Strategy Design Pattern

Problem Statement

You are building an e-commerce platform that initially had **Credit Card Payments**. As the platform expanded, the team added support for **PayPal**, **Stripe**, **Bitcoin**, and **Apple Pay**.

Initially, all payment methods were implemented inside one class: `PaymentProcessor`. As new payment options were added, this class became:

- Large and difficult to maintain
- Prone to bugs when changes were made
- Hard to extend without modifying the original logic

Result: The team faced merge conflicts, slow reviews, and high maintenance effort.

Core Requirements

1. The system should support **multiple payment methods**.
2. It should be **easy to add new strategies** without modifying existing code.
3. **Maintainability** and **extensibility** are top priorities.

Strategy Pattern: Simple Definition

Strategy Pattern is a **behavioral design pattern** that:

- Allows you to define a **family of algorithms** (or behaviors)
- Encapsulates each algorithm inside its own class
- Enables the **interchangeability** of these algorithms at runtime

Goal-Oriented Client Code

Before diving into the full structure, understand how the client is expected to use the system:

```
const processor = new Payment(new CardPayment());
processor.process(100);

processor.setStrategy(new UPIPayment());
processor.process(200);
```

Key Idea: You can swap strategies at runtime without changing the logic in `Payment`.

Code Implementation

Step 1: Define a `PaymentStrategy` Interface

```
interface PaymentStrategy {
    performPayment(amount: number): void;
}
```

This acts as a **contract** that all payment strategies must follow.

Step 2: Create Concrete Strategies

```
class CardPayment implements PaymentStrategy {
    performPayment(amount: number): void {
        console.log(`Rs.${amount} Payment performed using Card`);
    }
}

class UPIPayment implements PaymentStrategy {
    performPayment(amount: number): void {
        console.log(`Rs.${amount} Payment performed using UPI`);
    }
}
```

Each class has its own logic for handling payments.

Step 3: Build the Context Class

```
class Payment {
    paymentStrategy: PaymentStrategy;

    constructor(paymentStrategy: PaymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    setStrategy(paymentStrategy: PaymentStrategy): void {
        this.paymentStrategy = paymentStrategy;
    }

    process(amount: number): void {
        this.paymentStrategy.performPayment(amount);
    }
}
```

This is the **context** that uses any payment strategy object.

Step 4: Use the Pattern

```
const processor = new Payment(new CardPayment());
processor.process(100); // Card

processor.setStrategy(new UPIPayment());
processor.process(200); // UPI
```

Class Diagram Overview

Participants:

- `Payment` → Uses the strategy
- `PaymentStrategy` → Interface
- `CardPayment`, `UPIPayment`, etc. → Concrete implementations

Real-World Applications

- **Payment Gateways**: Card, UPI, PayPal, Stripe
- **Authentication**: Google, Facebook, Email login
- **Shipping Options**: Standard, Express, Overnight
- **Sorting Algorithms**: Quick Sort, Merge Sort, Bubble Sort
- **Pricing Logic**: Normal, Discounted, Premium pricing

Benefits

1. Can **switch algorithms** at runtime without affecting client code.
2. Encourages **Open/Closed Principle**: Add new strategies without modifying existing logic.
3. Reduces **code duplication** by separating algorithm implementations.
4. Promotes **composition over inheritance**.

Limitations

1. If only 2–3 strategies are required, it might be **overkill**.
2. For simple variations, **functional programming** (e.g., passing a function as a parameter) might be better.
3. **Too many classes**: Every new strategy adds a new file or class.

What is Tight Coupling?

- When one class is **heavily dependent** on another.
- You cannot change one class without affecting the other.
- Difficult to test and extend.

Example:

```
class Engine {
  start() {
    console.log("Engine started");
  }
}

class Car {
  private engine = new Engine(); // Tight Coupling
  drive() {
    this.engine.start();
    console.log("Car is driving");
  }
}
```

- **Problem**: Car is directly creating and depending on Engine.

What is Loose Coupling?

- Classes are **connected through abstraction** (like interfaces).
- You can change components without modifying the whole system.
- Easier to test, extend, and maintain

```
interface Vehicle {
  start(): void;
}

class Car implements Vehicle {
  start() {
    console.log("Car is starting");
  }
}

class Bike implements Vehicle {
  start() {
    console.log("Bike is starting");
  }
}

class Driver {
  constructor(private vehicle: Vehicle) {}

  drive() {
    this.vehicle.start();
    console.log("Driving...");
  }
}
```