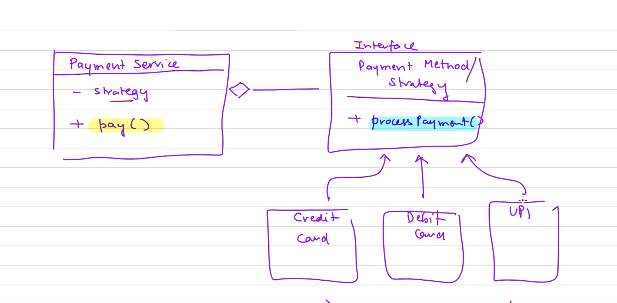


**Problem statement.**

1) So let's say we want to build a payment system where users can pay using different methods like credit cards, PayPal or even UPI payments.

2) So without the strategy pattern, what we typically we may do is we can have a function where we will put if else conditions to check what kind of payment method has been supplied, and accordingly, we can have our own business logic to handle that case.

3) But this often leads to less mentally less maintainable and less flexible code.And also it's going to violate the open closed principle.



**🔹 What is the Strategy Pattern?**

The **Strategy Pattern** is a behavioral design pattern that allows you to define a family of algorithms, encapsulate each one, and make them interchangeable at runtime.

It follows the **Open/Closed Principle** — you can add new strategies without modifying existing code.

**✅ Real-World Use Cases of Strategy Pattern**

| **Use Case** | **Example** |
| --- | --- |
| ✅ **Validation logic** | Apply different validators (e.g., email, phone) to an input |
| ✅ **Payment processing** | Choose between credit card, PayPal, or UPI payment strategies |
| ✅ **Sorting or filtering** | Apply different sorting strategies for products or data |
| ✅ **Compression** | Use ZIP, GZIP, or custom algorithms interchangeably |
| ✅ **Authentication** | Use LDAP, JWT, OAuth2 strategies for login |
| ✅ **Discount calculation** | Use strategies for seasonal, coupon, or membership discounts |
| ✅ **Routing logic** | Choose routing strategies in logistics or workflow engines |
| ✅ Advantages of Strategy Pattern  | **Advantage** | **Description** | | --- | --- | | ✅ **Open/Closed Principle** | Easily add new strategies without touching existing code | | ✅ **Separation of Concerns** | Keeps algorithms separate from the code that uses them | | ✅ **Interchangeable logic at runtime** | Swap strategies dynamically (e.g., user preference, config) | | ✅ **Improves testability** | Each strategy is isolated and easy to unit test | | ✅ **Reduces conditional logic** | Eliminates complex if-else or switch-case blocks |  ❌ Disadvantages of Strategy Pattern  | **Disadvantage** | **Description** | | --- | --- | | ❌ **Increased number of classes** | Each strategy requires a new class, which may clutter the codebase | | ❌ **Client must know which strategy to use** | Unless managed via factory/config, clients must choose the right strategy | | ❌ **Runtime misconfiguration risk** | Choosing the wrong strategy dynamically can cause unexpected behavior | | ❌ **Too granular for simple use cases** | For very simple logic, strategy pattern may feel over-engineered |  ✅ When to Use Strategy Pattern ✅ Use it **when**:   * You have multiple algorithms doing the same task differently * You want to isolate changing behavior in separate classes * You want to avoid long if-else/switch-case logic * You expect to add new variants of the algorithm often  ❌ When NOT to Use Strategy Pattern ❌ Avoid it **when**:   * There's only one or two simple logic branches * You don't expect the logic to change or grow * Strategy selection is hard to manage dynamically * You're better off with functional programming (e.g., passing lambdas)  🔁 Strategy Pattern vs Alternatives  | **Pattern** | **When to Use** | | --- | --- | | **Strategy** | Swappable algorithms/behaviors | | **State** | Behavior changes based on object state | | **Template Method** | Base algorithm is fixed but steps are customizable | | **Command** | Queue or execute operations on demand (e.g., UI, undo) |  ✅ Summary Table  | **Feature** | **Strategy Pattern** | | --- | --- | | Type | Behavioral Design Pattern | | Based on | Polymorphism (interface-based delegation) | | Key Benefit | Swappable and reusable behaviors | | Good For | Multiple interchangeable algorithms | | Not Ideal When | Logic is trivial or doesn’t change | |  |

// Strategy interface

interface PaymentStrategy {

void pay(double amount);

}

// Concrete strategies

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card");

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid " + amount + " using PayPal");

}

}

// Context

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amount) {

strategy.pay(amount);

}

}