Strategy Pattern

Resources

- Video → 41. All Behavioral Design Patterns | Strategy, Observer, State, Template, Command, Visitor, Memento
- Video → 2. Strategy Design Pattern explanation (Hindi) | LLD System Design | Design pattern in Java

Definition

> ***The Strategy pattern is a behavioral design pattern that defines multiple algorithms, encapsulates their logic in dedicated classes, and enables changing an algorithm's behavior at runtime. It's particularly useful when you have multiple ways to perform a task and want to choose the approach dynamically.***

Popular Real-life Examples

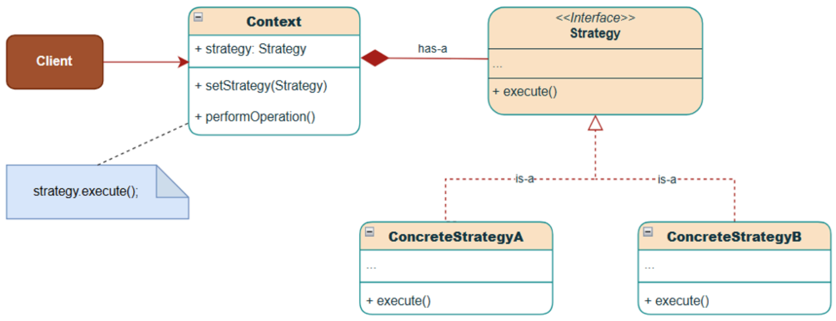
- **Courier Service** → Shipping Cost Calculation - Discount for Premium members, Flat fee, Distance-based based and weight-based computations.
- **Shopping Cart** → Payment Options - CrediCard, PayPal, UPI, Cash, etc.
- **Vehicle Manufacturing** → Different cars(like SUVs, EVs, etc) require different drive modes.

Problems without Strategy Pattern

Refer Strategy Pattern | Implementation section below for a better understanding.

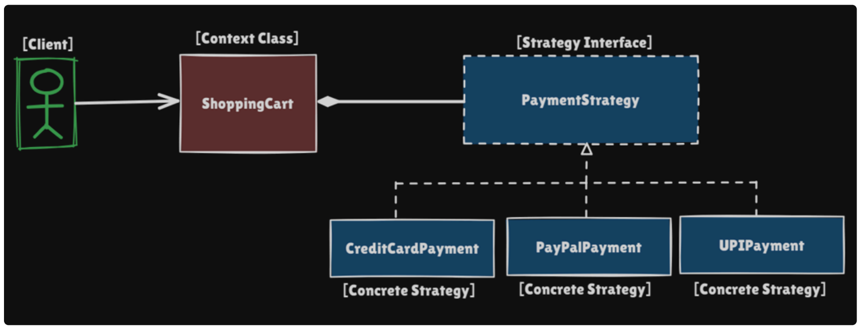
1. Massive Conditional Blocks leading to bloated classes
2. Violation of the Open/Closed Principle and Single Responsibility Violation
3. Code Duplication
4. Tight Coupling
5. Testing Complexity

Class Diagram



Structure of Strategy Pattern

Let's understand the Strategy Pattern using the Payment method example:



- **Strategy Interface(`PaymentStrategy`)**: Defines a common interface(contract) that all concrete strategies must implement.
- **Concrete Strategies(`CreditCardPayment` , `PayPalPayment` , `UPIPayment`)**: Different implementations of the strategy interface, each representing a specific algorithm or approach.
- **Context Class(`ShoppingCart`)**: The class that uses a strategy. It maintains a reference to a strategy object and delegates work to it. Context uses Concrete Strategies to choose the required behavior among the available family of algorithms at runtime.

Implementation

1. Different Drive Modes in a Vehicle

Without Strategy Pattern

```java
public class Vehicle {

    public void drive() {
        System.out.print("\n" + this.getClass().getSimpleName() + ":
");
        System.out.println("Driving Capability: Normal");
    }
}
```

```java
public class SportsVehicle extends Vehicle {

    // Overriding the drive method to provide specific behavior for
sports vehicles
    public void drive() {
        System.out.print("\n" + this.getClass().getSimpleName() + ": ");
        System.out.println("Driving Capability: Sports");
    }
}
```

```java
public class OffRoadVehicle extends Vehicle {

    // Overriding the drive method to provide specific behavior
    public void drive() {
        System.out.print("\n" + this.getClass().getSimpleName() + ": ");
        System.out.println("Driving Capability: Sports"); // code duplication
        // As sports drive mode is not available in the parent class, we need to override it and implement
        // the specific behavior for all new vehicle types that require sports drive mode
    }

}
```

```java
public class PassengerVehicle extends Vehicle {

    // Reusing the existing drive method from the parent class
    // Driving Capability: Normal
    // No new implementation required
}
```

```java
public class Demo {
    public static void main(String[] args) {
        System.out.println("Vehicle Drive Modes: Problem Demo");
        Vehicle vehicle = new Vehicle();

        // Sports vehicle - sports drive mode
        vehicle = new SportsVehicle();
        vehicle.drive();

        // Off-road vehicle - sports drive mode
        vehicle = new OffRoadVehicle();
        vehicle.drive();

        // Passenger vehicle - normal drive mode
        vehicle = new PassengerVehicle();
        vehicle.drive();
    }
}
```

With Strategy Pattern

```java
// Strategy interface - defines the contract for drive behavior
public interface DriveStrategy {
    public void drive();
}
```

```java
// Concrete strategy for normal drive mode
public class NormalDrive implements DriveStrategy {
    @Override
    public void drive() {
        System.out.println("Driving Capability: Normal");
    }
}
// Concrete strategy for sports drive mode
public class SportsDrive implements DriveStrategy {
    @Override
    public void drive() {
        System.out.println("Driving Capability: Sports");
    }
}
// Concrete strategy for electric drive mode
```

```java
public class EVDrive implements DriveStrategy {
    @Override
    public void drive() {
        System.out.println("Driving Capability: Electric");
    }
}
```

```java
// Context class - holds a reference to a strategy object
public class Vehicle {
    DriveStrategy driveStrategy;

    // constructor injection
    public Vehicle(DriveStrategy driveStrategy) {
        this.driveStrategy = driveStrategy;
    }

    public void drive() {
        System.out.print("\n" + this.getClass().getSimpleName() + ": ");
        driveStrategy.drive();
    }
}
// Concrete context subclass
public class GoodsVehicle extends Vehicle {

    public GoodsVehicle(DriveStrategy driveStrategy) {
        super(driveStrategy);
    }
}
// Concrete context subclass
public class SportsVehicle extends Vehicle {

    public SportsVehicle(DriveStrategy driveStrategy) {
        super(driveStrategy);
    }
}
// Concrete context subclass
public class OffRoadVehicle extends Vehicle {

    OffRoadVehicle(DriveStrategy driveStrategy) {
        super(driveStrategy);
    }
}
// Concrete context subclass
public class HybridVehicle extends Vehicle {

    public HybridVehicle(DriveStrategy driveStrategy) {
        super(driveStrategy);
    }
}
```

```java
// Client Code
public class Demo {
    public static void main(String[] args) {
        System.out.println("###### Strategy Design Pattern ######");
        System.out.println("###### Example: Vehicle Drive Modes ######");

        Vehicle vehicle = new SportsVehicle(new SportsDrive());
        vehicle.drive();

        vehicle = new GoodsVehicle(new NormalDrive());
        vehicle.drive();

        vehicle = new HybridVehicle(new EVDrive());
        vehicle.drive();

        vehicle = new GoodsVehicle(new NormalDrive());
        vehicle.drive();
    }
}
```

2. Shopping Cart Payment Methods

Without Strategy Pattern

```java
// A simple payment processor class - bloated with payment logic
public class PaymentProcessor {
    public void processPayment(String type, double amount) {
        switch (type) {
            case "credit_card" -> {
                // x lines of credit card logic
                System.out.println("Paid $" + amount + " using credit card");
            }
            case "paypal" -> {
                // y lines of PayPal logic
                System.out.println("Paid $" + amount + " using PayPal");
            }
            case "net_banking" -> {
                // z lines of bank transfer logic
                System.out.println("Paid $" + amount + " using bank transfer");
            }
            case "cash" -> {
                // 10 lines of cash on delivery logic
                System.out.println("Paid $" + amount + " using cash");
            }
            default -> throw new IllegalStateException("Unexpected value: " + type);
        }
        // Adding another payment method(crypto) requires modifying this class
        // This keeps growing with each new payment method
        // bad design
    }
}
```

```java
public class Demo {
    public static void main(String[] args) {
        System.out.println("Payment Processor: Problem Demo");
        PaymentProcessor processor = new PaymentProcessor();
        processor.processPayment("credit_card", 100);
        processor.processPayment("paypal", 200);
        processor.processPayment("net_banking", 300);
        processor.processPayment("cash", 400);
    }
}
```

With Strategy Pattern

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

```java
// Concrete strategy - for credit card payment
public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using credit card ending in "
                + cardNumber.substring(cardNumber.length() - 4));
    }
}
```

```java
// Concrete strategy - for PayPal payment
public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal account
" + email);
    }
}
```

```java
// Concrete strategy - for UPI payment
public class UPIPayment implements PaymentStrategy {
    private String upiId;

    public UPIPayment(String upiId) {
        this.upiId = upiId;
    }

    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using UPI ID " +
upiId);
    }
}
```

```java
// Context class - holds reference to a strategy object
public class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout(double amount) {

System.out.print(this.paymentStrategy.getClass().getSimpleName() + ":
");
        paymentStrategy.pay(amount);
    }
}
```

```java
// Client code - to simulate payment processing
public class Demo {
    public static void main(String[] args) {
        System.out.println("###### Strategy Design Pattern ######");
        System.out.println("###### Example: Payment Processor
######");

        // Create a shopping cart and set payment strategy
        ShoppingCart cart = new ShoppingCart();

        // Choosing payment behavior at runtime
        cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-
3456"));
        cart.checkout(100.0);
        cart.setPaymentStrategy(new
PayPalPayment("johndoe@example.com"));
        cart.checkout(200.0);
        cart.setPaymentStrategy(new UPIPayment("9988776655@ybl"));
        cart.checkout(300.0);
        // Adding another payment method(crypto) is as simple as
adding a new strategy class
        // No need to modify existing code - good design
        // cart.setPaymentStrategy(new CryptoPayment("BTC"));
        // cart.checkout(400.0);
    }
}
```