Autor: Philipp Khachik Version: 1.0 Datum: 2025-01-14

Adapter Pattern

Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to collaborate.

The Adapter pattern suggests that you place the interface conversion into a separate class called adapter, instead of trying to integrate it into existing classes.

Project Adapter Pattern

This project demonstrates the usage of the Adapter pattern to integrate different payment gateways (PayPal, Stripe) into a payment processing system.

Problem

Inkompatible Schnittstellen: Eine bestehende Zahlungs-App möchte verschiedene Zahlungsgateways (PayPal, Stripe) integrieren, aber die Schnittstellen passen nicht zusammen.

Lösung: Adapter Pattern

Adapter: Ein spezielles Objekt, das die Schnittstelle eines Objekts so umwandelt, dass ein anderes Objekt es verstehen kann. Wrapper: Der Adapter umhüllt ein Objekt, um die Komplexität der Umwandlung zu verbergen.

Code-Ausschnitte

```
    public interface IPaymentProcessor
    {
        Task<PaymentResult> ProcessPayment(PaymentDetails payment);
        Task<RefundResult> ProcessRefund(RefundDetails refund);
        Task<bool> ValidatePaymentDetails(PaymentDetails payment);
    }
```

```
    public class PayPalPaymentAdapter(PayPalGateway payPalGateway) : IPaymentProcessor
    {
        private readonly PayPalGateway _payPalGateway = payPalGateway;
```

```
public async Task<PaymentResult> ProcessPayment(PaymentDetails payment)
{
    var payPalRequest = new PayPalPaymentRequest
    {
        CreditCardNumber = payment.CardNumber,
```

```
        CardholderName = payment.CardHolderName,
        ExpirationDate = payment.ExpiryDate,
        SecurityCode = payment.Cvv,
        PaymentAmount = payment.Amount,
        Currency = payment.Currency
    };
```

```
try
{
    var payPalResult = await _payPalGateway.MakePayment(payPalRequest);
```

```
        return new PaymentResult
        {
            Success = payPalResult.PaymentStatus == "COMPLETED",
            TransactionId = payPalResult.PaymentId,
            Message = payPalResult.PaymentStatus == "COMPLETED" ? "Payment processed
successfully" : "Payment failed",
            ProcessedAt = payPalResult.ProcessedTime
        };
    }
    catch (Exception ex)
    {
        return new PaymentResult
        {
            Success = false,
            Message = ex.Message,
            ProcessedAt = DateTime.UtcNow
        };
    }
}
```

Key Points

IPaymentProcessor-Implementation: Die ursprünglichen Klassen (PayPalPaymentAdapter, StripePaymentAdapter) implementieren die IPaymentProcessor-Schnittstelle, um verschiedene Zahlungsanbieter einheitlich zu integrieren.

Adapter-Implementierung: Der `PayPalPaymentAdapter` und `StripePaymentAdapter` fungieren als Adapter-Klassen, die die Schnittstellen der jeweiligen Zahlungsanbieter in das Format der `IPaymentProcessor`-Schnittstelle übersetzen.

Methodenaufrufe: Der `PaymentService` verwendet die Adapter-Klassen, indem er deren `ProcessPayment`-Methode aufruft, was die Schnittstellenvereinheitlichung im Zahlungsprozess sicherstellt.

Beispiel-Code

```
    public class PaymentService
    {
        private readonly IPaymentProcessor _paymentProcessor;
```

```
public PaymentService(IPaymentProcessor paymentProcessor)
{
    _paymentProcessor = paymentProcessor;
}
```

```
public async Task<PaymentResult> ProcessOrderPayment(PaymentDetails paymentDetails)
{
    if (!await _paymentProcessor.ValidatePaymentDetails(paymentDetails))
    {
        return new PaymentResult
        {
            Success = false,
            Message = "Invalid payment details",
            ProcessedAt = DateTime.UtcNow
        };
    }
```

```
        return await _paymentProcessor.ProcessPayment(paymentDetails);
    }
}
```

```
            var paypalService = new PaymentService(
                new PayPalPaymentAdapter(new PayPalGateway())
            );
```

Vorteile und Nachteile

Vorteile

Einzelverantwortungsprinzip: Trennung von Schnittstellen-Umwandlung und Geschäftslogik. Offenes/Closed-Prinzip: Neue Adapter können ohne Änderung der Client-Code hinzugefügt werden.

Nachteile

Erhöhte Komplexität: Zusätzliche Schnittstellen und Klassen erforderlich.

Beziehungen zu anderen Mustern

Bridge: Ähnliche Struktur, aber mit unterschiedlichem Ziel. Decorator: Ähnliche Struktur, aber mit Fokus auf Erweiterung der Schnittstelle. Facade: Definiert eine neue Schnittstelle für bestehende Objekte, während der Adapter eine bestehende Schnittstelle umwandelt.