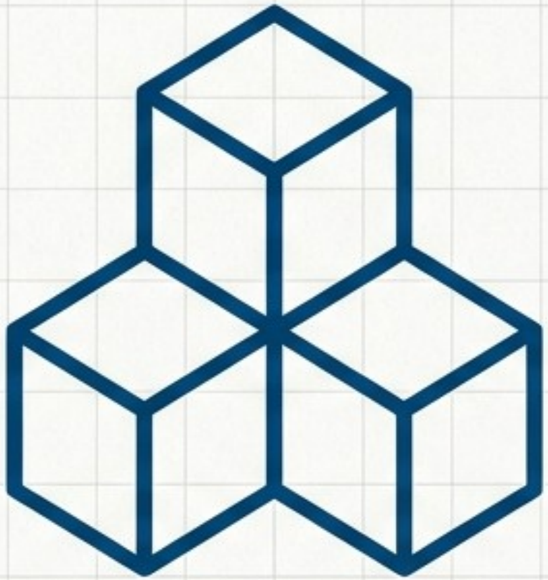


The Architect’s Blueprint

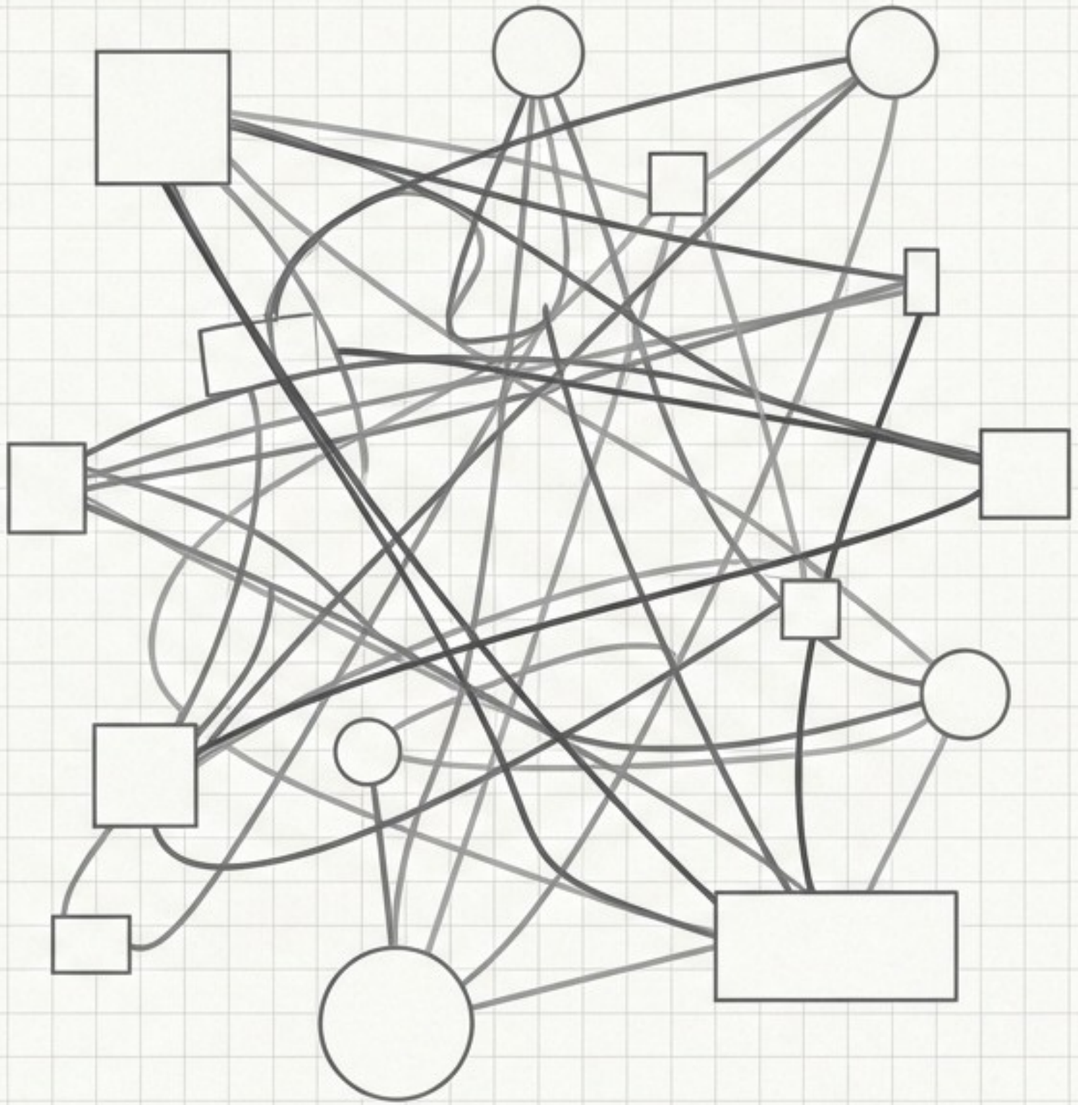
Building Flexible and Maintainable Systems with Structural Design Patterns

The Inevitable Chaos of Unstructured Systems

You change one class → 10 others break.

The Symptoms

- ❌ **Tightly Coupled**: Components are highly dependent, creating a fragile system.
- ❌ **Difficult to Modify**: Small changes cause cascading failures.
- ❌ **Full of Duplicate Code**: Logic is copied instead of reused.
- ❌ **Hard to Integrate**: Connecting with other systems is a major challenge.
- ❌ **Overly Complex:** A tangled web of objects calling each other directly.



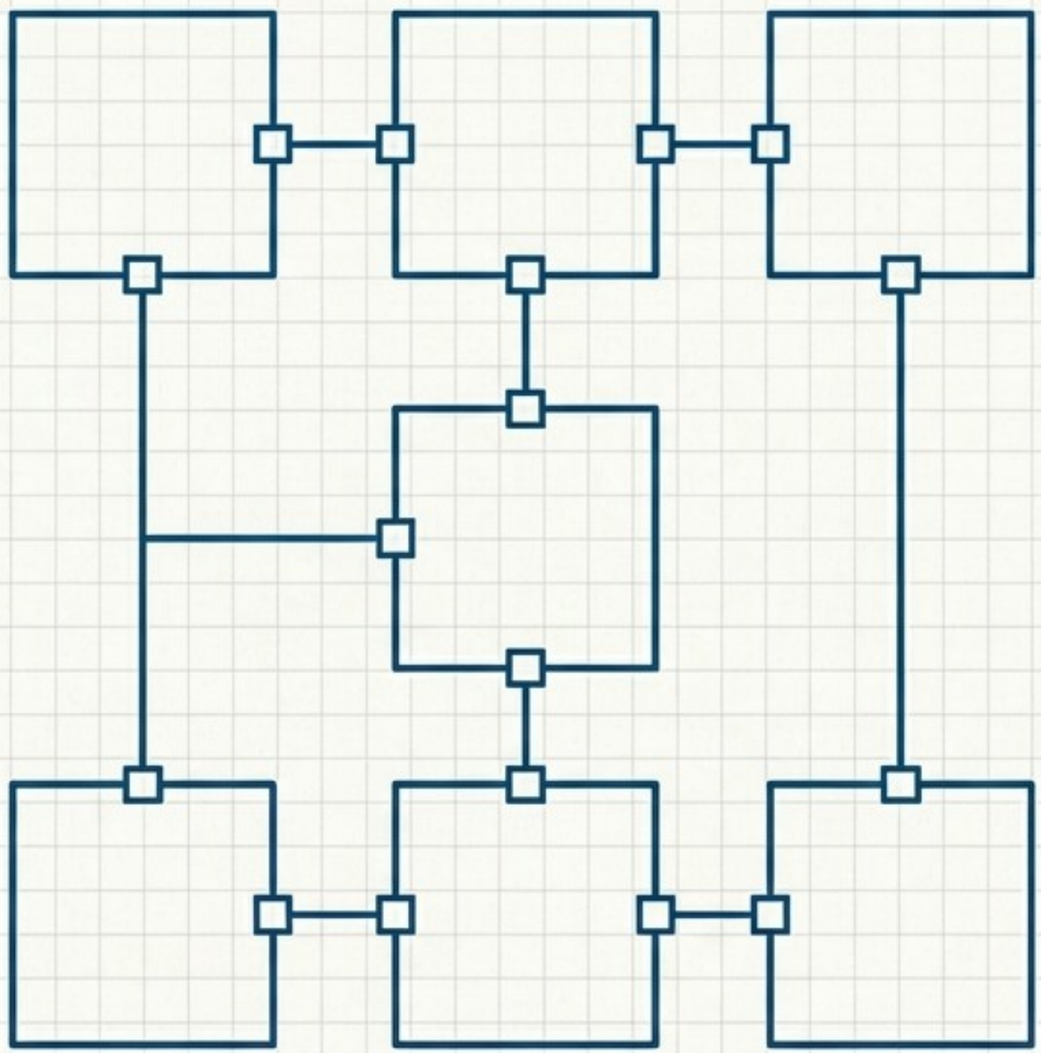
Bringing Order to Chaos: The Role of Structural Patterns

Structural patterns define how classes and objects are organised and combined, focusing on their relationships to build systems that are flexible, reusable, and easy to maintain.

Composition	Object Relationships
Wrappers & Adapters	Simplifying Complexity

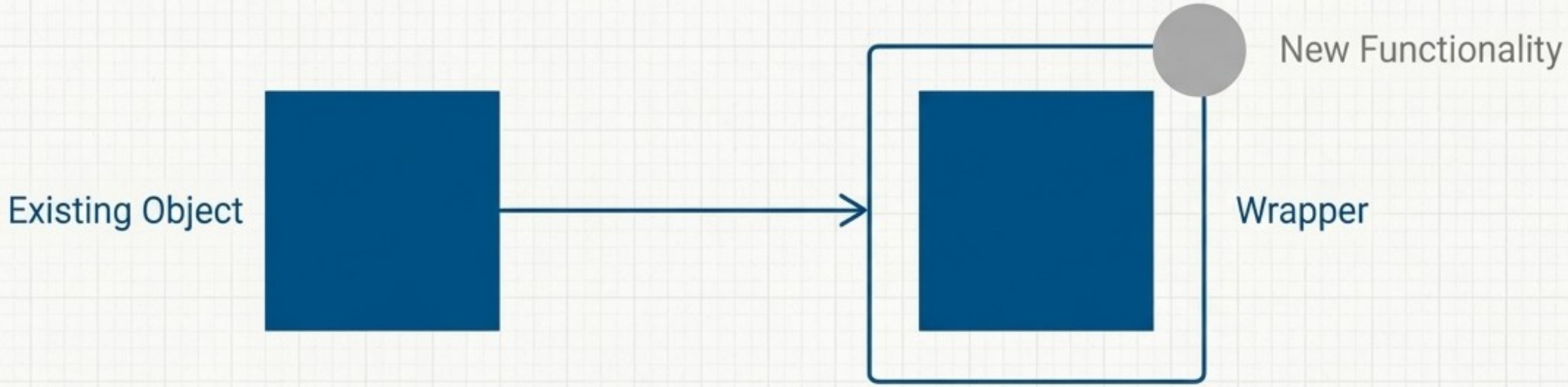
The Benefits

- ✔ **Modular**: Components are independent and interchangeable.
- ✔ **Extensible**: Aligned with the Open/Closed Principle (OCP) – extend functionality instead of modifying existing code.
- ✔ **Easier to Integrate**: Systems connect cleanly and predictably.



The Guiding Philosophy: Compose, Don’t Modify

Instead of modifying the internal code of existing objects, structural patterns use a more robust strategy: we compose, wrap, or combine them in intelligent ways.



Key Ideas Used Repeatedly

- **Composition over Inheritance:** A foundational principle for flexible design.
- **Adapters:** Placed around incompatible interfaces.
- **Wrappers:** To extend behaviour dynamically.
- **Proxies:** To control access to objects.
- **Facades:** To simplify complex subsystems.
- **Bridges:** To separate an abstraction from its implementation.

The Architect’s Toolkit: Seven Structural Patterns

Adapter

Make incompatible things work together.

Facade

Give a simple interface to a complex system.

Decorator

Add features without modifying a class.

Composite

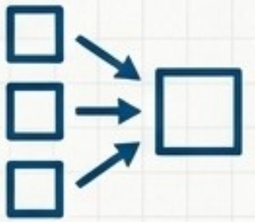
Treat individual objects and compositions uniformly.

Bridge

Separate abstraction from implementation.

Proxy

Control access to another object.



Flyweight

Reuse objects to save memory.

1. The Adapter Pattern

Makes two incompatible systems or interfaces work together without changing their source code.

The Analogy

Like using a plug adapter to charge a UK laptop in the US, the pattern converts one interface into another that a client expects.

The Enterprise Application

Scenario: Integrating a modern service with a legacy payment system.

Your system communicates using JSON, but the old system only understands XML. An Adapter sits in the middle, translating the JSON requests into XML and the XML responses back into JSON.

2. The Facade Pattern

Provides a single, simplified interface to a complex subsystem of classes, libraries, or microservices.

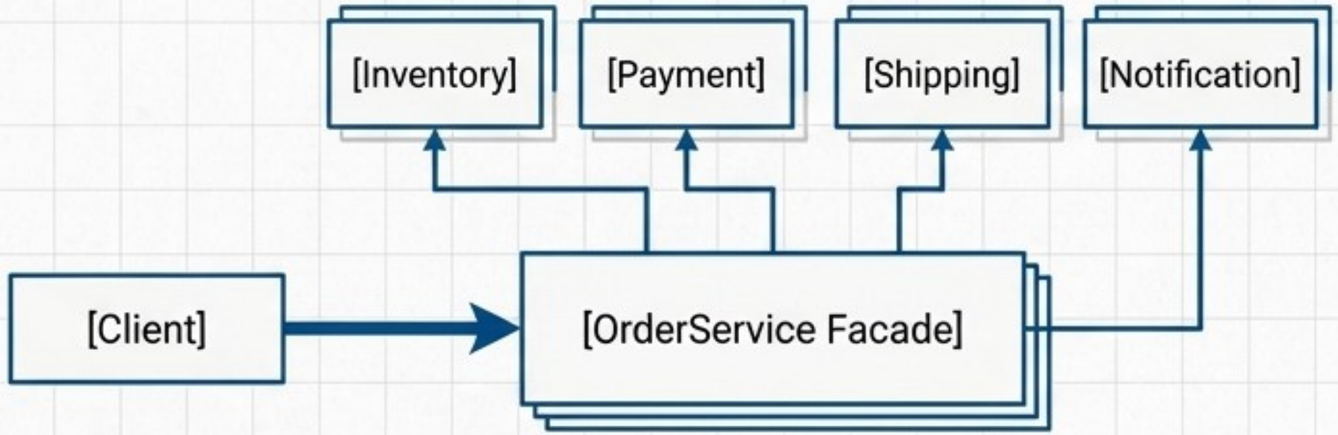
The Analogy

A remote provides simple buttons (Power, Volume Up). You don't need to know about the complex internal wiring and chips required to make the TV work.

The Enterprise Application

Scenario: An e-commerce ordering process.

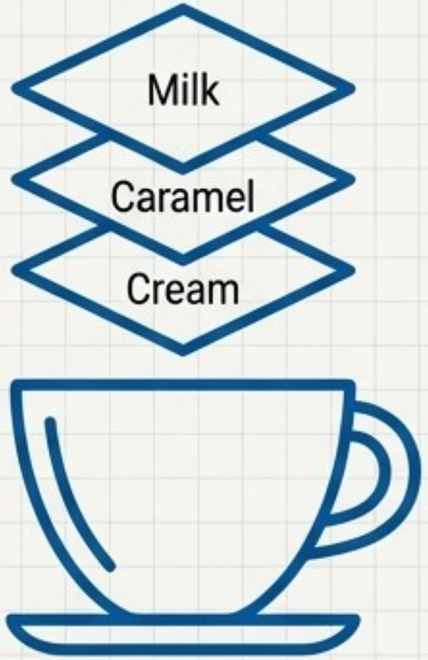
An `OrderService` acts as a facade. The client calls a single `placeOrder()` method on it. Internally, the facade orchestrates calls to multiple complex subsystems: `InventoryService`, `PaymentService`, `ShippingService`, and `NotificationService`.



3. The Decorator Pattern

Allows behaviour to be added to an individual object, either statically or dynamically, without affecting the behaviour of other objects from the same class.

The Analogy

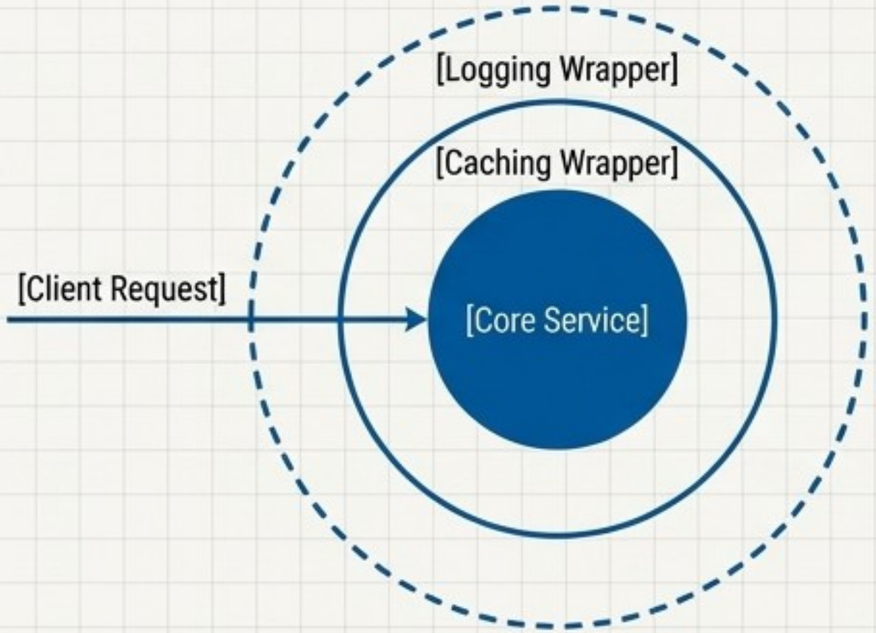


You start with a base coffee. You can then 'decorate' it with milk, then caramel, then cream. Each is a separate layer of functionality added on top of the original.

The Enterprise Application

Scenario: Adding cross-cutting concerns to a data service.

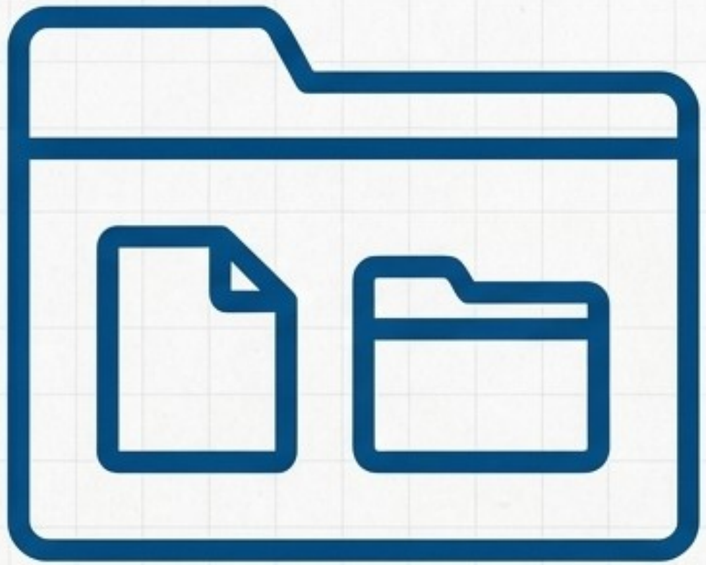
You have a core `DataRepository`. Instead of modifying it, you wrap it with decorators. A `CachingDataRepository` can wrap it to add caching, and a `LoggingDataRepository` can wrap that to add logging for every call.



4. The Composite Pattern

Composes objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

The Analogy

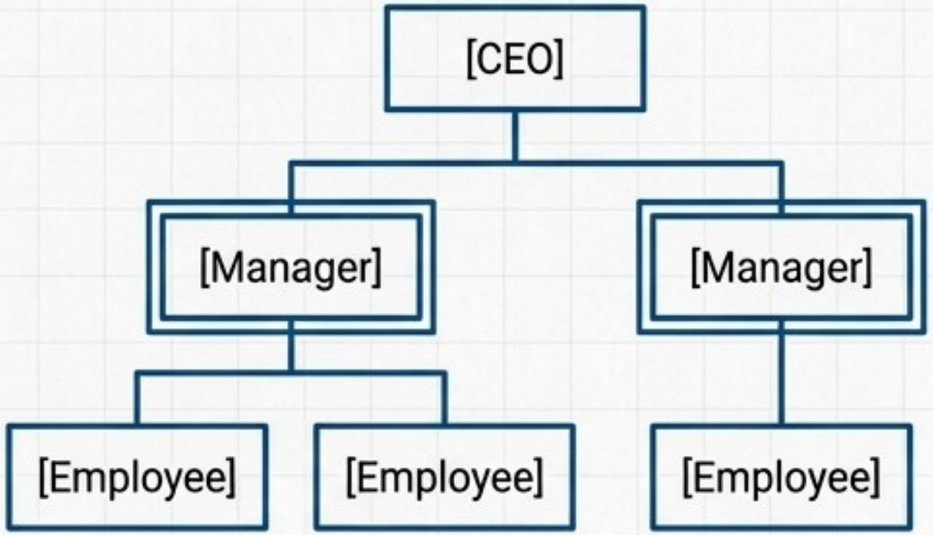


A file system. A folder can contain individual files or other folders (which themselves contain files/folders). You can perform operations like "get size" on both a single file and an entire folder.

The Enterprise Application

Scenario: Representing an organisational hierarchy.

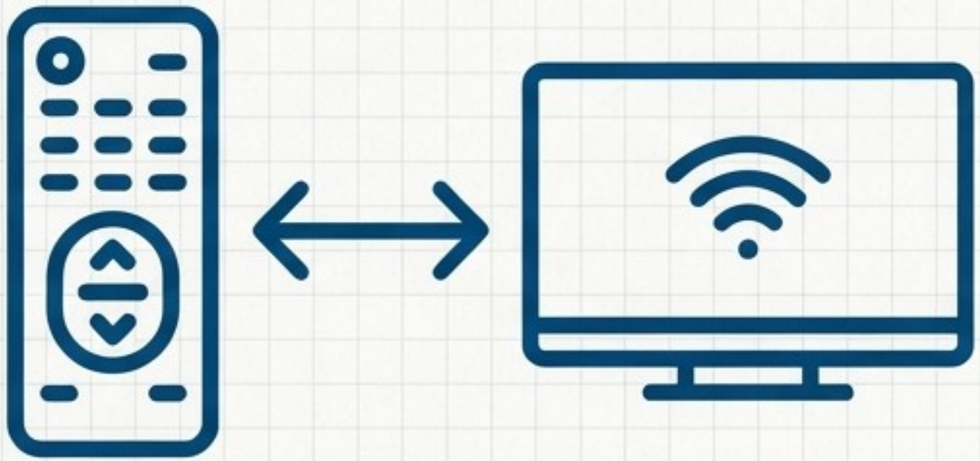
A CEO is at the top of the tree. Under them are Managers (who are also employees), and under them are individual Employees. The system can calculate total salary costs for an individual employee or an entire department using the same method call.



5. The Bridge Pattern

Decouples an abstraction from its implementation so that the two can evolve independently.

The Analogy

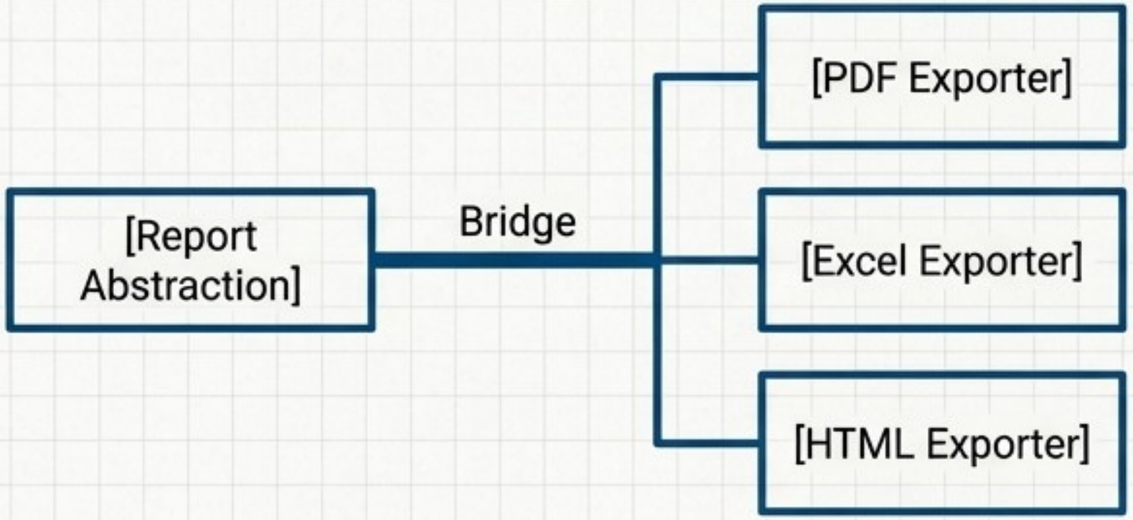


The remote control (the abstraction with buttons like 'channel up') is separate from the TV brand (the implementation - Sony, Samsung). You can use the same remote abstraction to control different TV implementations.

The Enterprise Application

Scenario: A cross-platform reporting system.

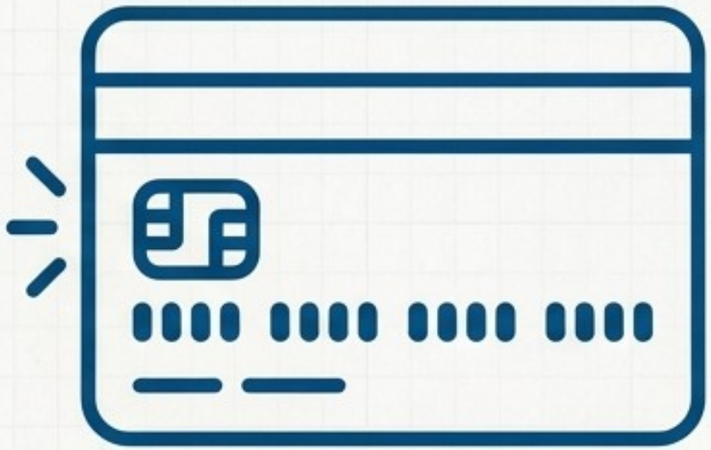
You have a `Report` abstraction (e.g., `SalesDataReport`). This can be implemented by multiple, interchangeable exporters: `PDFExporter`, `ExcelExporter`, `HTMLExporter`. The core report logic doesn't need to change when you add a new export format.



6. The Proxy Pattern

Provides a surrogate or placeholder for another object to control access to it.

The Analogy

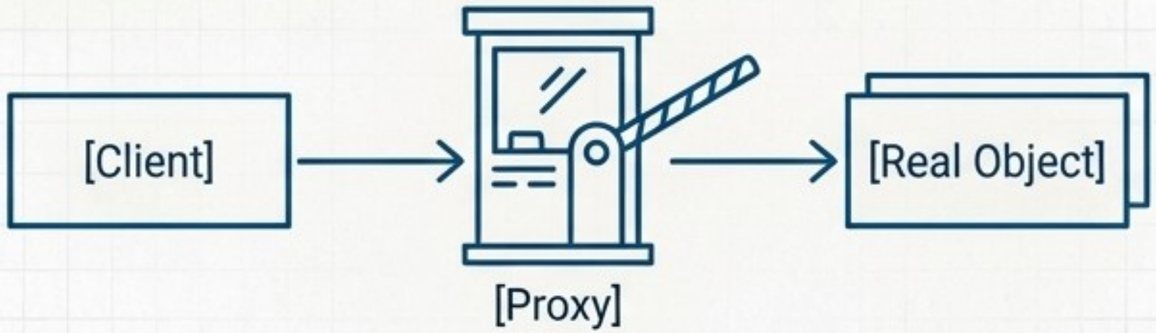


A credit card is a proxy for your bank account. It provides a controlled interface to your funds without giving direct access to the bank account itself. It handles security, transaction limits, etc.

The Enterprise Application

Scenario: Common uses in enterprise systems.

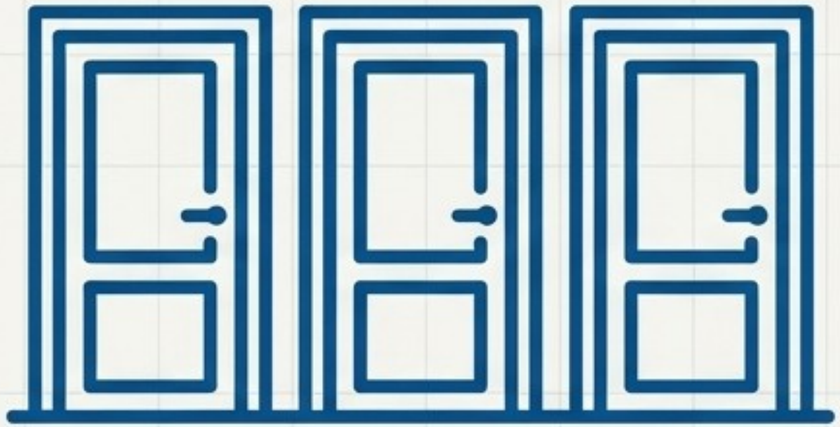
Proxies are used extensively for:
* **Lazy Loading**: A proxy for a large image object only loads the real image from disk when it's actually needed.
* **Access Control**: A security proxy checks a user's permissions before allowing a call to a sensitive method on the real object.
* **API Gateway**: Acts as a proxy to downstream microservices, handling rate limiting, authentication, and caching.



7. The Flyweight Pattern

Minimises memory usage by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

The Analogy

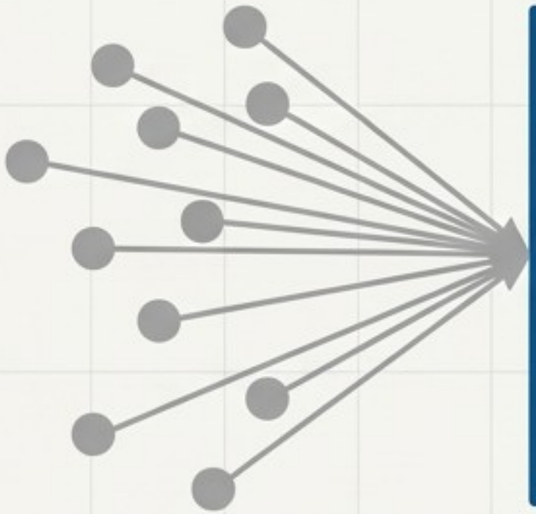


A hotel chain doesn't create unique design specifications for the TV, AC, and bed in every single room. It reuses the same "Standard Double Room" specifications (the shared, intrinsic state) for thousands of rooms.

The Enterprise Application

Scenario: Rendering a massive game world.

A game engine needs to render millions of objects like trees, rocks, and bullets. Instead of storing the mesh, texture, and physics data for every single tree, it creates one `**Tree**` flyweight object and then renders it in thousands of different positions with unique states (e.g., location, health).



[Tree Flyweight Object (Shared Data: Mesh, Texture)]

Structural Patterns: A Summary

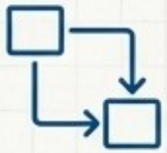
Pattern	Key Idea	Real-Life Analogy	Enterprise Example
Adapter	Make two systems compatible	Plug adapter	Legacy API integration
Facade	Simplify complex subsystems	TV remote	Orchestration service
Decorator	Add features dynamically	Add toppings to pizza	Logging / caching wrappers
Composite	Create tree-like structures	Folders in a file system	Organisational hierarchy
Bridge	Separate abstraction & implementation	Remote + TV	UI themes / data exporters
Proxy	Control access / lazy load	Credit card	API gateway / security
Flyweight	Optimise memory via sharing	Shared hotel room specs	Game engine objects

Beyond the Patterns: The Principles of Sound Structure

Structural patterns are not just isolated solutions; they are physical manifestations of fundamental principles that lead to robust, maintainable software.

Core Focus

Organising objects and classes into larger structures.



Building cleaner relationships between components.

Designing an architecture that can evolve without breaking.

Key Principles They Heavily Support

✓ **The Open/Closed Principle (OCP)**
You can extend a system's behaviour without modifying its source code. (Decorator, Bridge)

✓ **The Single Responsibility Principle (SRP)**
Classes are kept small and focused. (Facade, Proxy)

✓ **Composition over Inheritance**
Building complex functionality by combining simple objects is more flexible than inheriting from complex base classes. (The foundation for most patterns)

From Accidental Complexity to Intentional Design

Good structure is not an accident. It is the result of deliberate architectural decisions.

Structural patterns provide the vocabulary and the blueprint to make those decisions effectively, ensuring the systems you build today are the systems you can still maintain and extend tomorrow.