Strategy Design Pattern Explained

Understanding the Strategy Design Pattern

In the Strategy Design Pattern, the goal is to encapsulate varying behavior (functionality) into separate classes so that they can be interchanged dynamically without modifying the main class.

The main class (Context) performs some operation but delegates the variable behavior to a separate interface (Strategy). Different implementations of this interface represent different strategies (behaviors).

The main class has a Strategy and delegates the variable behavior to it, allowing flexibility and adherence to the Open/Closed Principle.

Real-World Example: Navigation App

A navigation app like Google Maps provides directions for different modes of transport — car, bike, or walking. Each mode has its own route calculation logic, so each can be represented as a different strategy implementing a common IRouteStrategy interface.

The NavigationApp (Context) class uses a route strategy but does not care which one—it just calls the BuildRoute() method. This makes it easy to switch between strategies (car, bike, walk) dynamically without changing the core logic.

```
// Strategy Interface
public interface IRouteStrategy
{
void BuildRoute(string start, string end);
}

// Concrete Strategies
public class CarRouteStrategy : IRouteStrategy
{
public void BuildRoute(string start, string end)
{
Console.WriteLine($"Calculating fastest car route from {start} to {end} using highways.");
}
}

public class BikeRouteStrategy : IRouteStrategy
{
```

```
public void BuildRoute(string start, string end)
{
Console.WriteLine($"Calculating safe and scenic bike route from {start} to {end}.");
}
}

public class WalkRouteStrategy : IRouteStrategy
{
public void BuildRoute(string start, string end)
{
Console.WriteLine($"Calculating shortest walking path from {start} to {end} through sidewalks.");
}
}

// Context
public class NavigationApp
{
private IRouteStrategy _routeStrategy;

public void SetRouteStrategy(IRouteStrategy routeStrategy)
{
_routeStrategy = routeStrategy;
}

public void BuildRoute(string start, string end)
{
_routeStrategy.BuildRoute(start, end);
}
}

// Usage
var app = new NavigationApp();
app.SetRouteStrategy(new CarRouteStrategy());
app.BuildRoute("Home", "Office");

app.SetRouteStrategy(new BikeRouteStrategy());
app.BuildRoute("Home", "Park");

app.SetRouteStrategy(new WalkRouteStrategy());
app.BuildRoute("Home", "Store");
```

Real-World Example: Payment Gateway System

Scenario: An e-commerce platform supports multiple payment methods like Credit Card, PayPal, UPI, or Apple Pay. Each payment method has different logic for processing payments.

Instead of writing if-else statements inside a single class, each payment method can be extracted into its own class implementing a common interface (IPaymentStrategy).

This makes the code extensible and easier to maintain. When a new payment method (like Apple Pay) is introduced, you simply add a new strategy class without modifying existing code.

```
// Strategy Interface
public interface IPaymentStrategy
{
void Pay(decimal amount);
}

// Concrete Strategies
public class CreditCardPayment : IPaymentStrategy
{
public void Pay(decimal amount)
{
Console.WriteLine($"Paid {amount} using Credit Card.");
}
}

public class PayPalPayment : IPaymentStrategy
{
public void Pay(decimal amount)
{
Console.WriteLine($"Paid {amount} using PayPal.");
}
}

public class UpiPayment : IPaymentStrategy
{
public void Pay(decimal amount)
{
Console.WriteLine($"Paid {amount} using UPI.");
}
}

// Context
```

```
public class PaymentService
{
private IPaymentStrategy _paymentStrategy;

public void SetPaymentStrategy(IPaymentStrategy paymentStrategy)
{
_paymentStrategy = paymentStrategy;
}

public void ProcessPayment(decimal amount)
{
_paymentStrategy.Pay(amount);
}
}

// Usage
var paymentService = new PaymentService();
paymentService.SetPaymentStrategy(new UpiPayment());
paymentService.ProcessPayment(2000);

paymentService.SetPaymentStrategy(new PayPalPayment());
paymentService.ProcessPayment(3500);
```

How to Identify When to Use Strategy Pattern

Use the Strategy Pattern when:
- You have multiple ways to perform the same operation (e.g., Pay, Sort, Authenticate).
- You see many if-else or switch statements based on type or mode.
- You need to change behavior dynamically at runtime.
- You want to follow the Open/Closed Principle.

Why it helps:
The Strategy Pattern helps by reducing code duplication, improving readability, and making the system more extensible. It follows the Open/Closed Principle — open for extension but closed for modification. When you need a new behavior, you simply add a new strategy class without touching the existing logic.