





Strategy Design Pattern

Strategy Pattern Guide  | Pick the Best Algorithm Dynamically 

Topic Tags:

System Design


LLD

 Github Codes Link: <https://github.com/aryan-0077/CWA-LowLevelDesignCode>

Strategy Design Pattern: A Real-Life Example in Software Engineering

When it comes to software development, flexibility and scalability are key factors in building systems that can evolve over time without becoming unmanageable. The Strategy Design Pattern is a powerful tool that enables software engineers to achieve just that by allowing different algorithms or behaviors to be selected dynamically at runtime.

Introduction to the Strategy Pattern

In simple terms, the Strategy Pattern allows you to define a family of algorithms or behaviors, and choose the one to use during runtime. It is like having a toolbox  where you can pick the best tool (or strategy) for the task at hand. This approach avoids hardcoding multiple behaviors into one class and promotes flexibility by separating the behavior logic into different classes.

Why is it Called the Strategy Pattern? 🏆

The name Strategy comes from the idea of different strategies to solve the same problem (in this case, processing payments). Each strategy encapsulates a different way to process payments, and we can switch between them dynamically based on user input or system requirements. This makes the system more flexible and easier to extend.

Real-Life Scenario: Payment Processing in E-commerce 🛒💳

Imagine you're developing an e-commerce platform where users can make purchases using various payment methods like Credit Cards, PayPal, or Cryptocurrency. Each payment method has its own unique processing logic.

Without the Strategy Pattern, you'd likely have a large, monolithic class with many if-else or switch statements, checking for the payment method and executing the specific logic for each one. But what happens when you need to add another payment method (like Apple Pay or Stripe)? It becomes a nightmare to manage and extend. 🤯

The Traditional Approach: Payment Processing 💳

Step 1: The Problem – Different Payment Methods 🧩

We start with a PaymentProcessor class. This class will check the payment method (Credit Card, PayPal, or Crypto) and handle the payment accordingly.

Now, we don't want to keep writing a bunch of different methods for each payment method, so we try using an if-else block to determine the payment method and process it. But we need to change this block every time we add a new payment method. Let me show you how it looks:

Code Example: Traditional Payment Processor 🖥️

Java

```
1 public class PaymentProcessor {
2     // This method will process payment based on payment method type
3     public void processPayment(String paymentMethod) {
4         if (paymentMethod.equals("CreditCard")) {
5             // Process Credit Card payment
6             System.out.println("Processing credit card payment...");
7         } else if (paymentMethod.equals("PayPal")) {
8             // Process PayPal payment
9             System.out.println("Processing PayPal payment...");
10        } else if (paymentMethod.equals("Crypto")) {
11            // Process Crypto payment
12            System.out.println("Processing crypto payment...");
```

```
13     } else {  
14         // If an unsupported payment method is entered  
15         System.out.println("Payment method not supported.");  
16     }  
17 }  
18 }
```

How It Works:

- We have a single method called `processPayment()`.
- Inside the method, we check what type of payment method the user has selected using `if-else` statements.
- For each payment method (Credit Card, PayPal, or Crypto), we print a message like "Processing credit card payment..."

What Happens When We Want to Add a New Payment Method? 🤔

Let's say you now want to add a new payment method, like Stripe.

If we were using the above approach, we'd have to modify the `processPayment()` method like this:

Java

```
1 public class PaymentProcessor {  
2     // This method will process payment based on payment method type  
3     public void processPayment(String paymentMethod) {  
4         if (paymentMethod.equals("CreditCard")) {  
5             // Process Credit Card payment  
6             System.out.println("Processing credit card payment...");  
7         } else if (paymentMethod.equals("PayPal")) {  
8             // Process PayPal payment  
9             System.out.println("Processing PayPal payment...");  
10        } else if (paymentMethod.equals("Crypto")) {  
11            // Process Crypto payment  
12            System.out.println("Processing crypto payment...");  
13        } else if (paymentMethod.equals("Stripe")) { // New method added  
14            // Process Stripe payment  
15            System.out.println("Processing Stripe payment...");  
16        } else {  
17            // If an unsupported payment method is entered  
18            System.out.println("Payment method not supported.");  
19        }  
20    }  
21 }
```

```
20     }  
21 }
```

What's Wrong with This? 🤔

Here's where the problem comes in:

- Adding new payment methods: Every time you want to add a new payment method, you have to go into the `processPayment()` method and modify the code.
- Code duplication: We keep repeating similar blocks of code for each payment method, which can get messy when we add more and more methods.
- Scalability issues: As you keep adding new methods (Stripe, Google Pay, Apple Pay, etc.), this if-else block becomes harder to maintain and less flexible. Imagine what happens when you have 20 payment methods. The code gets huge and difficult to read.

Step 2: Slight Improvement Using Interfaces – PaymentProcessor Class 🔄

In Step 1, we had a monolithic method that handled every payment method type with an if-else block. The problem with that approach is that we had to modify the method each time we added a new payment method. This leads to code duplication and hard-to-maintain code. In Step 2, we make a slight improvement by using interfaces. We will define a `PaymentMethod` interface that each payment method will implement. This is a good improvement, but we will still have to modify the `PaymentProcessor` class when we add a new payment method.

Let's take a look at how we can improve the code by using interfaces.

Step 2: Slight Improvement Using Interfaces

PaymentMethod Interface

Instead of hardcoding the payment methods inside the `PaymentProcessor` class, we can define an interface `PaymentMethod` with a method `processPayment()`. All payment methods will implement this interface and provide their own implementation of `processPayment()`.

Java

```
1 // PaymentMethod interface (defines the common method for all payment  
2 public interface PaymentMethod {  
3     void processPayment(); // Abstract method for processing payments  
4 }
```

Now, let's create separate classes for each payment method, and each class will implement the `PaymentMethod` interface:

Concrete Payment Method Classes

Java

```
1 public class CreditCardPayment implements PaymentMethod {
2     public void processPayment() {
3         System.out.println("Processing credit card payment...");
4     }
5 }
6
7 public class PayPalPayment implements PaymentMethod {
8     public void processPayment() {
9         System.out.println("Processing PayPal payment...");
10    }
11 }
12
13 public class CryptoPayment implements PaymentMethod {
14     public void processPayment() {
15         System.out.println("Processing crypto payment...");
16     }
17 }
18
19 public class StripePayment implements PaymentMethod {
20     public void processPayment() {
21         System.out.println("Processing Stripe payment...");
22     }
23 }
```

PaymentProcessor Class in Step 2 📌

Now that we have modularized the payment methods into separate classes, the next step is to make the PaymentProcessor class work with these payment strategy classes. We can pass the payment method as a parameter to the PaymentProcessor class.

However, here's the catch: while this is better, we still need to modify the PaymentProcessor class every time a new payment method is added.

Java

```
1 public class PaymentProcessor {
2     // This method processes payment based on the payment method ty
3     public void processPayment(String paymentMethod) {
4         if (paymentMethod.equals("CreditCard")) {
5             CreditCardPayment creditCard = new CreditCardPayment();
6             creditCard.processPayment(); // Process Credit Card pay
7         } else if (paymentMethod.equals("PayPal")) {
8             PayPalPayment payPal = new PayPalPayment();
9             payPal.processPayment(); // Process PayPal payment
```

```
10         } else if (paymentMethod.equals("Crypto")) {
11             CryptoPayment crypto = new CryptoPayment();
12             crypto.processPayment(); // Process Crypto payment
13         } else if (paymentMethod.equals("Stripe")) {
14             StripePayment stripe = new StripePayment();
15             stripe.processPayment(); // Process Stripe payment
16         } else {
17             System.out.println("Payment method not supported.");
18         }
19     }
20 }
```

What's the Issue Now? ⚠️

Even though we've moved the payment logic to individual classes (for each payment method), we still have to modify the PaymentProcessor class every time we introduce a new payment method. This is because we are still checking the payment method inside the processPayment() method and manually creating instances of the corresponding class.

Example: Adding a New Payment Method (Apple Pay)

To add a new payment method, like Apple Pay, we would need to:

1. Create a new strategy class for Apple Pay.
2. Modify the PaymentProcessor class and add a new else if block for Apple Pay.

Here's how we would do it:

Java

```
1 public class PaymentProcessor {
2     // This method processes payment based on the payment method type
3     public void processPayment(String paymentMethod) {
4         if (paymentMethod.equals("CreditCard")) {
5             CreditCardPayment creditCard = new CreditCardPayment();
6             creditCard.processPayment(); // Process Credit Card payment
7         } else if (paymentMethod.equals("PayPal")) {
8             PayPalPayment paypal = new PayPalPayment();
9             paypal.processPayment(); // Process PayPal payment
10        } else if (paymentMethod.equals("Crypto")) {
11            CryptoPayment crypto = new CryptoPayment();
12            crypto.processPayment(); // Process Crypto payment
13        } else if (paymentMethod.equals("Stripe")) {
14            StripePayment stripe = new StripePayment();
15            stripe.processPayment(); // Process Stripe payment
16        } else if (paymentMethod.equals("ApplePay")) { // New payment m
```

```
17     ApplePayPayment applePay = new ApplePayPayment();
18     applePay.processPayment(); // Process Apple Pay payment
19 } else {
20     System.out.println("Payment method not supported.");
21 }
22 }
23 }
```

Why Is This Still a Problem? 🚫

1. Adding New Payment Methods:

Every time a new payment method is added, you need to go into the PaymentProcessor class and add a new else if block. This results in code duplication and poor maintainability.

2. Scalability Issues:

As the number of payment methods increases (imagine 20+ methods), the PaymentProcessor class will become massive, making it hard to read and hard to modify.

Step 3: The Strategy Pattern – The Right Way. 🧠 ✨

Now that we've seen the limitations of the traditional approach, let's apply the Strategy Design Pattern to solve the problem more elegantly.

In the Strategy Pattern, we create a family of algorithms (in this case, payment methods), and we allow the client (in this case, PaymentProcessor) to choose the appropriate algorithm at runtime. The key benefit is that we can easily add new payment methods without modifying the existing code.

Let's break it down step-by-step and walk through the complete code.

1: Define the Strategy Interface 🎯

The first step is to define a common interface that all the payment methods will follow. This interface will have a method called processPayment(), which each payment method class will implement.

Java

```
1 // PaymentStrategy interface (defines the common method for all payme
2 public interface PaymentStrategy {
3     void processPayment(); // Abstract method for processing payments
4 }
```

Here, we've created a `PaymentStrategy` interface with a single method `processPayment()`. Each payment method will implement this interface and provide its own implementation of the `processPayment()` method.

2: Implement Concrete Payment Strategies 🏠💻

Now, we create the concrete payment strategies. These are the actual implementations for each payment method like Credit Card, PayPal, Crypto, etc.

Java

```
1 // Concrete strategy for credit card payment
2 public class CreditCardPayment implements PaymentStrategy {
3     public void processPayment() {
4         System.out.println("Processing credit card payment...");
5     }
6 }
7
8 // Concrete strategy for PayPal payment
9 public class PayPalPayment implements PaymentStrategy {
10    public void processPayment() {
11        System.out.println("Processing PayPal payment...");
12    }
13 }
14
15 // Concrete strategy for crypto payment
16 public class CryptoPayment implements PaymentStrategy {
17    public void processPayment() {
18        System.out.println("Processing crypto payment...");
19    }
20 }
21
22 // Concrete strategy for Stripe payment
23 public class StripePayment implements PaymentStrategy {
24    public void processPayment() {
25        System.out.println("Processing Stripe payment...");
26    }
27 }
```

Each class (like `CreditCardPayment`, `PayPalPayment`, etc.) implements the `PaymentStrategy` interface. They each have their own version of `processPayment()` that contains the logic for processing that specific payment method.

3: Modify the PaymentProcessor Class to Use the Strategy 🛠️

The key idea in the Strategy Pattern is that we will delegate the payment processing to the appropriate strategy. So, we'll modify the PaymentProcessor class to hold a reference to a PaymentStrategy and delegate the call to processPayment().

Here's how we do that:

Java

```
1 public class PaymentProcessor {
2     private PaymentStrategy paymentStrategy; // Reference to a payment
3     // Constructor to set the payment strategy
4     public PaymentProcessor(PaymentStrategy paymentStrategy) {
5         this.paymentStrategy = paymentStrategy;
6     }
7
8     // Process payment using the current strategy
9     public void processPayment() {
10         paymentStrategy
11             .processPayment(); // Delegate the payment processing to the
12     }
13
14     // Dynamically change payment strategy at runtime
15     public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
16         this.paymentStrategy = paymentStrategy;
17     }
18 }
```

In this class:

1. The PaymentProcessor class has a reference to a PaymentStrategy (i.e., one of the payment methods).

2. It uses this reference to call the processPayment() method.

3. We can dynamically change the strategy at runtime using the setPaymentStrategy() method. This means that if the user wants to switch from CreditCard to PayPal, we can change the strategy without changing the rest of the code!

4: Use the Strategy Pattern in Action 🎉

Now, let's use our PaymentProcessor class with different payment strategies.

Here's how it works:

Java

```
1 public class Main {
2     public static void main(String[] args) {
3         // Create strategy instances for each payment type
4         PaymentStrategy creditCard = new CreditCardPayment();
5         PaymentStrategy payPal = new PayPalPayment();
6         PaymentStrategy crypto = new CryptoPayment();
7         PaymentStrategy stripe = new StripePayment();
8         // Use the Strategy Pattern to process payments
9         PaymentProcessor processor =
10             new PaymentProcessor(creditCard); // Initially using Credit
11         processor.processPayment(); // Processing credit card payment..
12         // Dynamically change the payment strategy to PayPal
13         processor.setPaymentStrategy(payPal);
14         processor.processPayment(); // Processing PayPal payment...
15         // Switch to Crypto
16         processor.setPaymentStrategy(crypto);
17         processor.processPayment(); // Processing crypto payment...
18         // Switch to Stripe
19         processor.setPaymentStrategy(stripe);
20         processor.processPayment(); // Processing Stripe payment...
21     }
22 }
```

Explanation of Code:

- Create strategy instances:

We create different strategy objects like CreditCardPayment, PayPalPayment, etc.

- PaymentProcessor:

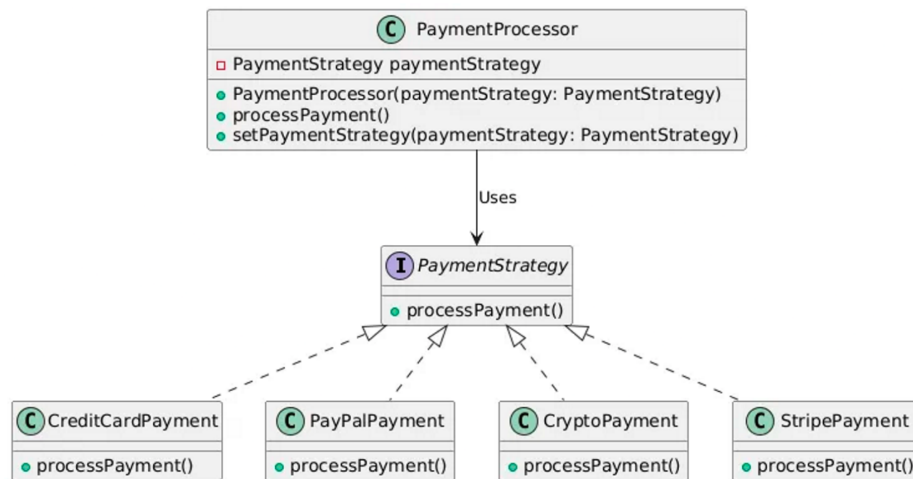
We instantiate PaymentProcessor and pass a specific payment strategy (e.g., CreditCardPayment) to it.

- Dynamically change strategies:

We can change the payment method dynamically using the setPaymentStrategy() method, and no modification to the PaymentProcessor class is required.

- Process payment:

We call processor.processPayment() to process the payment using the current strategy.



Advantages of the Strategy Pattern 🌟

1. Flexibility:

We can switch between different payment strategies at runtime without modifying the **PaymentProcessor** class. 🔄

2. Maintainability:

New payment methods can be added by simply creating new strategy classes. We don't need to touch the existing code. 🛠️

3. Separation of Concerns:

Each payment method has its own class, making the code easier to understand and maintain. 🖌️

4. Extensibility:

As new payment methods become available, we can simply add them by creating new strategy classes. 💡

Real-Life Use Cases for the Strategy Pattern 🌐

• Payment Methods 🏠 :

Process payments via different methods like Credit Card, PayPal, Crypto, etc.

• Sorting Algorithms 📊 :

Use different sorting strategies (e.g., quick sort, merge sort) depending on the situation.

• Shipping Costs 📦 :

Calculate shipping costs based on various factors such as location, delivery speed, and package size.

Conclusion 🎯

The Strategy Pattern is a powerful tool for making your code modular, flexible, and scalable. By encapsulating behaviors (like payment methods) into separate strategy classes, you can easily change or add new behaviors without modifying the existing code. This results in a cleaner, more maintainable codebase that can adapt to future requirements without significant changes. ✨🚀