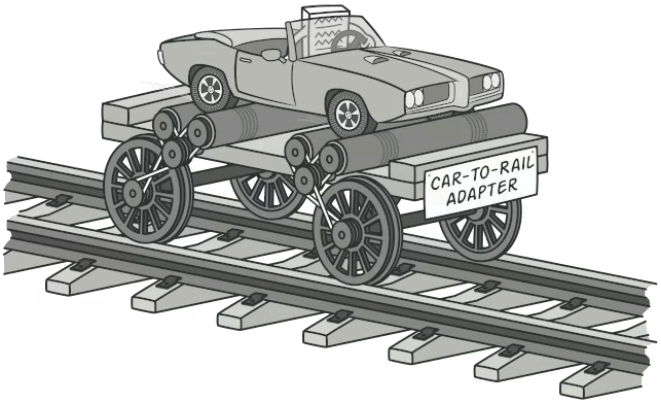
Adapter und Facade Pattern: Grundlagen, Aufbau und Praxisbeispiele

Dieses Dokument bietet eine strukturierte Einführung in zwei fundamentale strukturelle Entwurfsmuster: das Adapter- und das Facade-Muster. Wir beleuchten deren Funktion, ihren typischen Aufbau gemäß der Unified Modeling Language (UML) und demonstrieren ihre Anwendung anhand praktischer Beispiele, insbesondere mit Fokus auf die Implementierung in C#.

Ziel ist es, ein tiefes Verständnis für die Einsatzgebiete und den entscheidenden Unterschied zwischen diesen beiden Mustern zu vermitteln, um die Softwarearchitektur zu vereinfachen und die Kopplung zu reduzieren.

Vorstellung der Muster und ihre Funktion

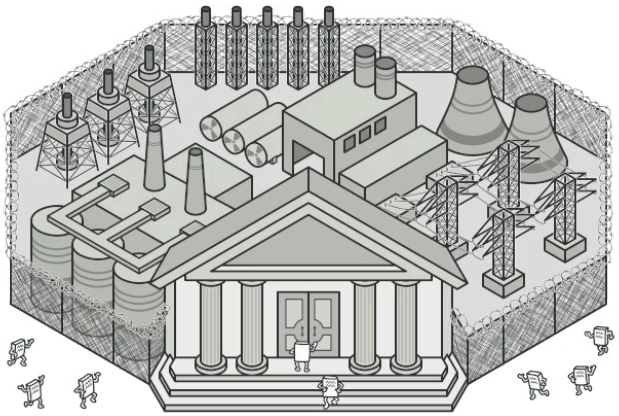
Adapter Pattern



Das Brückenglied für Inkompatibilität. Das Adapter Pattern ist ein strukturelles Entwurfsmuster, dessen Hauptzweck darin besteht, zwei ansonsten nicht kompatible Schnittstellen miteinander zu verbinden. Der Adapter fungiert als vermittelnde Schicht (Wrapper), die die Anfrage des Clients so umwandelt, dass das dahinterliegende Objekt (Adaptee) sie verarbeiten kann.

- Verbindet inkompatible Schnittstellen.
- Ermöglicht die Wiederverwendung bestehender Klassen.
- Arbeitet nach dem Prinzip "translate and forward".

Facade Pattern



Die einfache Vorderseite für komplexe Systeme. Das Facade Pattern (Facade) bietet eine vereinfachte und einheitliche Schnittstelle zu einer komplexen Gruppe von Klassen in einem Subsystem. Es verbirgt die Komplexität und die Details der Implementierung und entkoppelt den Client vom Subsystem.

- Vereinfacht den Zugriff auf ein komplexes Subsystem.
- Reduziert die Abhängigkeiten des Clients.
- Bietet einen einzigen Einstiegspunkt.

Adapter: Schnittstellen-Anpassung

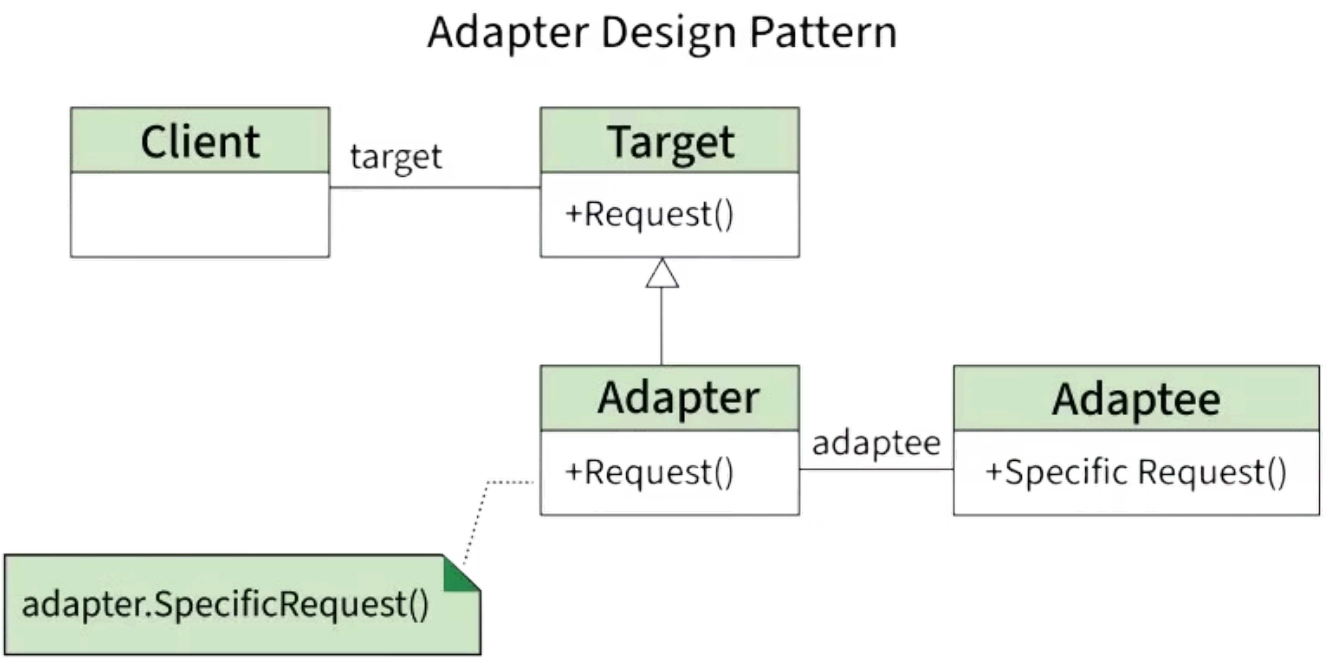
Fokus auf Kompatibilität zwischen **zwei** Elementen.

Facade: Subsystem-Vereinfachung

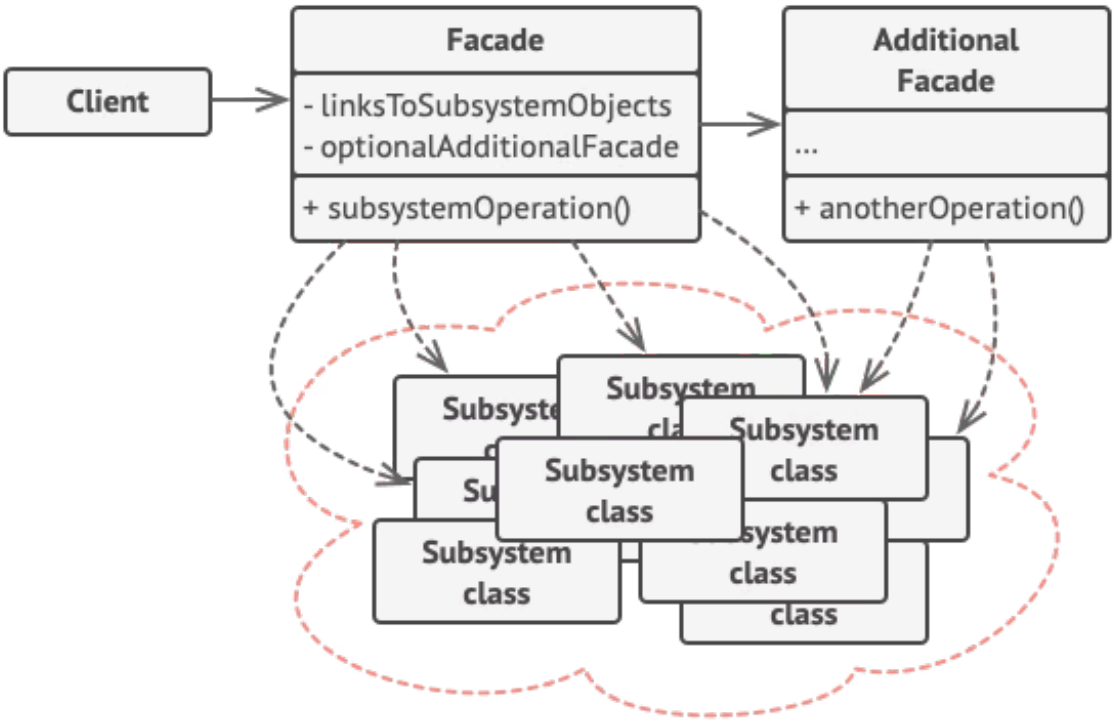
Fokus auf Vereinfachung einer **Gruppe** von Klassen.

Aufbau der Muster (UML)

Adapter UML-Struktur



Facade UML-Struktur



Während der Adapter eine transformationale Rolle spielt (Übersetzung von Interfaces), spielt die Facade eine aggregierende Rolle (Bündelung von Funktionalität).

Beispiele aus dem Internet

Wir demonstrieren die Muster an zwei Systemen in C#.

1. Adapter Pattern: Payment Service

```
// Target Interface - Einheitliche Schnittstelle
public interface IPaymentProcessor {
    bool ProcessPayment(decimal amount, string currency);
    string GetTransactionId();
}

// Adaptee 1 - Existierende PayPal API
public class PayPalAPI {
    public string MakePayment(double euros) {
        Console.WriteLine($"PayPal: Zahlung von {euros} EUR");
        return "PAYPAL-" + Guid.NewGuid().ToString().Substring(0, 8);
    }

// Adapter für PayPal
public class PayPalAdapter : IPaymentProcessor {
    private PayPalAPI _payPalAPI;
    private string _transactionId;

    public PayPalAdapter() {
        _payPalAPI = new PayPalAPI();
    }

    public bool ProcessPayment(decimal amount, string currency) {
        // Konvertierung von decimal zu double
        double euroAmount = (double)amount;
        // Aufruf der PayPal-spezifischen Methode
        _transactionId = _payPalAPI.MakePayment(euroAmount);
        return !string.IsNullOrEmpty(_transactionId);
    }

    public string GetTransactionId() {
        return _transactionId;
    }
}
```

```
// Adaptee 2 - Existierende Stripe API
public class StripeService {
    public StripeResponse ChargeCard(int cents, string curr) {
        Console.WriteLine($"Stripe: Charging {cents} cents in {curr}");
        return new StripeResponse {
            Success = true,
            Id = "stripe_" + DateTime.Now.Ticks
        };
    }
}

public class StripeResponse {
    public bool Success { get; set; }
    public string Id { get; set; }
}

// Adapter für Stripe
public class StripeAdapter : IPaymentProcessor {
    private StripeService _stripeService;
    private string _transactionId;

    public StripeAdapter() {
        _stripeService = new StripeService();
    }

    public bool ProcessPayment(decimal amount, string currency) {
        // Konvertierung von EUR zu Cents
        int cents = (int)(amount * 100);
        // Aufruf der Stripe-spezifischen Methode
        var response = _stripeService.ChargeCard(cents, currency);
        _transactionId = response.Id;
        return response.Success;
    }

    public string GetTransactionId() {
        return _transactionId;
    }
}
```

```
// Client Code
public class PaymentService {
    public void ExecutePayment(IPaymentProcessor processor, decimal amount) {
        Console.WriteLine("\nStarte Zahlungsvorgang...");
        if (processor.ProcessPayment(amount, "EUR")) {
            Console.WriteLine($"Erfolg! Transaction ID: " +
                              processor.GetTransactionId());
        }
        else {
            Console.WriteLine("Zahlung fehlgeschlagen!");
        }
    }
}

// Verwendung
class Program {
    static void Main() {
        var paymentService = new PaymentService();

        // Verwendung mit PayPal
        IPaymentProcessor paypal = new PayPalAdapter();
        paymentService.ExecutePayment(paypal, 49.99m);

        // Verwendung mit Stripe (gleiche Schnittstelle!)
        IPaymentProcessor stripe = new StripeAdapter();
        paymentService.ExecutePayment(stripe, 99.99m);
    }
}
```

2. Facade Pattern: Vereinfachte Steuerung

Die `HomeFacade` bietet dem Benutzer eine einzige Methode zur Steuerung mehrerer Geräte, ohne dass er die einzelnen Interfaces kennen muss.

```
// Subsystem-Klassen (z.B. ITemperatureControl ist hier integriert)
public class LightSystem { public void SetBrightness(int level) { /* ... */ } }
public class MusicSystem { public void PlayAmbiance() { /* ... */ } }

// Die Facade
public class HomeFacade
{
    private readonly ITemperatureControl _thermostat;
    private readonly LightSystem _lights = new LightSystem();
    private readonly MusicSystem _music = new MusicSystem();

    public HomeFacade(ITemperatureControl thermostatAdapter)
    {
        _thermostat = thermostatAdapter;
    }

    // Vereinfachte Facadenmethode
    public void StartRelaxationMode()
    {
        Console.WriteLine("Aktivierung des Entspannungsmodus...");
        _thermostat.SetTargetTemperature(21); // Adapter wird genutzt
        _lights.SetBrightness(30);
        _music.PlayAmbiance();
    }
}
```