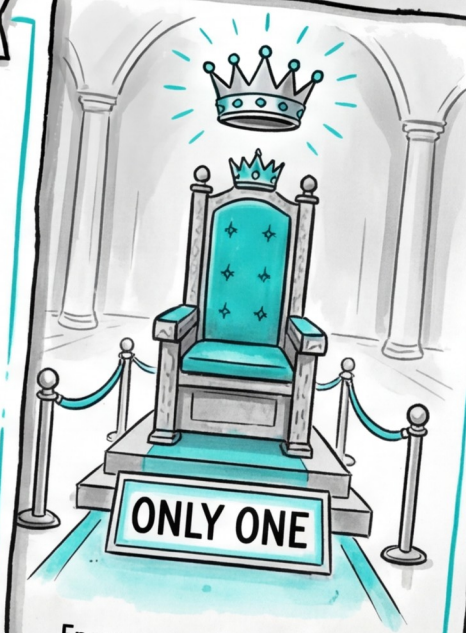
PROTOTYPE



Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

CARD 5

SINGLETON

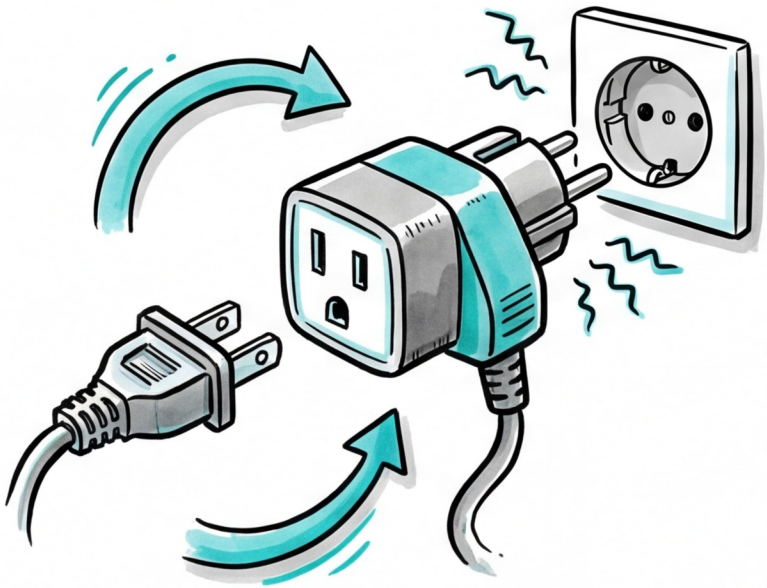


Ensure a class has only one instance and provide a global point of access to it.

STRUCTURAL PATTERNS — Part 1

Adapter, Decorator, Facade

ADAPTER



Makes **incompatible** interfaces work together

DECORATOR



Add **behavior** without changing the original

FACADE

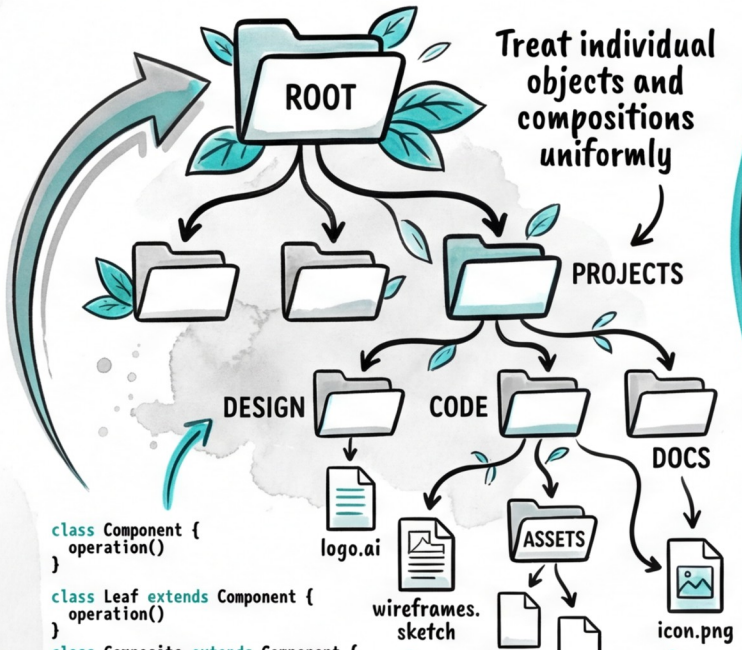


Simple interface to complex system

'Structural Patterns – Part 2'

Composite, Proxy, Bridge

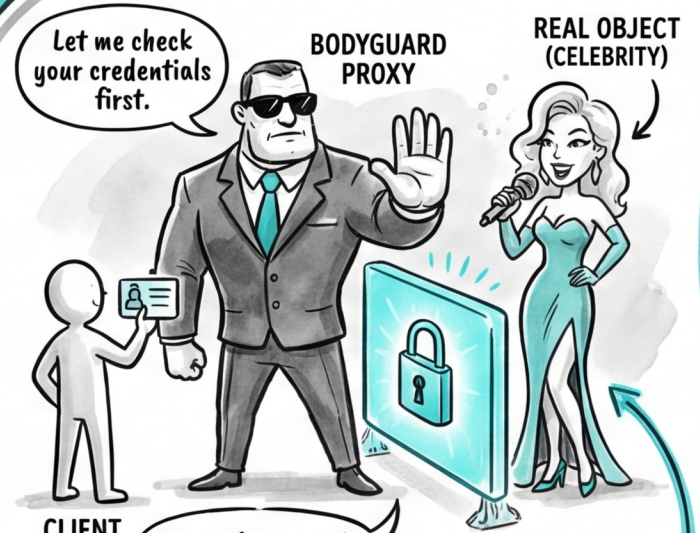
COMPOSITE




```
class Component {
  operation()
}

class Leaf extends Component {
  operation()
}
class Composite extends Component {
  children = []
  add(child)
  remove(child)
  operation() {
    // Delegates to children
    for child in children:
      child.operation()
  }
}
```

PROXY

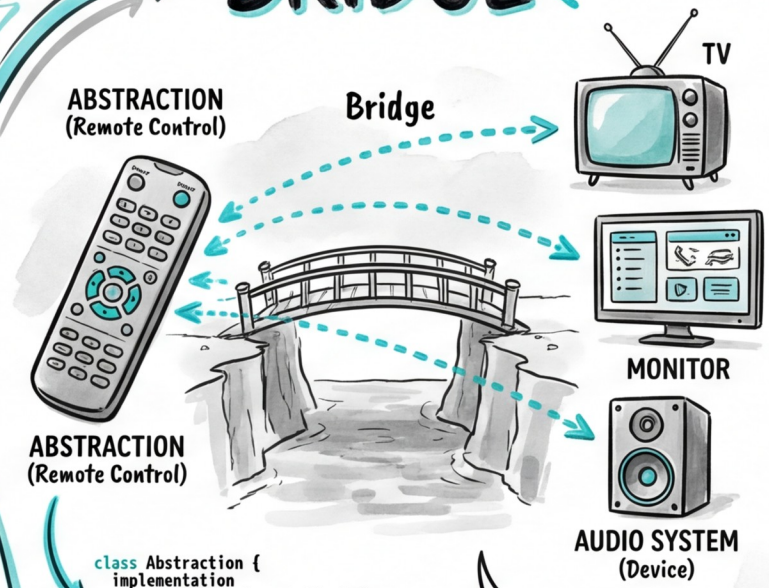



```
class Subject {
  request()
}

class RealSubject extends Subject {
  request() {
    // Performs actual work
  }
}
```

```
class Proxy extends Subject {
  realSubject
  request() {
    if (checkAccess()) {
      if (!realSubject) {
        realSubject = new RealSubject()
      realSubject.request()
      logAccess()
    }
  }
  checkAccess() { ... }
  logAccess() { ... }
}
```

BRIDGE




```
class Abstraction {
  implementation
  constructor(implementation) {
    this.implementation = implementation
  }
  operation() {
    implementation.operationImp()
} }

interface Implementation {
  operationImp()
}

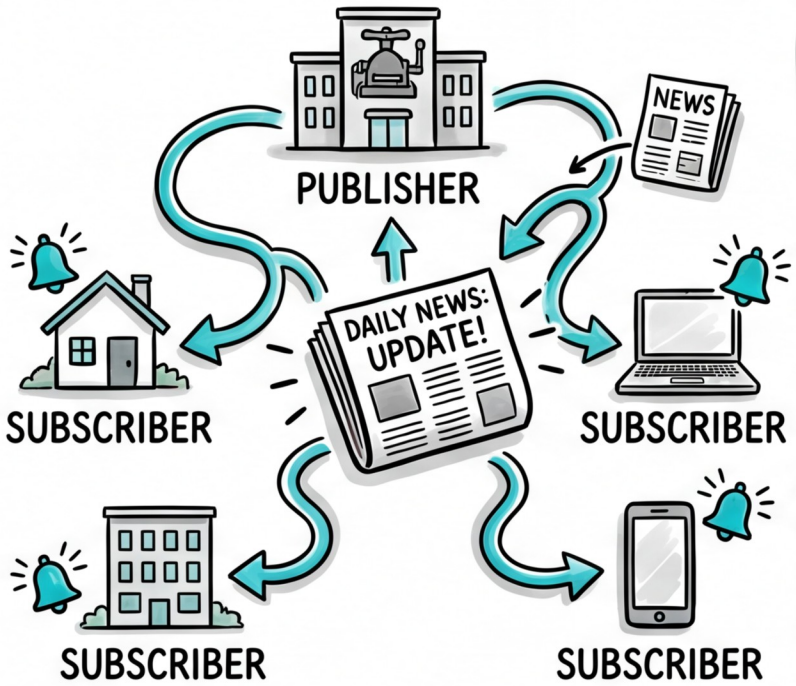
class ConcreteImplementationA implements Implementation {
  operationImp()
}
class ConcreteImplementationB implements Implementation {
  operationImp()
}
```

Separate abstraction from implementation

BEHAVIORAL PATTERNS — Part 1

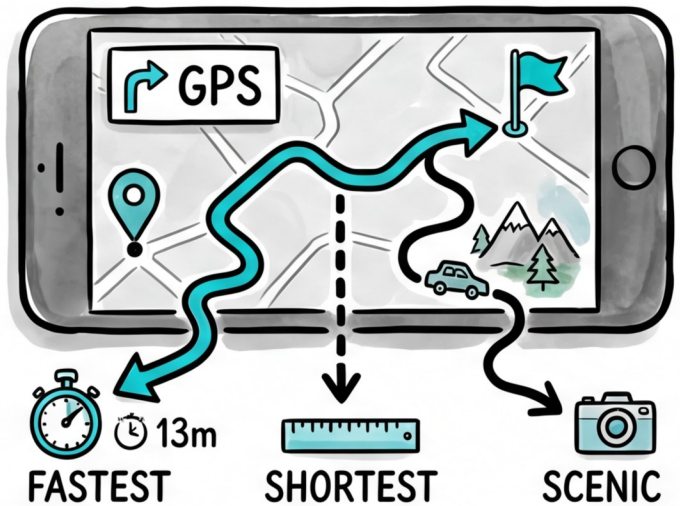
Observer, Strategy, Command

OBSERVER



"When one changes, all dependents update"

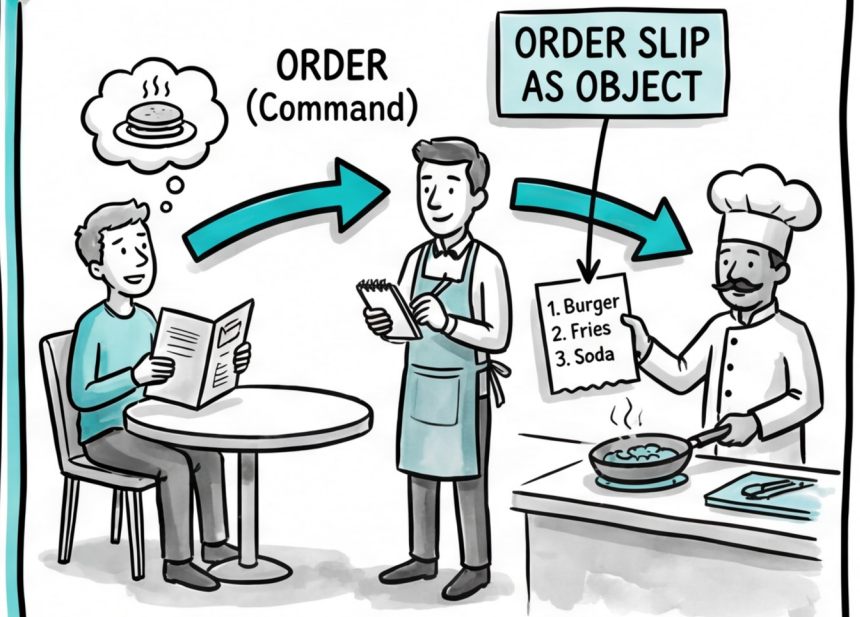
STRATEGY



Same destination different algorithms

"Swap algorithms at runtime"

COMMAND



Turn requests into objects

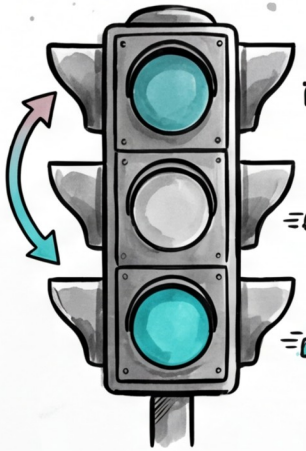
UNDO

Undo button on command

BEHAVIORAL PATTERNS — Part 2

State, Mediator, Template Method

STATE



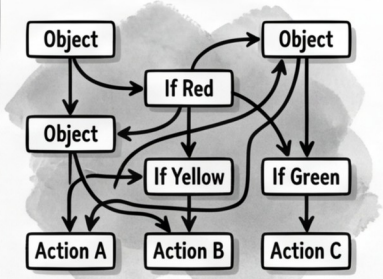
RED STATE
Behavior: Stop

YELLOW STATE
Behavior: Prepare to Stop

GREEN STATE
Behavior: Go

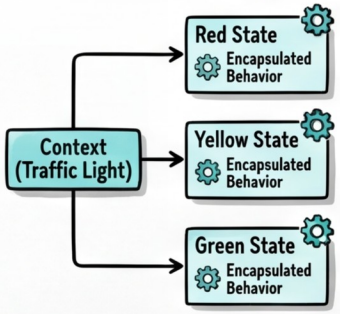
> "Object changes behavior when state changes"

BEFORE
Spaghetti Code / Conditional Chaos



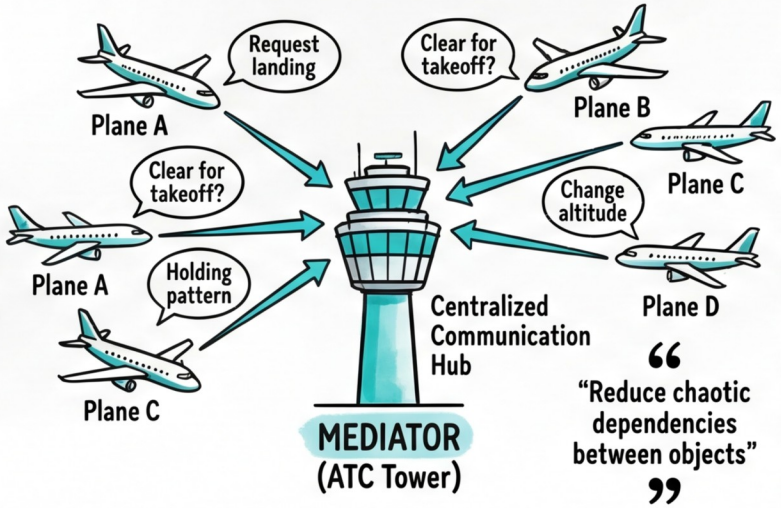
Complex, nested conditionals, hard to maintain

AFTER
Clean State-Based Logic



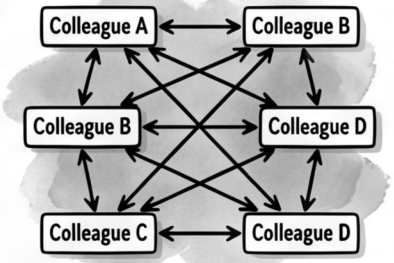
States are separate classes, easy to add/modify

MEDIATOR



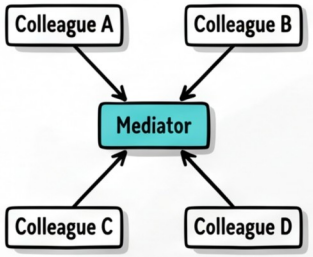
> "Reduce chaotic dependencies between objects"

BEFORE
Chaotic Dependencies / Tight Coupling



Direct, many-to-many connections, hard to change

AFTER
Decoupled via Mediator



Objects only talk to Mediator, simplified interactions

TEMPLATE METHOD

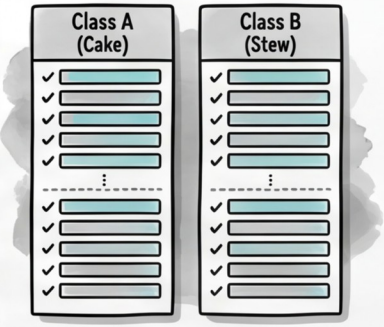


> "Define skeleton, let subclasses fill details"

Cake
Prepare: Mix flour, sugar, eggs
Serve: Frost & Slice

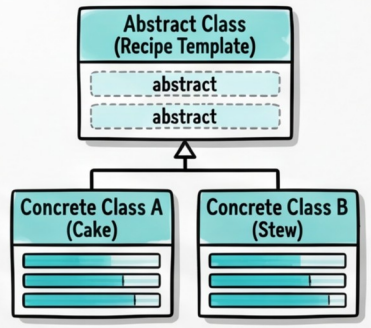
Stew
Prepare: Chop veg, sear meat
Serve: Ladle into bowls

BEFORE
Code Duplication / Rigid Structure



Repeated code, hard to update common steps

AFTER
Shared Template / Flexible Subclasses



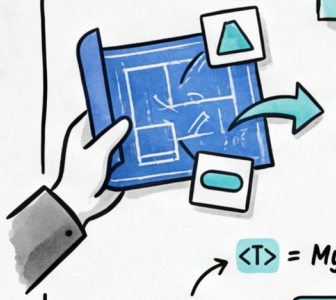
Common logic in base class, variations in subclasses

You already use patterns daily

PATTERNS IN TypeScript

Modern Implementation Examples

FACTORY



```ts
createCompoonent<T>(type: string,
                    props?: any): T
```

TS

<T> = MyComponent

<T> = AnotherComponent

Generic Type Parameter

UI BUTTON

Form Input

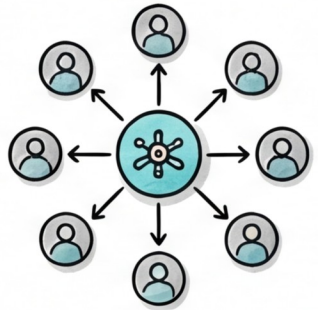
Form

Abstraction of Object Creation

Flexible Creation

OBSERVER

- One-to-Many Dependency
- Automatic Updates
- Decoupling Subjects and Observers



```ts
const [state, setState] =
  useState(initialState);
```

```ts
useEffect(() => {
  // React to changes
}, [state]);
```

TS

Subject / Observable State

React's Hook System is an Observer Implementation!

Observer / Subscriber Reacts

STRATEGY

```ts
type Middleware = (req: Request,
                   res: Response,
                   next: NextFunction)
                   => void;
```

TS

Auth Strategy

```ts
const auth: Middleware
= (req, res, next) => {
  /* authenticate */
  next();
};
```

Logger Strategy

```ts
const logger: Middleware
= (req, res, next) => {
  /* log request */
  next();
};
```

RateLimit Strategy

```ts
const ratlit: Middleware
= (req, res, next) => {
  /* limit reqs */
  next();
};
```

- Interchangeable Algorithms
- Encapsulate Behavior
- Runtime Flexibility

DECORATOR



Metadata & Behavior Injection

```ts
@Injectable()
class Service {
  constructor(private readonly dep: Dependency) {}
}
```

Target Class / Function

Dependency Injection Container uses metadata

Dependency Injection

- Attach Responsibilities Dynamically
- Meta-programming
- Clean Separation of Concerns

SINGLETON

- One-to-Many Dependency
- Automatic Updates
- Decoupling Subjects and Observers



```ts
import { PrismaClient } from "@prisma/client";

export const db = new PrismaClient();
// Shared Instance
```

TS

API Route

Background Job

User Service

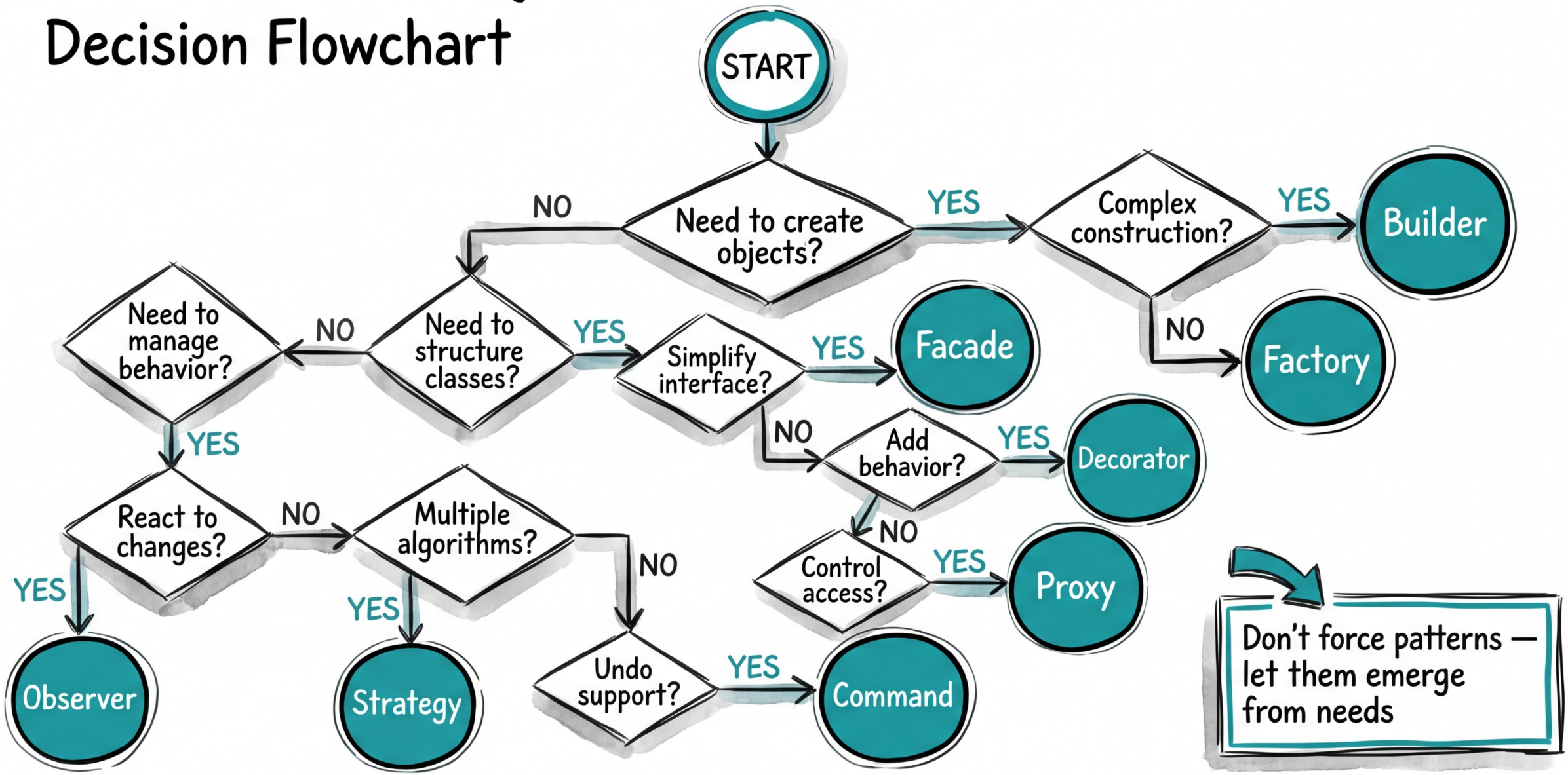
db

- Ensure Single Instance
- Global Access Point
- Resource Management

Sketchnote by [Your Name] - 2024

CHOOSING THE RIGHT PATTERN

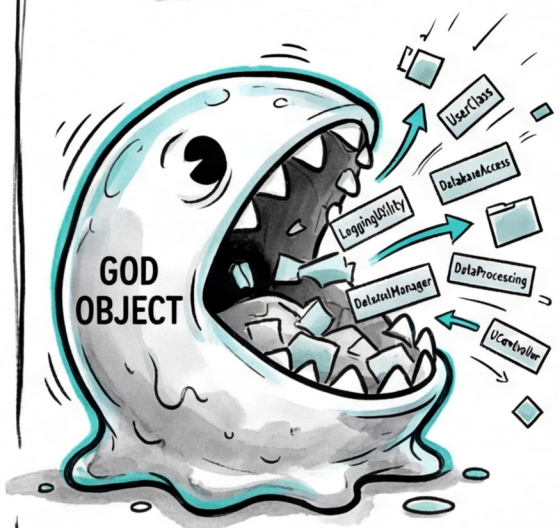
Decision Flowchart



ANTI-PATTERNS

When Patterns Go Wrong

GOD OBJECT



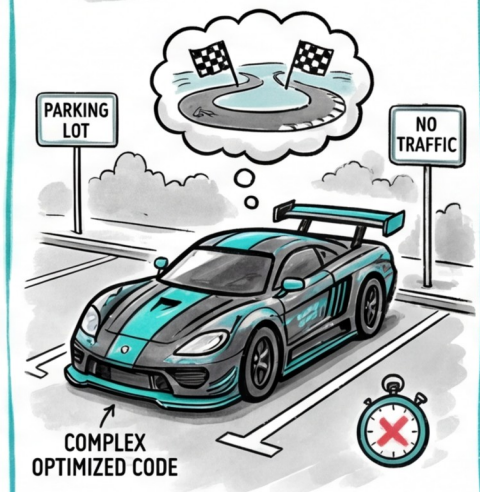
SPAGHETTI CODE



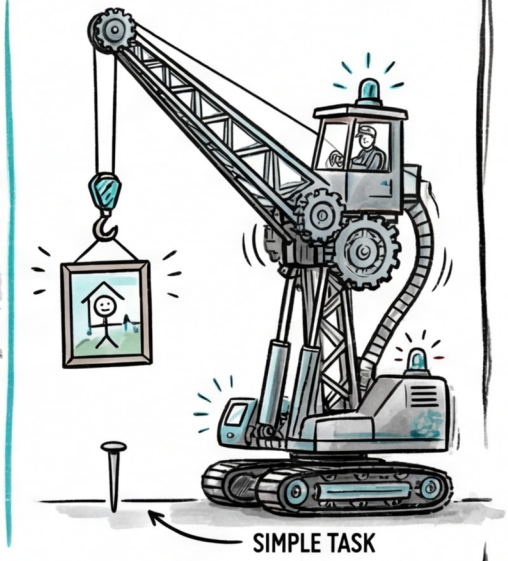
GOLDEN HAMMER



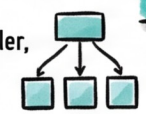
PREMATURE OPTIMIZATION



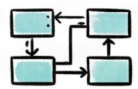
OVER-ENGINEERING



Instead, use...
SINGLE RESPONSIBILITY PRINCIPLE.
Decompose into smaller, focused classes!



Instead, use...
MODULARITY & CLEAN ARCHITECTURE.
Define clear boundaries and interfaces!



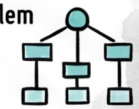
Instead, use...
THE RIGHT TOOL FOR THE JOB.
Select the best technology and pattern for the specific problem!



Instead, use...
MEASURE FIRST.
Optimize only when necessary and based on performance data!



Instead, use...
KISS (Keep It Simple, Stupid) & **YAGNI** (You Ain't Gonna Need It).
Solve the current problem without unnecessary complexity!



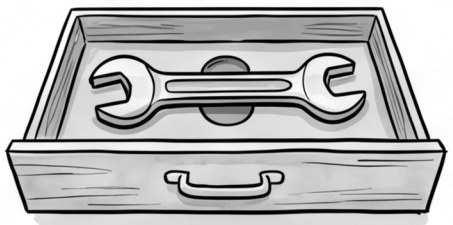
"THE SIMPLEST SOLUTION THAT WORKS"

'SOLID PRINCIPLES'

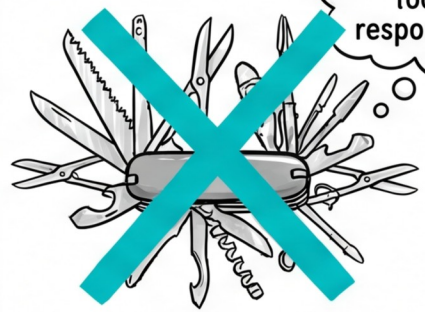
The Foundation of Good Design

S — SINGLE RESPONSIBILITY

ONE TOOL PER DRAWER



Focus on one task

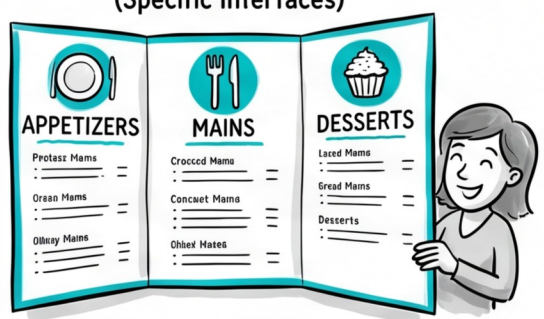


I — INTERFACE SEGREGATION

GIANT LIST
(Bloated Interface)

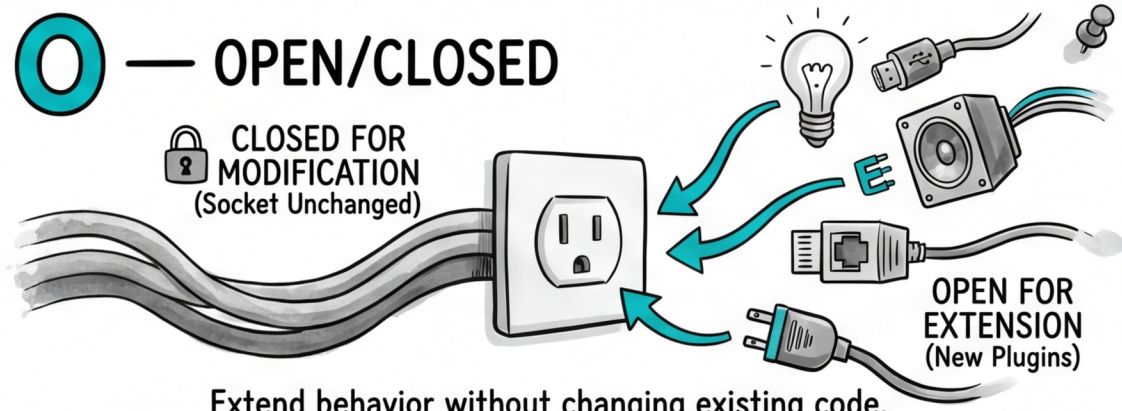


SEGREGATED MENU
(Specific Interfaces)



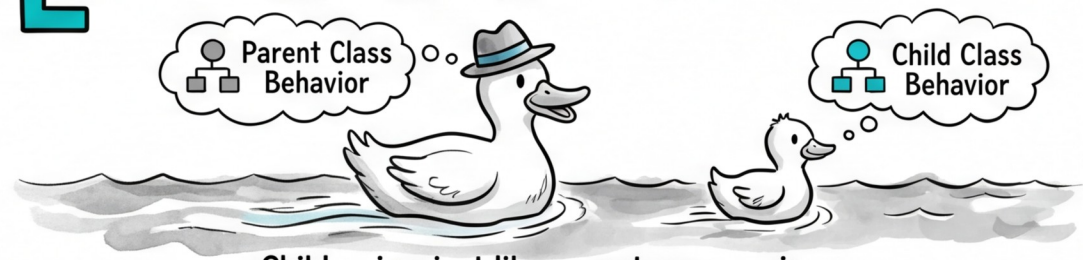
Clients shouldn't be forced to depend on interfaces they don't use.

O — OPEN/CLOSED



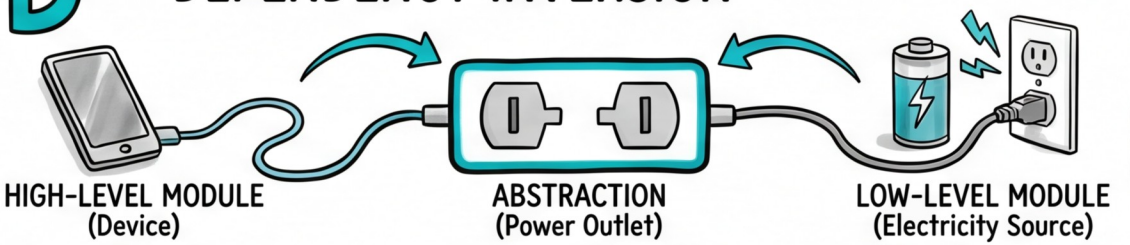
Extend behavior without changing existing code.

L — LISKOV SUBSTITUTION



Child swims just like parent, no surprises.
Subclasses should be substitutable for base classes.

D — DEPENDENCY INVERSION

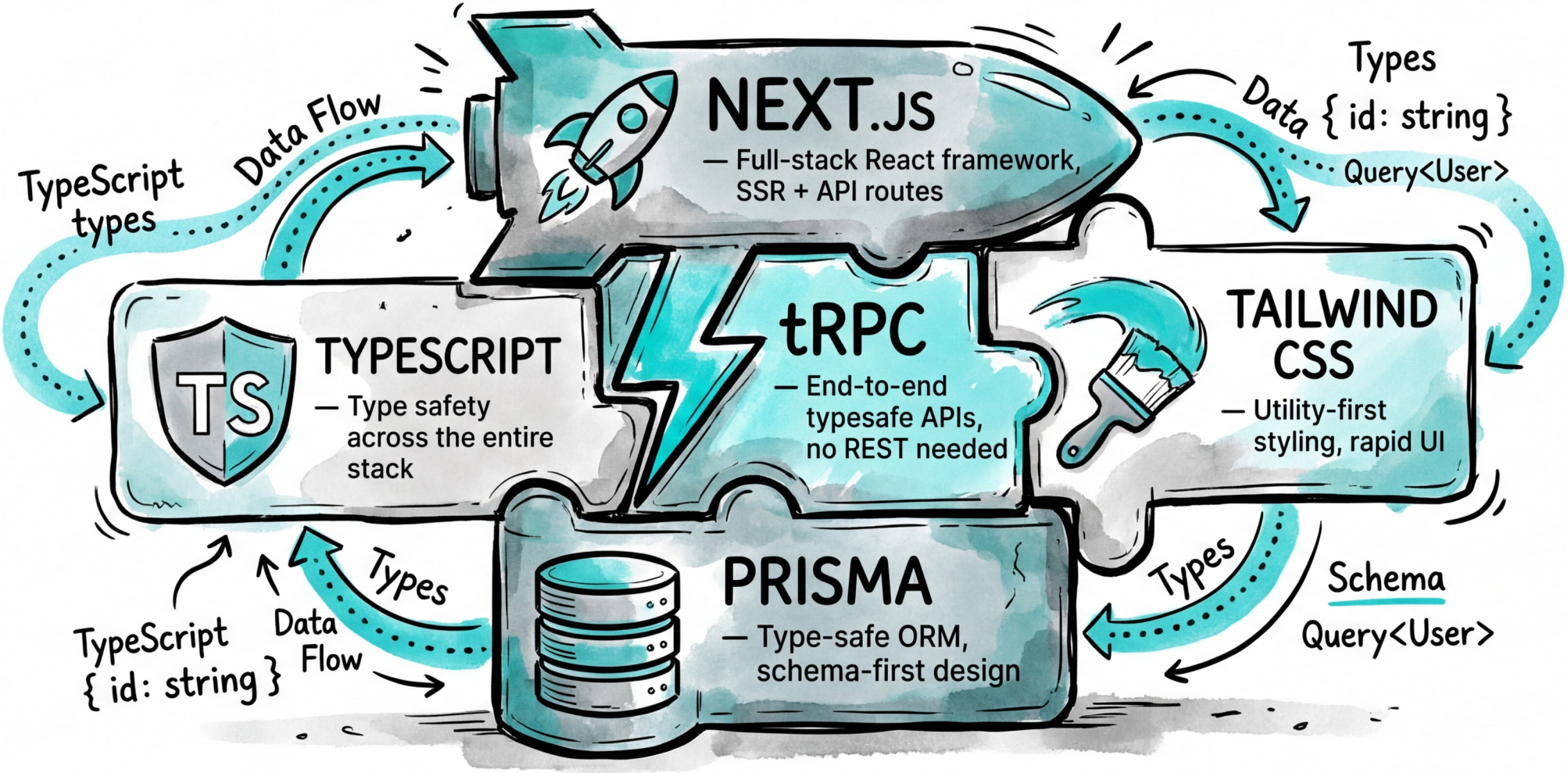


Depend on abstractions, not on concretions. Both should depend on the abstraction.


'VIBE CODING'
Building Software with AI Assistants
DESCRIBE
GENERATE
REVIEW
ITERATE
SHIP
Build me a dashboard
// Dashboard.js
const React = require('react');
function Dashboard() {
return
}
You describe the vibe, AI writes the code

THE T3 STACK

Next.js + TypeScript + tRPC + Tailwind + Prisma



Full-stack TypeScript, end to end

WHY PATTERNS MATTER FOR VIBE CODING

Structure for AI-Generated Code

WITHOUT PATTERNS

AI generates spaghetti code



Each prompt creates inconsistent structure!

Tangled Mess!

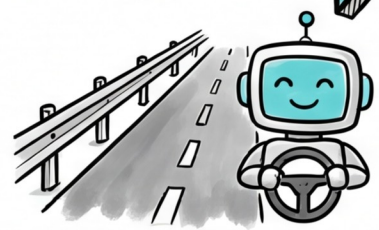
Example 1

Use Repository pattern for data access

```
prisma.user.findMany()
prisma.post.create()
```

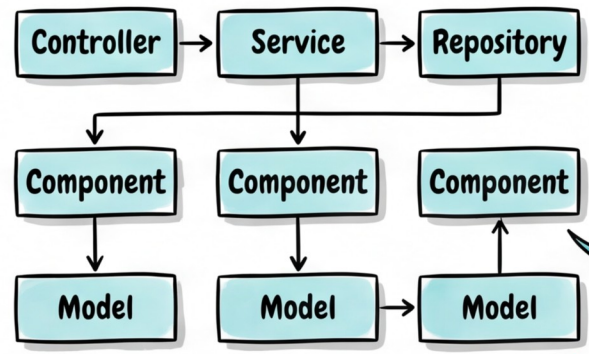
→ consistent Prisma usage ✓

Patterns give AI guardrails



WITH PATTERNS

AI generates code following clear patterns



Consistent structure

Modular components

Consistent structure!

Modular components!

Example 2

Use Factory for components

```
ComponentFactory.create('Button')
ComponentFactory.create('Card')
```

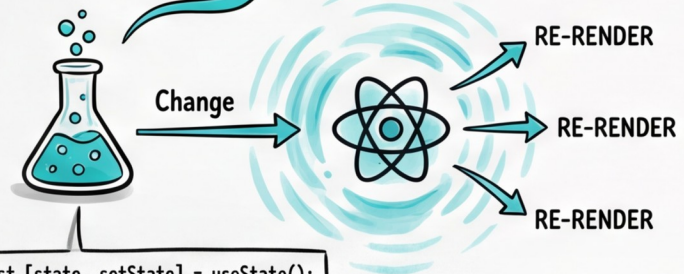
→ consistent React components ✓

> "Patterns are the prompt engineering of architecture"

PATTERNS IN THE T3 STACK

You Already Use Them

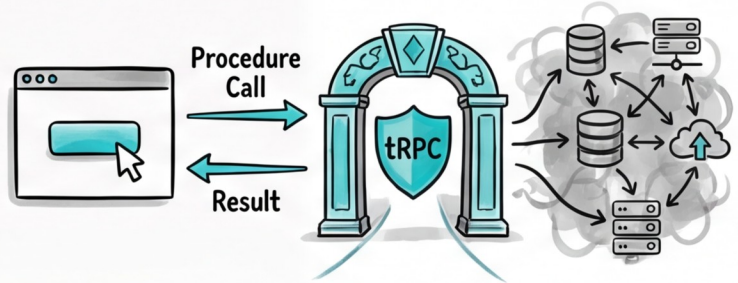
OBSERVER → React useState/useEffect




```
const [state, setState] = useState();
useEffect(() => {
  // Re-runs on change
}, [state]);
```

Component re-renders when state changes.

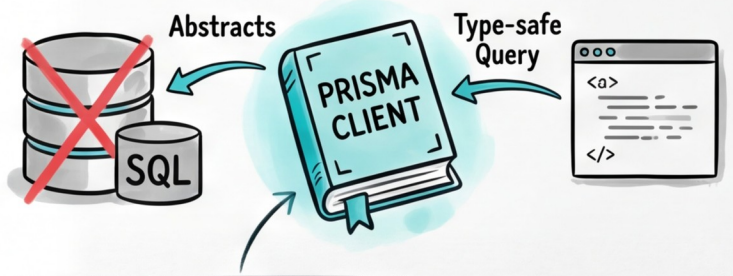
FACADE → tRPC router




```
trpc.user.get.useQuery();
// Simple call, complex backend hidden
```

Simple procedure calls hiding complex backend logic.

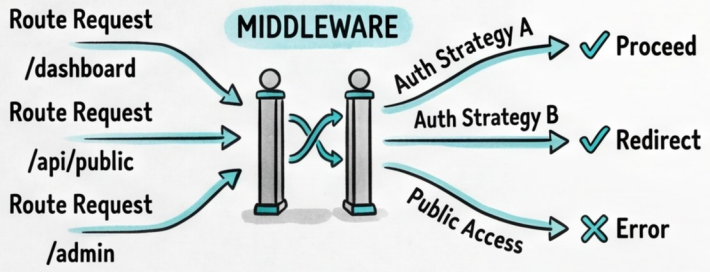
REPOSITORY → Prisma




```
const users = await db.user.findMany();
// No raw SQL needed
```

db.user.findMany() abstracts SQL entirely.

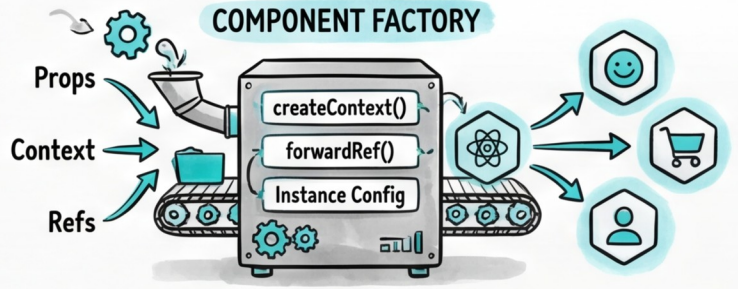
STRATEGY → Next.js middleware




```
export function middleware(request) {
  // Different checks based on path
}
```

Different auth strategies per route.

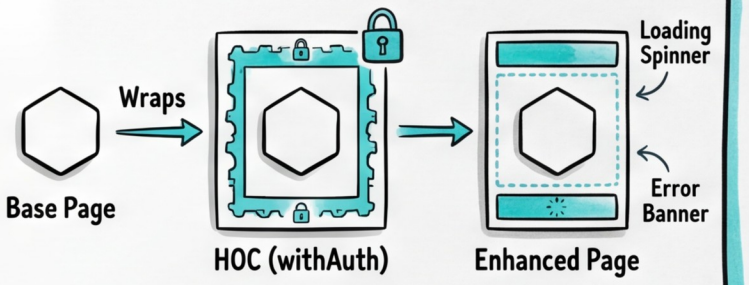
FACTORY → React components




```
const MyContext = createContext();
const FancyButton = forwardRef(...)
```

createContext(), forwardRef() creating configured instances.

DECORATOR → Higher-order components



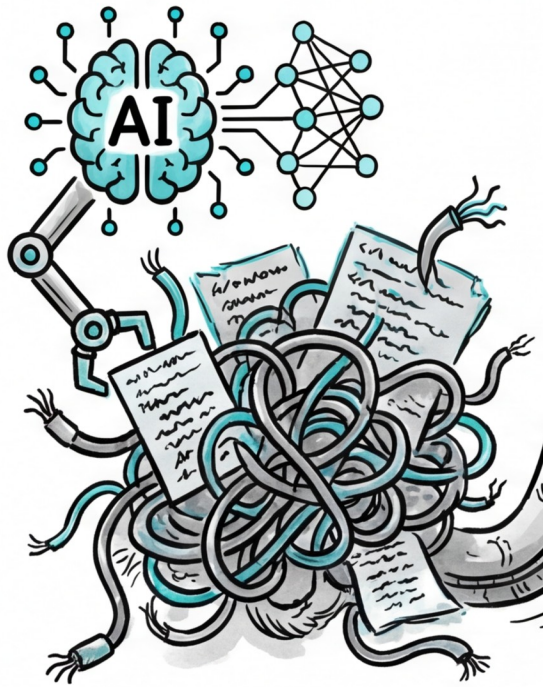

```
export default withAuth(ProfilePage);
// Adds auth behavior, keeps component intact
```

withAuth(Page), adding behavior without changing original.

REFACTORING AI CODE

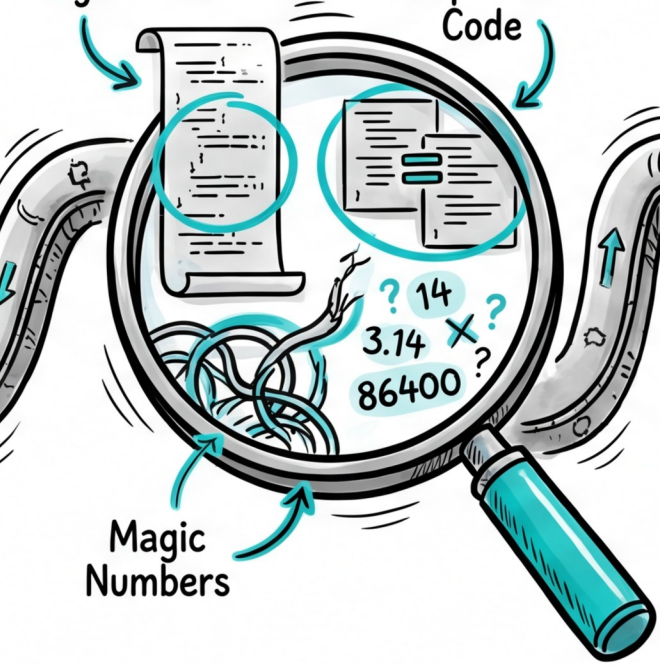
— From Vibes to Production Quality —

1 GENERATE

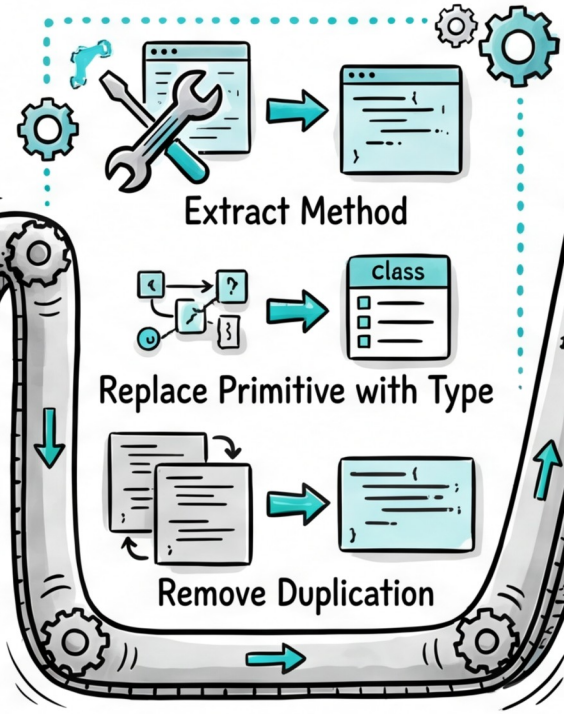


AI produces working but messy code (tangled wires)

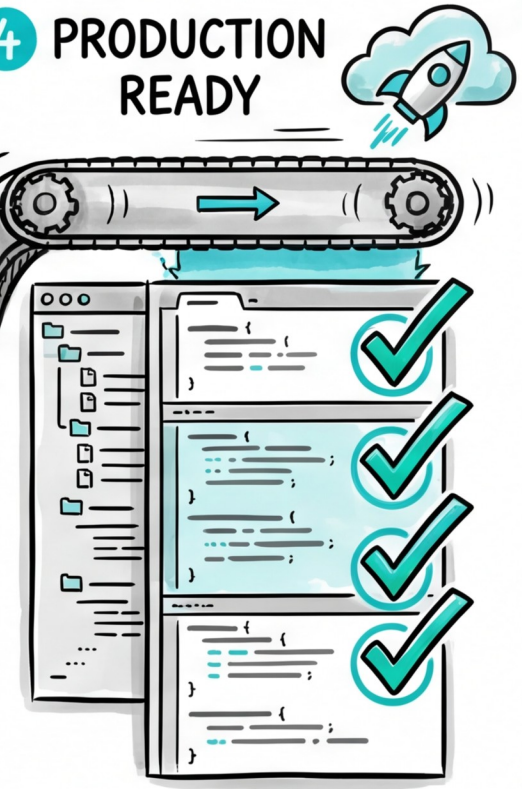
2 IDENTIFY SMELLS



3 APPLY TECHNIQUES



4 PRODUCTION READY



clean, tested, typed code with green checkmarks

AI writes the first draft. You refactor it into art.

ESLint ✓ | Prettier | TypeScript strict mode | Jest ✓

CLEAN ARCHITECTURE + AI

Best Practices for the AI Era

Patterns + Refactoring + Types = Maintainable AI Code

Type everything

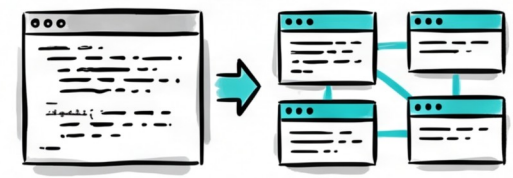
- TypeScript strict
- no ~~any~~

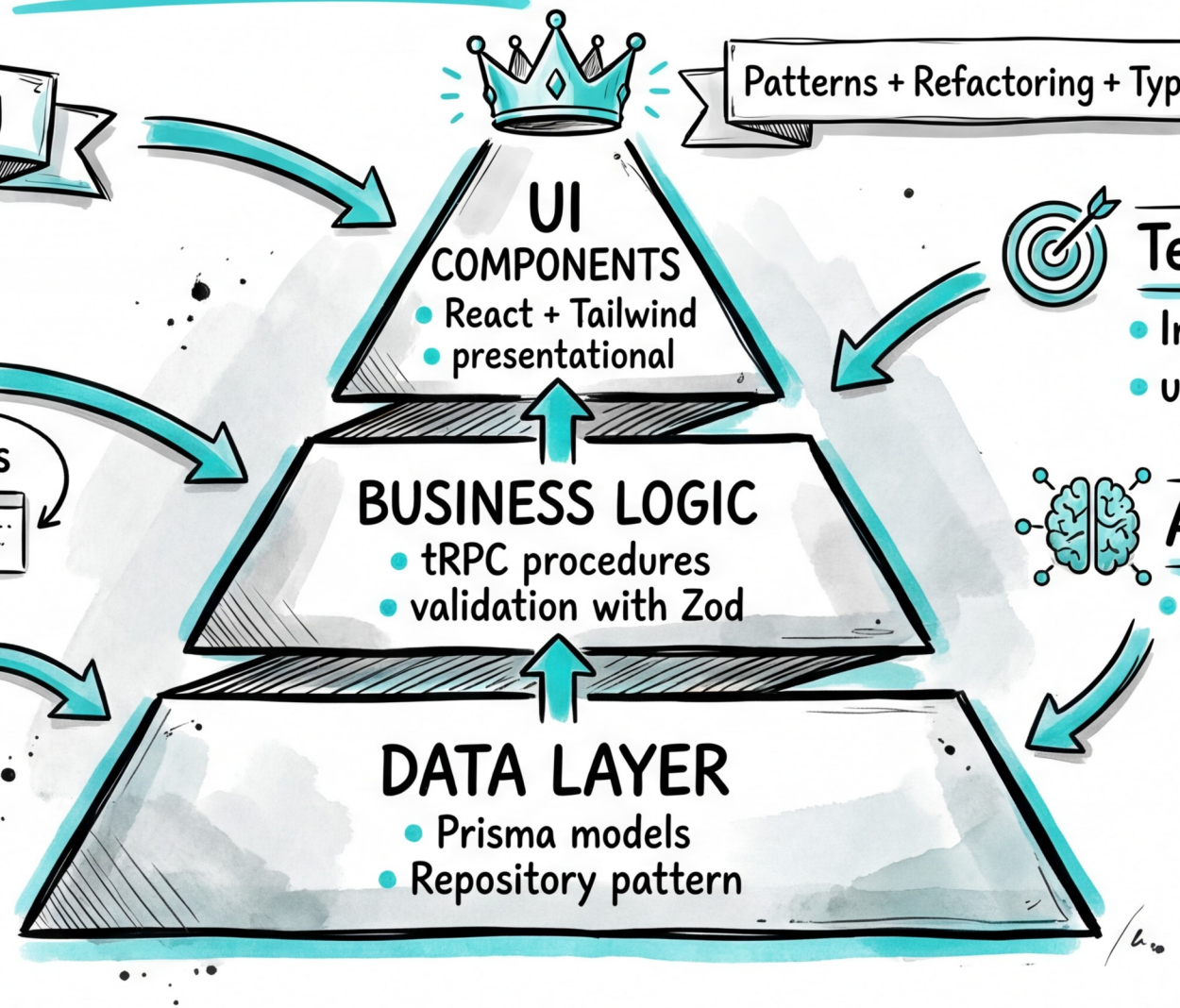
Schema first

- Zod + Prisma schemas define contracts

Small functions

- Extract until each does one thing





Test boundaries

- Integration tests on tRPC
- unit tests on utils

AI-friendly structure

- clear file organization helps AI understand context

