Design Patterns

Day 1 - History, Benefits, Classification

Manimaran / Aug 30, 2025

References

📖 Books

- *Head First Design Patterns* by Eric Freeman & Elisabeth Robson
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four)

🌐 Online Resource

- https://refactoring.guru/design-patterns
  (Excellent beginner-friendly explanations with diagrams and code samples in multiple languages)

History of Design Patterns

- Christopher Alexander (architecture → software)
- GoF (Gang of Four) - Design Patterns: Elements of Reusable Object-Oriented Software, 1994 book → 23 core patterns
- Influence on modern frameworks

Why Use Patterns?

- Avoid reinventing the wheel
- Reusable, proven solutions
- Common vocabulary among developers
- Some Criticisms

Metrics for Good Design

- Low coupling
- High cohesion
- Reusability, Extensibility, Maintainability

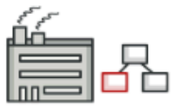
Classification of Patterns

- **Creational** – object creation
- **Structural** – object composition
- **Behavioral** – object interaction
- **Application** – architectural styles (MVC, MVP, MVVM, MVI, VIPER, Microservices)

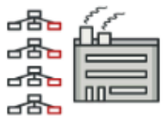
Catalog Introduction

Creational patterns

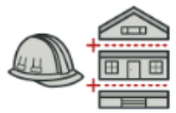
These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.



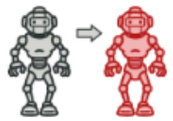
Factory Method



Abstract Factory



Builder

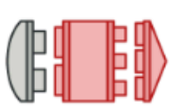


Prototype

Singleton

Structural patterns

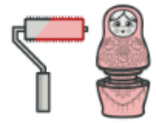
These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



Adapter

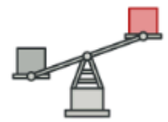
Bridge

Composite



Decorator

Facade



Flyweight

Proxy

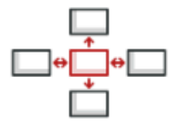
Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

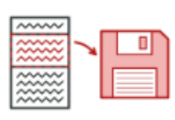
Chain of Responsibility

Command

Iterator

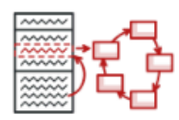


Mediator



Memento

Observer



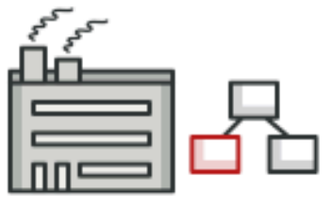
State

Strategy

Template Method

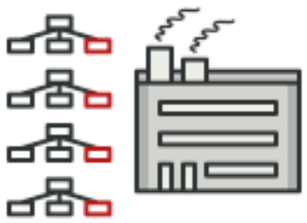
Visitor

Creational Patterns



Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

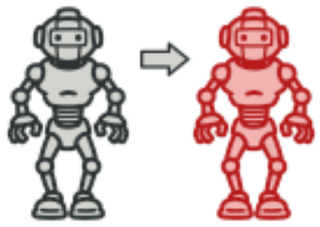


Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

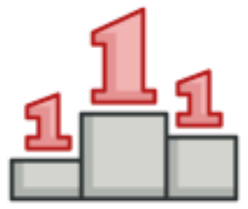
Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Prototype

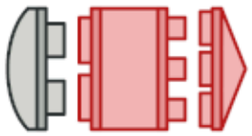
Lets you copy existing objects without making your code dependent on their classes.



Singleton

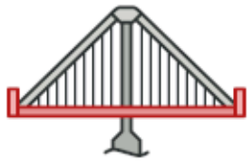
Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Structural Patterns



Adapter

Allows objects with incompatible interfaces to collaborate.

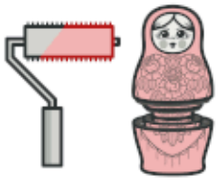


Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

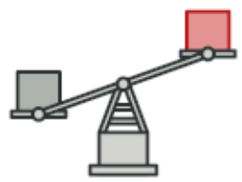


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

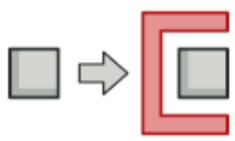
Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Behavioral Patterns

Chain of Responsibility

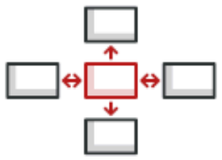
Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

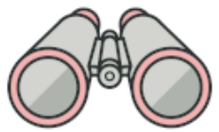


Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

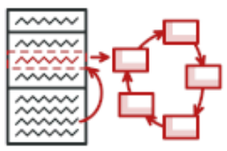
Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.



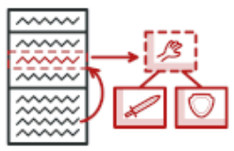
Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.



State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Visitor

Lets you separate algorithms from the objects on which they operate.

Application Patterns

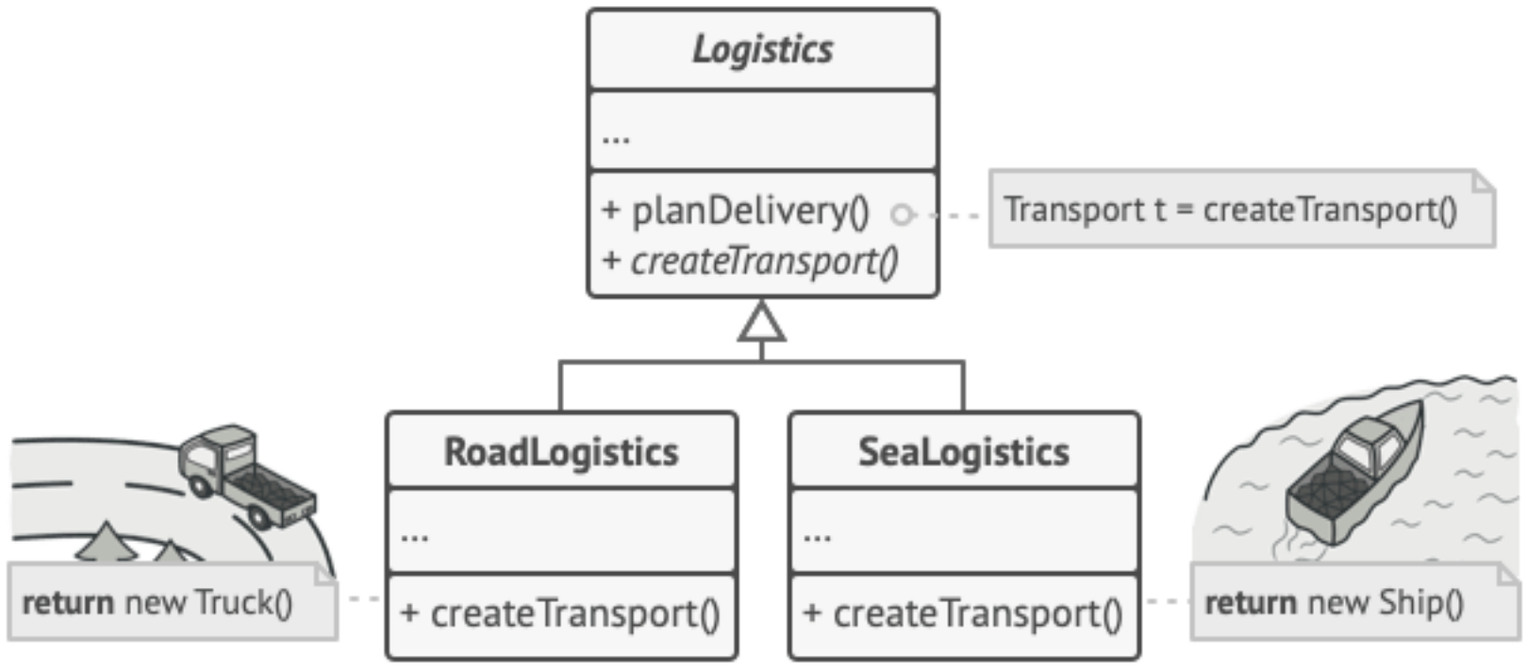
- MVC (Model-View-Controller)
- MVP (Model-View-Presenter)
- MVVM (Model-View-ViewModel)
- MVI (Model-View-Intent)
- VIPER (View-Interactor-Presenter-Entity-Router)
- Microservices Patterns (API Gateway, Circuit Breaker, CQRS)

Intro to Creational Patterns

- Encapsulate object creation → flexibility, maintainability
- Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

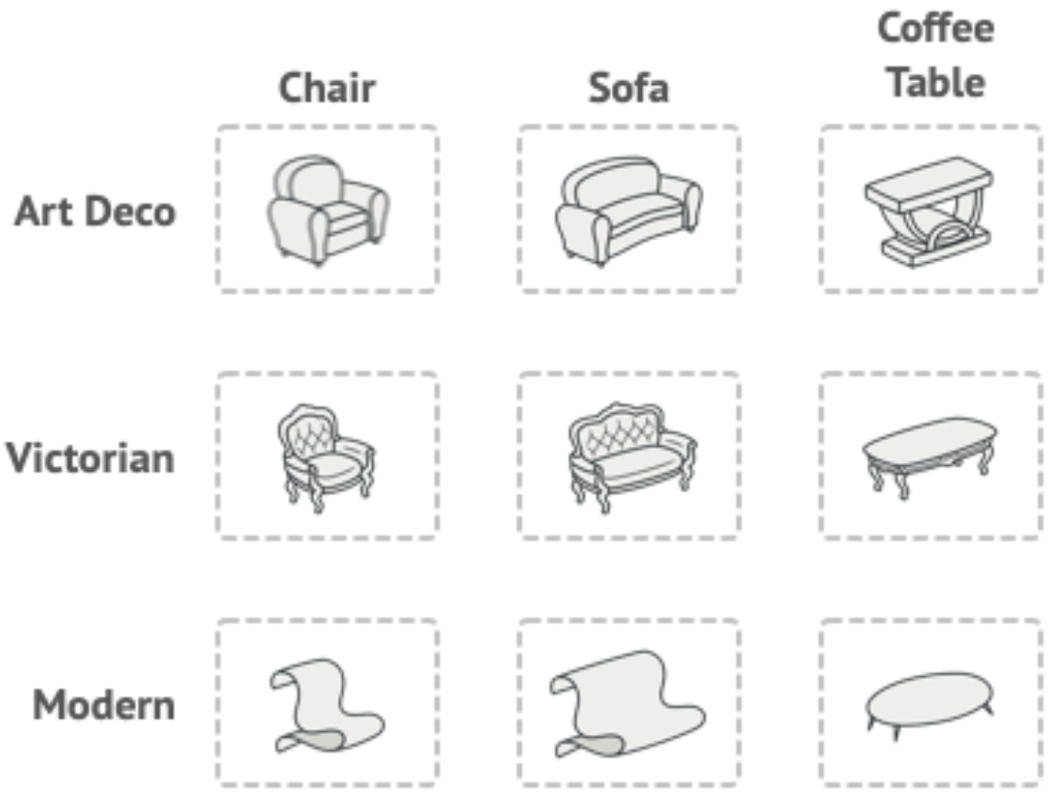
Factory Method

- Define a **common interface** for creating objects.
- Let **subclasses (or a factory class)** decide which concrete class to instantiate.
- Helps avoid spreading `new` keyword + `if/else` everywhere.
- **Real World:** Hiring factory (developer, tester)



Abstract Factory

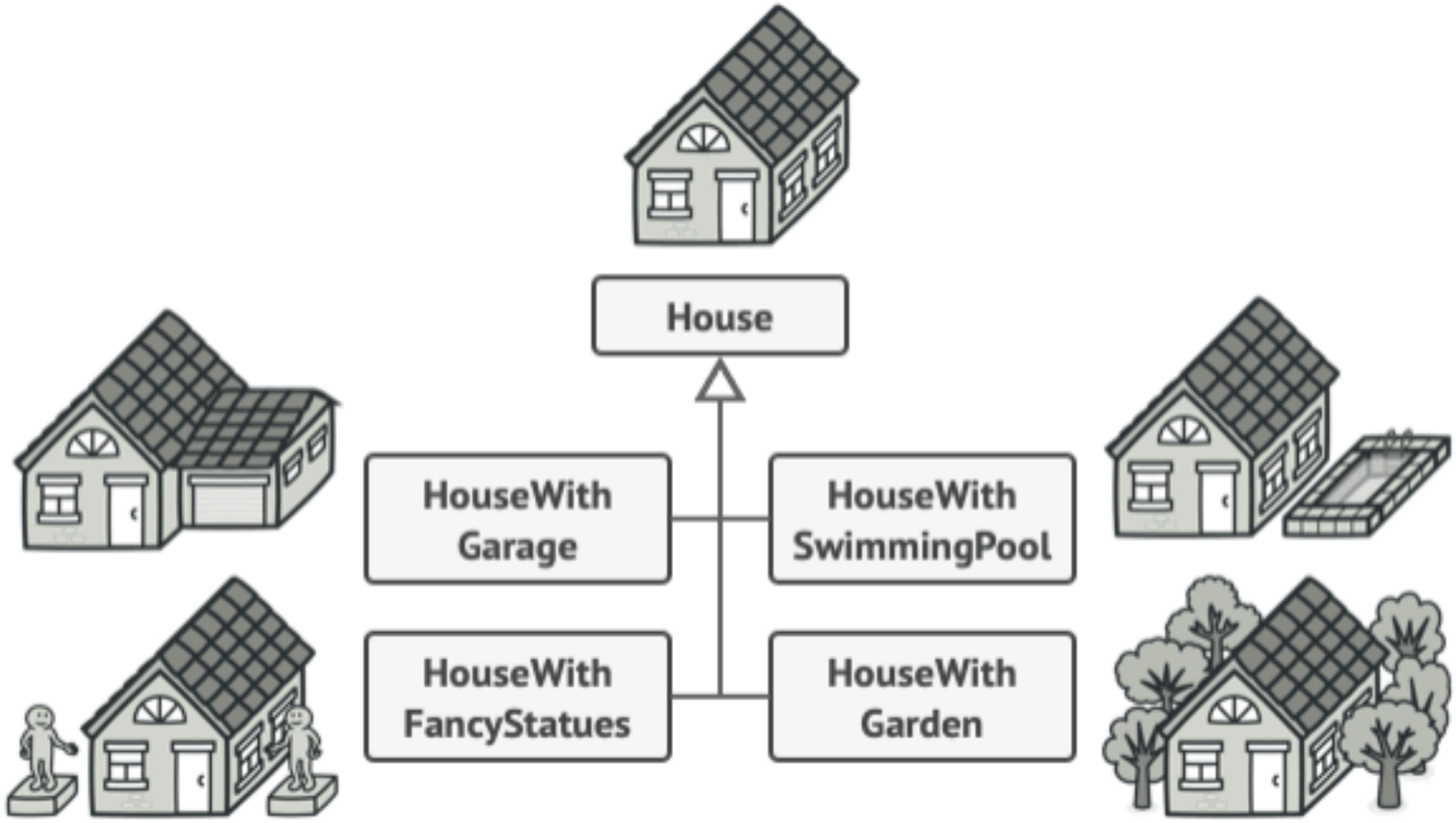
- Create families of related objects
- **Real World:** GUI toolkit → Light & Dark theme components
- **Code Demo** → FurnitureFactory



Product families and their variants.

Builder

- Step-by-step object construction
- **Real World:** House Builder
- House h = new House(2, true, true, false, "marble", "sloped");



Quiz

Q: You need different payment methods (Card, UPI, Wallet). Which Creational Pattern fits?

Ans: Factory Method

Q: Which pattern helps you switch between *Light and Dark* UI themes easily?

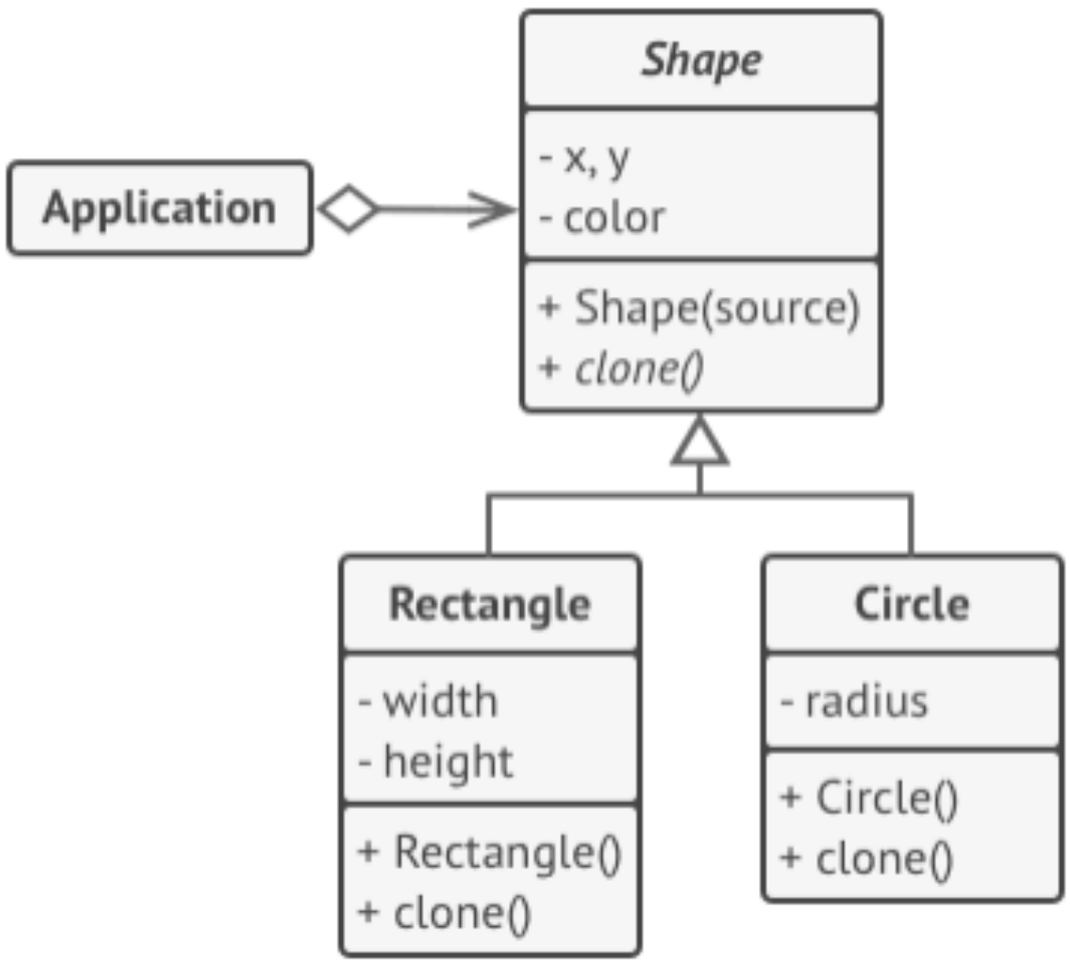
Ans: Abstract Factory

Q: You need different Laptop configurations (RAM, SSD, GPU). Which pattern?

Ans: Builder

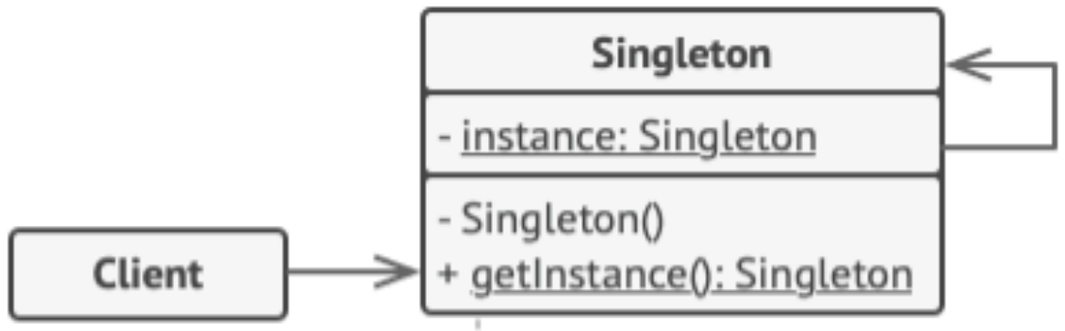
Prototype

- Clone existing object → save creation cost
- **Real World:** Shapes



Singleton

- Only one instance → global access point
- **Real World:** Database connection, Logger




```
if (instance == null) {
    // Note: if you're creating an app with
    // multithreading support, you should
    // place a thread lock here.
    instance = new Singleton()
}
return instance
```

1 The **Singleton** class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

Quiz

Q: In a game, you duplicate enemies with same properties. Which pattern?

Ans: Prototype

Q: Which pattern ensures a single instance across the app?

Ans: Singleton

Review

- Factory Method

  Creates **one object** at a time - Hire Developer or Tester

- Abstract Factory

  Creates a **family of objects** - Modern furniture set (Chair + Sofa)

- Builder

  Builds objects **step by step -** Build a House (floors, garage, garden)

- Prototype

  Clone an existing object - Copy shapes in a drawing app

- Singleton

  Only **one instance** in the system - Logger or Database connection

Problem 1: Vehicle Factory

You are building a transport booking system. The system should support **Car**, **Bike**, and **Bus** as different vehicle types.

- Each vehicle must implement a `drive()` method.
- You should use a **Factory Method** to create the correct vehicle at runtime based on input.

Task:

1. Define a `Vehicle` interface with `drive()`.
2. Implement `Car`, `Bike`, and `Bus`.
3. Implement a `VehicleFactory` class that returns the correct object.
4. In `main()`, take input `"car"`, `"bike"`, `"bus"` → return and call `drive()`.

Problem 2: Pizza Builder

You need to design a Pizza ordering system.

- A Pizza can have **dough**, **sauce**, and **toppings**.
- You should allow flexible combinations (e.g., thin crust + tomato sauce + cheese).
- Implement the Pizza creation using the **Builder Pattern**.

Task:

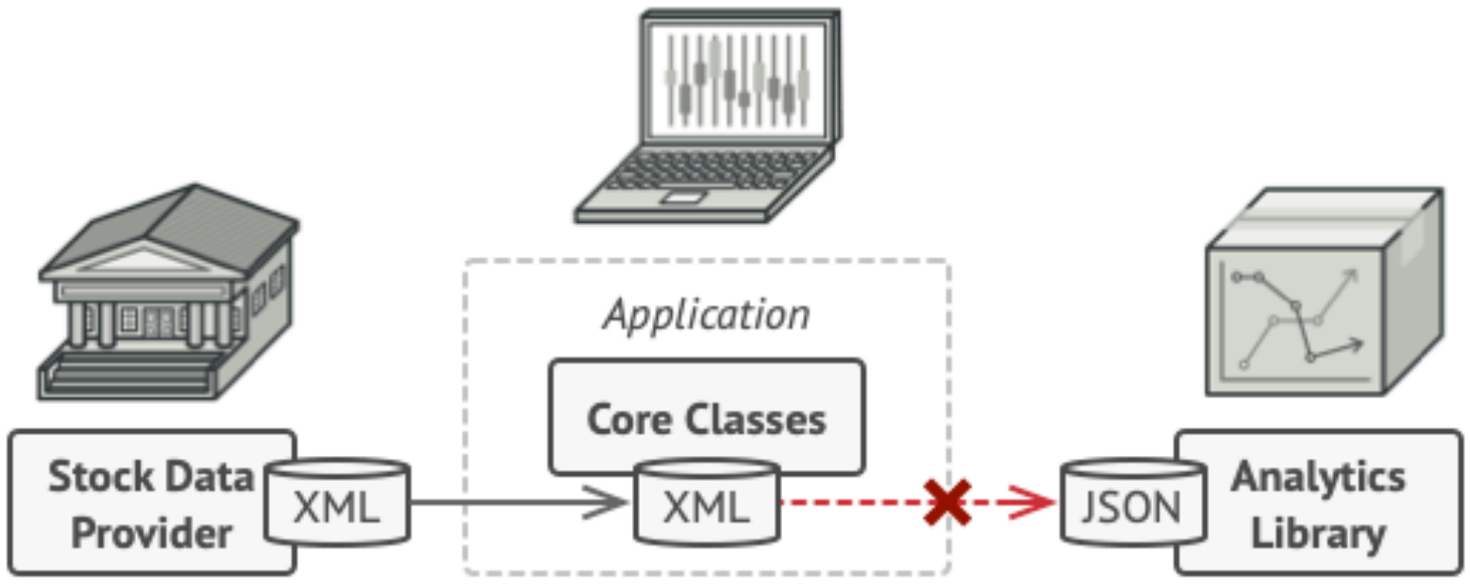
1. Create a `Pizza` class with attributes: dough, sauce, toppings.
2. Implement a `PizzaBuilder` class with `setDough()`, `setSauce()`, `addTopping()` methods.
3. Build a custom pizza in `main()` and print the details.

Intro to Structural Patterns

- Object composition → flexibility, reusability
- Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

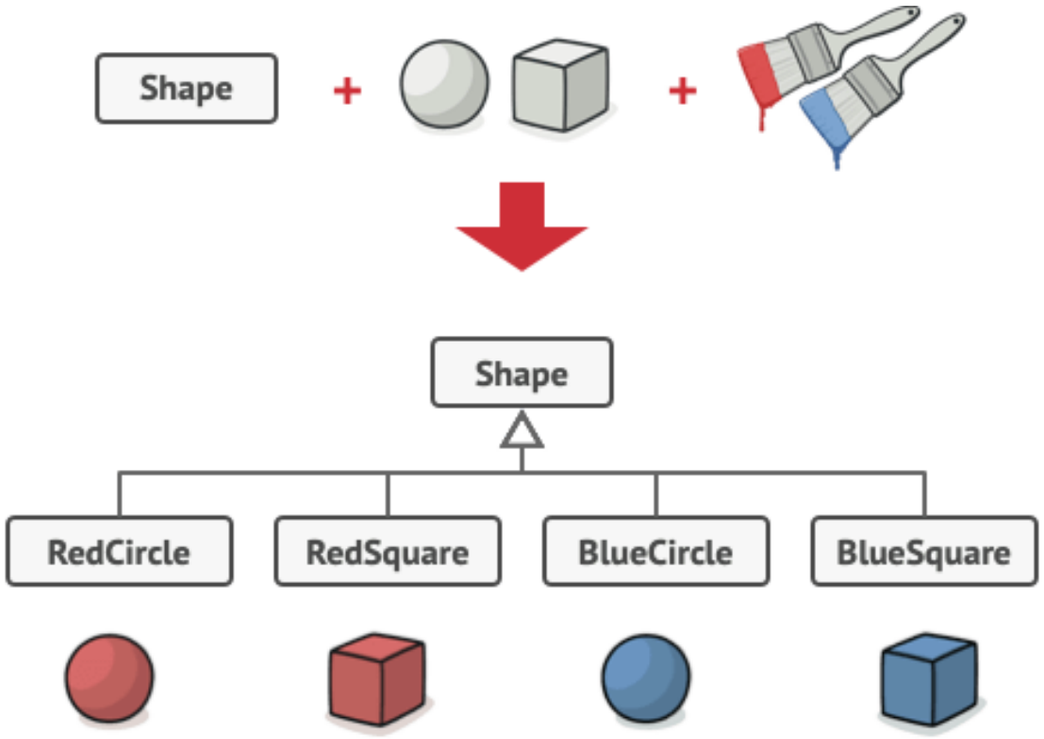
Adapter

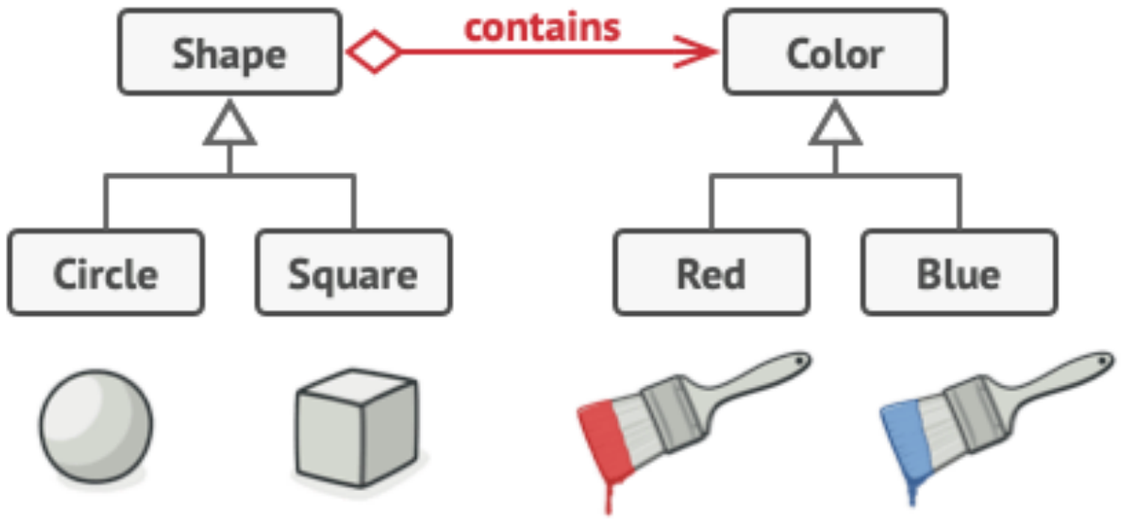
- Convert one interface into another
- **Real World:** Power plug adapter, legacy payment API wrapper



Bridge

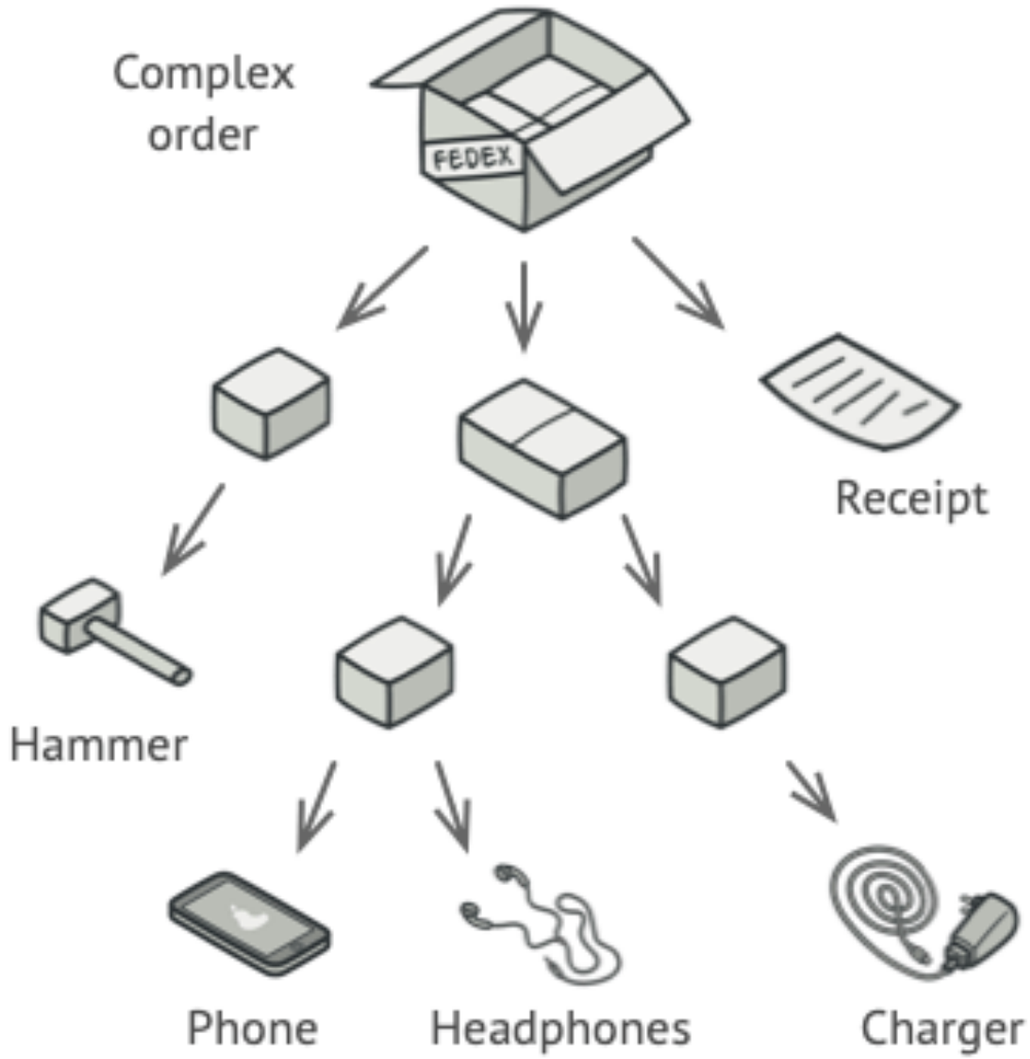
- Decouple abstraction from implementation
- **Real World:** Shape + Color combinations





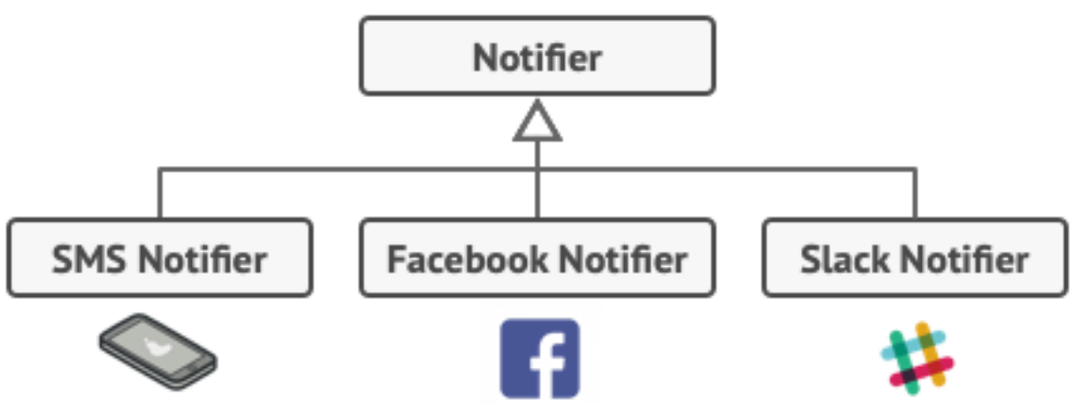
Composite

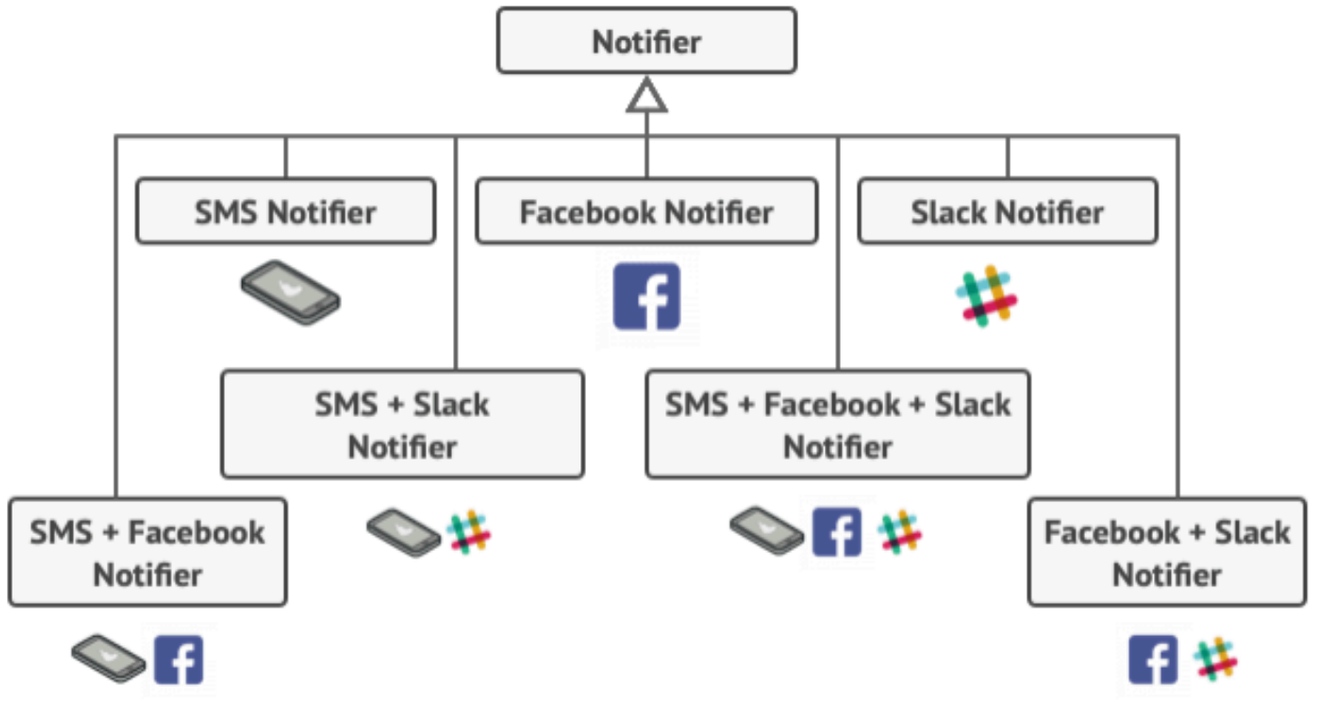
- Hierarchical structures (tree) / object tree
- **Real World:** File system (Folders + Files), Org Chart



Decorator

- Add responsibilities dynamically / Wrapper
- **Real World: Notifier** example (SMS, Email, Slack)





Quiz

Q: TV remote controlling different TV brands. Which pattern?

Ans: Bridge

Q: Windows Explorer tree is example of?

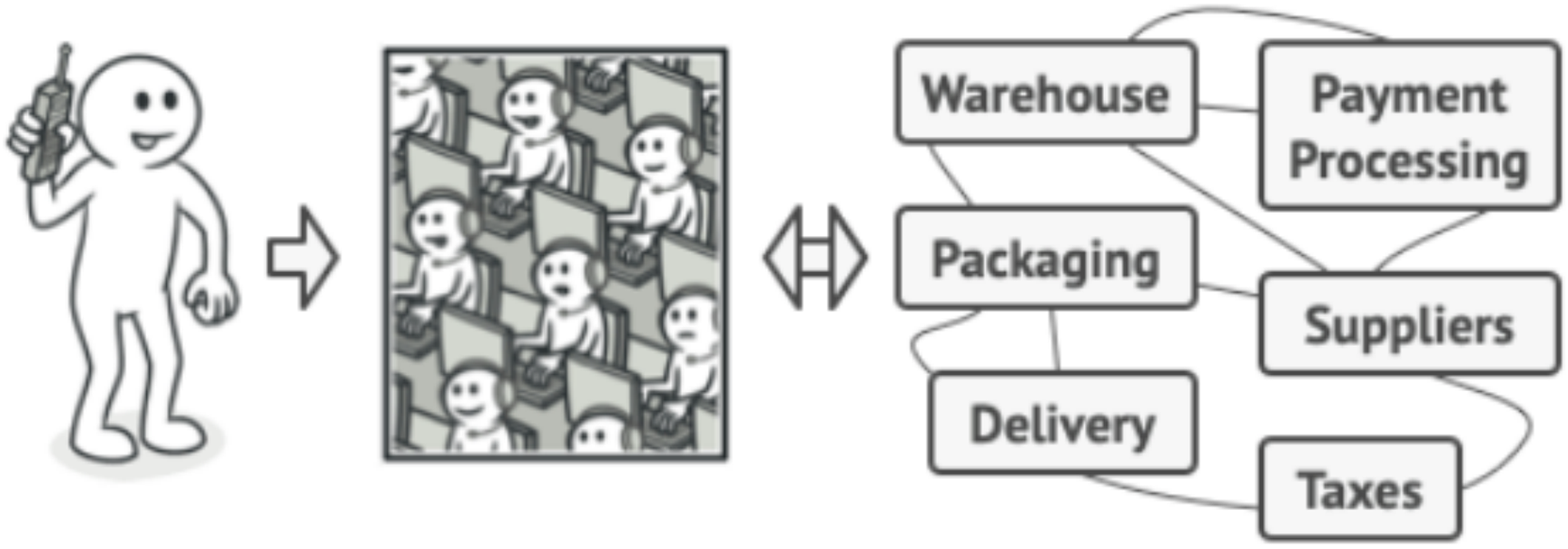
Ans: Composite

Q: Using a USB-to-HDMI connector is an example of?

Ans: Adapter

Facade

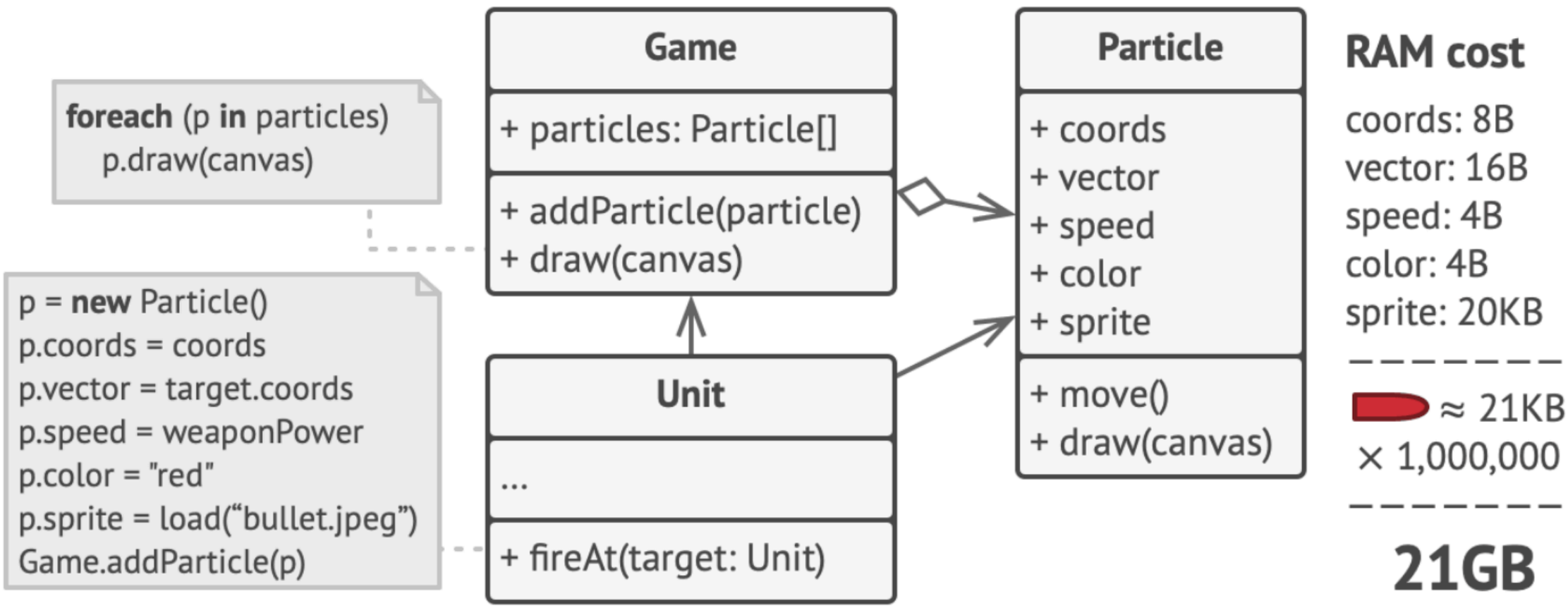
- Simplify complex subsystem
- **Real World:** Home Theater system → one remote

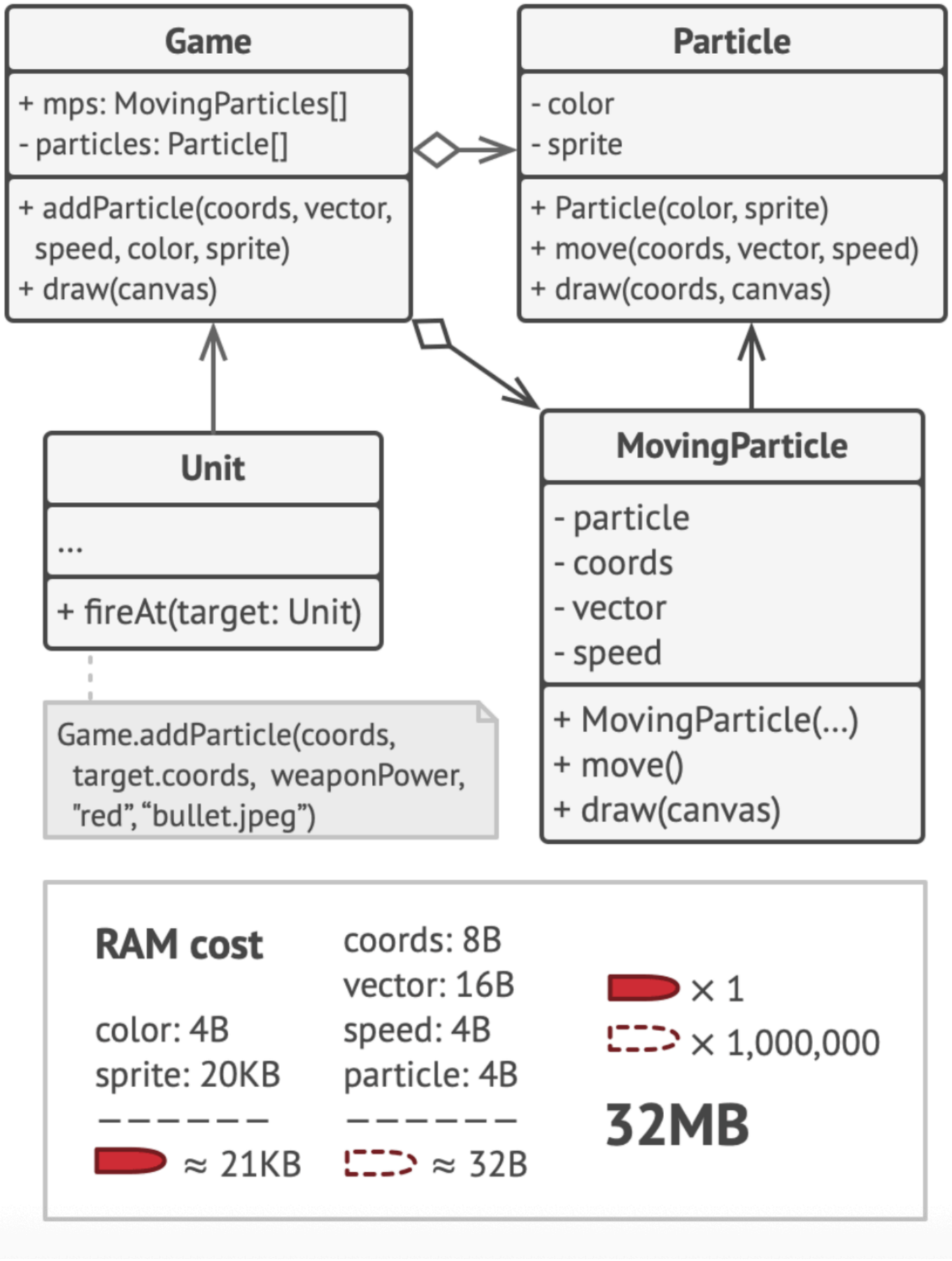


Placing orders by phone.

Flyweight

- Share data across objects → memory optimization, also known as: Cache
- **Real World:** Characters in a text editor




Game
+ mps: MovingParticles[]
- particles: Particle[]
+ addParticle(coords, vector, speed, color, sprite)
+ draw(canvas)
Particle
- color
- sprite
+ Particle(color, sprite)
+ move(coords, vector, speed)
+ draw(coords, canvas)
Unit
...
+ fireAt(target: Unit)
Game.addParticle(coords, target.coords, weaponPower, "red", "bullet.jpeg")
MovingParticle
- particle
- coords
- vector
- speed
+ MovingParticle(...)
+ move()
+ draw(canvas)
RAM cost
color: 4B
sprite: 20KB
≈ 21KB
coords: 8B
vector: 16B
speed: 4B
particle: 4B
≈ 32B
× 1
× 1,000,000
32MB

Proxy

- Substitute object to control access
- Types of Proxy:
  - **Virtual** → delay heavy object creation
  - **Protection** → control access (permissions)
  - **Remote** → represent object in another system/network
  - **Smart** → add caching, logging, security
- **Real World:** Virtual proxy for image loading, Credit card proxy to bank

Quiz

Q: Which pattern optimizes memory in large-scale apps?

Ans: Flyweight

Q: Netflix button to play movie (instead of multiple service calls). Which pattern?

Ans: Facade

Q: Which pattern delays loading heavy objects until needed?

Ans: Proxy

Problem 3: Media Player Adapter

You are building a media player that should play both **MP3** and **MP4** files.

- The existing `MP3Player` can play only MP3.
- A new `MP4Player` is available but has a different interface (`playMp4()`).
- Use an **Adapter Pattern** to integrate MP4 into your `MediaPlayer` interface.

Task:

1. Define `MediaPlayer` interface with `play(filename)`.
2. Implement `MP3Player` normally.
3. Wrap `MP4Player` using `MediaAdapter` so it can be used as `MediaPlayer`.
4. Test with both MP3 and MP4 files.

Problem 4: Coffee Decorator

You need to design a Coffee ordering system.

- Base product: `SimpleCoffee`.
- Add-ons: Milk, Sugar, Whipped Cream.
- Customers can order coffee with multiple add-ons.

Task:

1. Create a `Coffee` interface with `getDescription()` and `getCost()`.
2. Implement `SimpleCoffee`.
3. Implement decorators (`MilkDecorator`, `SugarDecorator`, etc.).
4. In `main()`, build coffee with different combinations and print description + cost.

Prep Assignment

📝 Recall from Today

- Pick 1 Creational + 1 Structural pattern.
- Write a real-world analogy for each.

🔮 Look Ahead

- Find a situation where:
  - One change affects many others.
  - You switch approach depending on context.
  - A button triggers an action .

🌐 Mini Research

- Identify one design pattern used in:
  - Java libraries / Android / React / any framework.