Perfect — let's take a **concrete, real-world example** with a **problem-first** approach and then show how the **Strategy Pattern** offers a clean solution. This is the best way to understand its real value.

🧩 Problem Scenario: Payment Processing System

🎯 Requirement:

You are building an **e-commerce application** that supports multiple payment methods:

- Credit Card
- PayPal
- UPI

The application should be able to process any of these payments based on **user selection at runtime**.

❌ Naive Implementation (Problem)

You might write code like this in your `PaymentService`:

```
public class PaymentService {
    public void processPayment(String paymentType) {
        if (paymentType.equals("CREDIT_CARD")) {
            // process credit card
        } else if (paymentType.equals("PAYPAL")) {
            // process PayPal
        } else if (paymentType.equals("UPI")) {
            // process UPI
        }
    }
}
```

😖 What's Wrong Here?

- 🚫 Violates **Open/Closed Principle**: adding new payment types = modify existing code.
- 🚫 Logic is **tightly coupled**.
- 🚫 Hard to **unit test** and maintain.
- 🚫 Long and **bloated** if-else chains.

✅ Solution: Strategy Pattern

We use Strategy to **decouple the payment processing logic** from the `PaymentService`. Each payment method is encapsulated as a separate strategy.

🧠 Strategy Pattern Breakdown

🔷 Step 1: Define the Strategy Interface

```
public interface PaymentStrategy {
    void pay(double amount);
}
```

🔷 Step 2: Implement Different Payment Strategies

```
public class CreditCardPayment implements PaymentStrategy {
    public void pay(double amount) {
        // logic to process credit card payment
    }
}

public class PayPalPayment implements PaymentStrategy {
    public void pay(double amount) {
        // logic to process PayPal payment
    }
}

public class UPIPayment implements PaymentStrategy {
    public void pay(double amount) {
        // logic to process UPI payment
    }
}
```

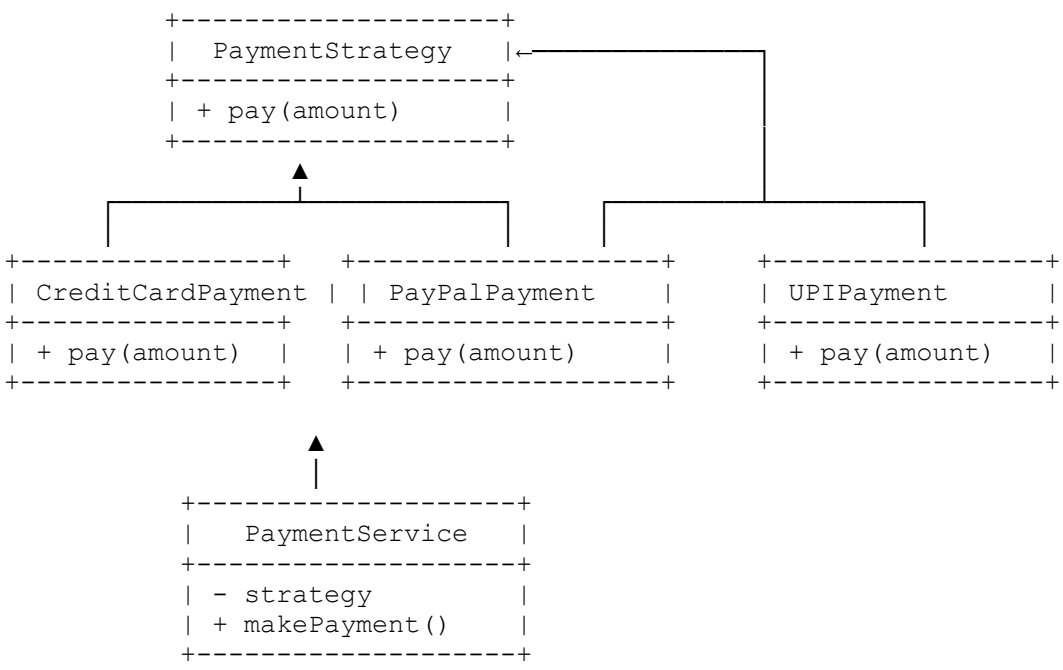
🔷 Step 3: Context Class Delegates the Strategy

```
public class PaymentService {
    private PaymentStrategy paymentStrategy;

    public PaymentService(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void makePayment(double amount) {
        paymentStrategy.pay(amount);
    }
}
```

🏗️ Class Diagram



🧪 Runtime Behavior

```
PaymentService service = new PaymentService(new PayPalPayment());
service.makePayment(250.0); // Pay using PayPal

service = new PaymentService(new UPIPayment());
service.makePayment(300.0); // Now pay using UPI
```

✔️ Easy to change payment method
✔️ No need to touch `PaymentService` for new methods
✔️ Open for extension, closed for modification

✅ Benefits Gained

Benefit	Explanation
✅ Reusability	Each payment strategy is reusable and independent
✅ Flexibility	Easily switch behavior at runtime
✅ Extensibility	Add new payment types without touching existing code
✅ Cleaner Code	Removes conditional logic from `PaymentService`

📌 Summary

Before (Problem)	After (Strategy)
Tightly coupled logic	Decoupled and interchangeable behaviors
Hard to maintain	Easy to extend
`if-else` chains	Polymorphism and interface-based delegation
Risk of bugs on change	Isolated changes per strategy class

Would you like a **visual image** of this example with arrows and boxes? I can create a clean diagram to make this even clearer.