Strategy Pattern .NET (C#)

Merwan Chinta · Follow
Published in CodeNx · 4 min read · Feb 3, 2024

The Strategy Pattern is a behavioral design pattern that enables an algorithm's behavior to be selected at runtime. The main idea is to define a family of algorithms, encapsulate each one, and make them interchangeable. The Strategy Pattern lets the algorithm vary independently from clients that use it.

Components

1. **Context**: A class that contains a reference to a Strategy instance. It delegates the execution of a task to the strategy's implementation without having to understand the details of how the strategy works.
2. **Strategy Interface**: This defines a common interface for all concrete strategies. It declares a method the context uses to execute a strategy.
3. **Concrete Strategies**: These are individual classes that implement the Strategy interface. Each represents a different algorithm or way of performing a task.

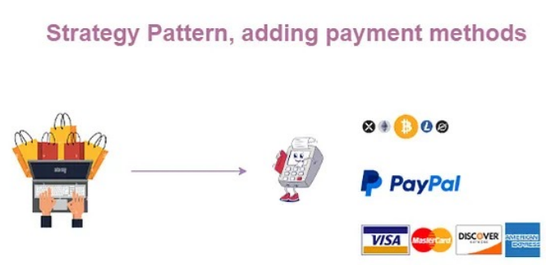
Principles and Policies

1. **Encapsulation of Variations**: The Strategy Pattern encapsulates the varying parts of the algorithm from the parts that stay the same. This means changes to the algorithm don't affect the client code.
2. **Program to an Interface, not an Implementation**: The client interacts with the strategy through an interface, not directly with a concrete implementation. This means the client code can work with any strategy that implements the interface.
3. **Favor Composition over Inheritance**: Instead of inheriting behavior, the Strategy Pattern uses composition to delegate responsibility to a strategy object. This provides more flexibility in choosing the appropriate behavior.
4. **Open/Closed Principle**: The system should be open for extension but closed for modification. New strategies can be added without changing the context or the way client code uses the system.
5. **Single Responsibility Principle**: Each strategy class has a single responsibility, representing a specific algorithm or behavior. This makes them easy to understand, implement, and test.

By following these principles, the Strategy Pattern provides a flexible structure for handling varying algorithms or behaviors, making it easier to manage, extend, and modify the functional parts of an application.

Payment Processing System

Let's consider a payment processing system where different payment strategies might be required, such as credit card payments, PayPal, or cryptocurrency. Payment strategies can often change or require additions, making it a suitable use case for the Strategy Pattern.



Strategy Pattern, adding new payment methods — Image source: Created by Author


```
// Usage
var paymentProcessor = new PaymentProcessor();
paymentProcessor.ProcessPayment(100.00m, "CreditCard");
```

Now, suppose you need to add PayPal as a payment method. You might modify the `PaymentProcessor` class and add another if-else condition.


```
public class PaymentProcessor
{
    public void ProcessPayment(decimal amount, string method)
    {
        if (method == "CreditCard")
        {
            // Logic to process credit card payment
            Console.WriteLine($"Processing {amount} via Credit Card");
        }
        else if (method == "PayPal")
        {
            // Logic to process PayPal payment
            Console.WriteLine($"Processing {amount} via PayPal");
        }
        // As new payment methods are added, more if-else statements are added h
```

```
    }
}

// Usage
var paymentProcessor = new PaymentProcessor();
paymentProcessor.ProcessPayment(100.00m, "CreditCard");
paymentProcessor.ProcessPayment(75.50m, "PayPal");
```

Problems with this Approach

1. Scalability: Each new payment method requires adding more if-else conditions to the `ProcessPayment` method, making it grow indefinitely. Merwan Chinta
2. Maintainability: Over time, the `ProcessPayment` method becomes increasingly complex and harder to manage.
3. Open/Closed Principle Violation: The class is not closed for modification. Every time a new payment type is added, you have to modify this class. Merwan Chinta

With Strategy Pattern

Now, let's refactor the code to use the Strategy Pattern, making it more flexible and maintainable. Merwan Chinta

Strategy Interface

```
public interface IPaymentStrategy
{
    void ProcessPayment(decimal amount);
}
```

Concrete Strategies Merwan Chinta

```
public class CreditCardPaymentStrategy : IPaymentStrategy
{
    public void ProcessPayment(decimal amount)
    {
        // Logic to process Credit Card payment
        Console.WriteLine($"Processing {amount} via Credit Card");
    }
}

public class PayPalPaymentStrategy : IPaymentStrategy
{
    public void ProcessPayment(decimal amount)
    {
        // Logic to process PayPal payment
        Console.WriteLine($"Processing {amount} via PayPal");
    }
}
```

Context Class

```
public class PaymentProcessor
{
    private IPaymentStrategy _paymentStrategy;

    public PaymentProcessor(IPaymentStrategy paymentStrategy)
    {
        _paymentStrategy = paymentStrategy;
    }

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

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

Usage Merwan Chinta

```
var creditCardPayment = new PaymentProcessor(new CreditCardPaymentStrategy());
creditCardPayment.ProcessPayment(100.00m);
// Output: Processing 100.00 via Credit Card

var payPalPayment = new PaymentProcessor(new PayPalPaymentStrategy());
payPalPayment.ProcessPayment(75.50m);
// Output: Processing 75.50 via PayPal
```

Adding a new payment method in the context of the Strategy Pattern is a straightforward process and doesn't require changing existing code, which is one of the main advantages of this pattern. Merwan Chinta

Let's walk through adding a new Cryptocurrency payment strategy to the existing system and compare how this approach simplifies the extension of functionality.

Adding Cryptocurrency Payment Strategy Merwan Chinta

Define the new concrete strategy by creating a class that implements the `IPaymentStrategy` interface.

```
public class CryptoPaymentStrategy : IPaymentStrategy
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing {amount} via Cryptocurrency");
        // Actual cryptocurrency processing logic
    }
}
```

Using the new strategy is as simple as instantiating the `PaymentProcessor` with the new `CryptoPaymentStrategy`

```
var cryptoPayment = new PaymentProcessor(new CryptoPaymentStrategy());
cryptoPayment.ProcessPayment(50.00m);
// Output: Processing 50.00 via Cryptocurrency
```

Adding a new payment strategy in the context of the Strategy Pattern demonstrates the pattern's ability to handle changes and extensions efficiently. It allows new functionalities to be added seamlessly without impacting the existing codebase, significantly reducing the risk of

introducing bugs and making the system more maintainable and scalable.

Advantages using Strategy Pattern

Merwan Chinta

- Ease of Extension: You simply create a new class that implements the `IPaymentStrategy` interface. There's no need to modify any existing code.
- Adherence to Open/Closed Principle: The system is open for extension but closed for modification. You can add new payment strategies without altering existing classes.
- Simplicity and Maintainability: The `PaymentProcessor` class remains simple and does not grow in complexity as new payment methods are added. Each payment method is encapsulated in its own class, making the system easier to understand and maintain.

. . .

I trust this information has been valuable to you. 🌟 Wishing you an enjoyable and enriching learning journey!

📚 For more insights like these, feel free to follow 👉 **Merwan Chinta**

Technology | Programming | Software Engineering | Dotnet Core | JavaScript

332 3

Published in CodeNx

308 Followers · Last published Oct 13, 2024

Dive into the software engineering realm. Discover cloud tech, .NET, Azure, Kubernetes, AWS, Machine Learning, Python, Generative AI, LLM's and more. Unravel the secrets of coding and innovation with Codex.

Follow

Written by Merwan Chinta

1.6K Followers · 4 Following

🚧 Roadblock Eliminator & Learning Advocate 🖥️ Software Architect 🚀 Efficiency & Performance Guide 🌐 Cloud Tech Specialist

Follow

Responses (3)

What are your thoughts? Respond

Erwin Dupuis
Feb 28, 2024 (edited)

Hi thanks for your article, as always it's a pleasure to read you !

I'm just wondering : Even with this approach, at some point we'll still need an if/else condition to know which payment method has been selected right ?

10 1 reply Reply

Zamkinos
Jun 10, 2024 (edited)

Thanks for your sharing. I have a little question: Why do you need "SetPaymentStrategy" method in "PaymentProcessor" class. Your constructor already does that. Or am I missing something here?

2 1 reply Reply

Kleinstadtcafé
Jun 13, 2024

If I read this right, dont you still have a huge if statement inside your payment method? You just added a class in between which does nothing but implement an interface. So my thought to this is you have your payment method that calls the... more

1 reply Reply

More from Merwan Chinta and CodeNx

In CodeNx by Merwan Chinta

Minimal APIs in .NET 8: A Simplified Approach to Build...

Introduction

Nov 21, 2023 375 4

In CodeNx by Chaitanya (Chey) Penmetsa

Implement Mediator Pattern with MediatR in C#

In this blog, we'll explore the implementation of the mediator pattern in C#. To grasp the...

Feb 4, 2024 39 1

In CodeNx by Merwan Chinta

Exception Handling in .NET Core Web API

Jan 14, 2024 326 2

In CodeNx by Merwan Chinta

OAuth 2.0

OAuth 2.0 is an authorization framework that enables applications to obtain limited acces...

Feb 13, 2024 259

See all from Merwan Chinta | See all from CodeNx

Recommended from Medium

In Level Up Coding by #hope

Dependency Injection in C#

Dependency Injection (DI) in C# is a design pattern and a technique used to achieve...

Jul 31, 2024 40

Yohan Malshika

Interface Segregation Principle in C#

Avoiding Code Bloat: Mastering ISP for Cleaner C# Interfaces

Dec 17, 2024 36 1

Lists

General Coding Knowledge
20 stories · 1886 saves

Stories to Help You Grow as a Software Developer
19 stories · 1572 saves

Coding & Development
11 stories · 983 saves

ChatGPT prompts
51 stories · 2508 saves

Is It Vritra - SDE I

Microservices in .NET Made Easy |Expert Strategies for Handling...

For .NET developers, Imagine you're building a big, complex software system. Now, think...

Sep 2, 2024 154

Vemuri Vamsi

Implementing the Bulkhead Pattern for Resilience in .NET wit...

In the previous article https://medium.com/@vemurivi/resilient-...

Oct 5, 2024 72

Ravi Patel

Understanding the Adapter Pattern: A Comprehensive Guide...

Design patterns are critical for solving common software design challenges and...

Sep 23, 2024

Stefan Đokić

How to create .NET Custom Guard Clause

Guard clauses in .NET are a programming practice used to improve code's readability...

Oct 5, 2024 5

See more recommendations