Strategy Design Pattern

The Strategy Design Pattern is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

Customer Requirement:

Imagine you are developing a payment processing system for an e-commerce platform. The system needs to support multiple payment methods like credit card, PayPal, and Bitcoin. Each payment method has its own processing logic. The customer wants to be able to select the payment method at runtime without changing the existing codebase every time a new payment method is introduced.

Solution: Strategy Pattern

The Strategy Pattern fits this scenario perfectly. It allows you to define a family of payment algorithms, encapsulate each one, and make them interchangeable. The client can select which payment method to use at runtime, and the system will execute the corresponding algorithm.

Components of the Strategy Pattern:

1. Context: Maintains a reference to a strategy object. Interacts with the strategy to execute the desired behavior.

2. Strategy Interface: Defines the interface common to all supported algorithms.

3. Concrete Strategies: Implement the algorithm defined by the Strategy interface.

Example: Payment Processing System

1. Strategy Interface (PaymentStrategy):

```java
public interface PaymentStrategy {
    void pay(int amount);

}
```

2. Concrete Strategies (CreditCardPayment, PayPalPayment, BitcoinPayment):

```java
public class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    private String cardHolderName;



    public CreditCardPayment(String cardNumber, String cardHolderName) {

        this.cardNumber = cardNumber;

        this.cardHolderName = cardHolderName;

    }



    @Override

    public void pay(int amount) {

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

    }

}



public class PayPalPayment implements PaymentStrategy {

    private String email;



    public PayPalPayment(String email) {

        this.email = email;

    }
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}

public class BitcoinPayment implements PaymentStrategy {
    private String walletAddress;

    public BitcoinPayment(String walletAddress) {
        this.walletAddress = walletAddress;
    }

    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Bitcoin.");
    }
}
```

3. Context (PaymentContext):

```java
public class PaymentContext {
    private PaymentStrategy paymentStrategy;

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

    public void executePayment(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

4. Client Code:

```java
public class StrategyPatternDemo {
    public static void main(String[] args) {
        PaymentContext paymentContext = new PaymentContext();

        // Pay using Credit Card
        paymentContext.setPaymentStrategy(new CreditCardPayment("123456789", "John Doe"));
        paymentContext.executePayment(100);

        // Pay using PayPal
        paymentContext.setPaymentStrategy(new PayPalPayment("john@example.com"));
        paymentContext.executePayment(200);

        // Pay using Bitcoin
        paymentContext.setPaymentStrategy(new BitcoinPayment("1FzWLkZ7gV4F3bZXzDR4yZ77Q1jqXPfF2T"));
        paymentContext.executePayment(300);
    }
}
```

How It Works:

- Context (PaymentContext): The PaymentContext class is configured with a PaymentStrategy object. It simply delegates the payment processing to the strategy object.
- Strategy Interface (PaymentStrategy): The PaymentStrategy interface defines the pay method that all concrete strategies must implement.
- Concrete Strategies (CreditCardPayment, PayPalPayment, BitcoinPayment): These classes implement the PaymentStrategy interface and provide the specific implementation for each payment method.

Why It is a Behavioral Design Pattern:

The Strategy Pattern is a behavioral design pattern because it focuses on the behavior of the objects. It allows the behavior (algorithm) of a method to be selected at runtime. The pattern provides a way to configure a class with one of many possible behaviors (strategies).

Benefits of the Strategy Pattern:

1. Flexibility: The Strategy Pattern allows you to change the algorithm or behavior of a class at runtime.

2. Encapsulation: It encapsulates the algorithm, separating it from the context, which reduces the complexity of the context.

3. Extensibility: Adding new strategies (algorithms) is easy because you do not need to modify the existing context class.

When to Use the Strategy Pattern:

- When you have a family of algorithms, and you want to make them interchangeable.
- When you need different variants of an algorithm that can be selected at runtime.

- When you want to avoid conditional statements for selecting the algorithm.