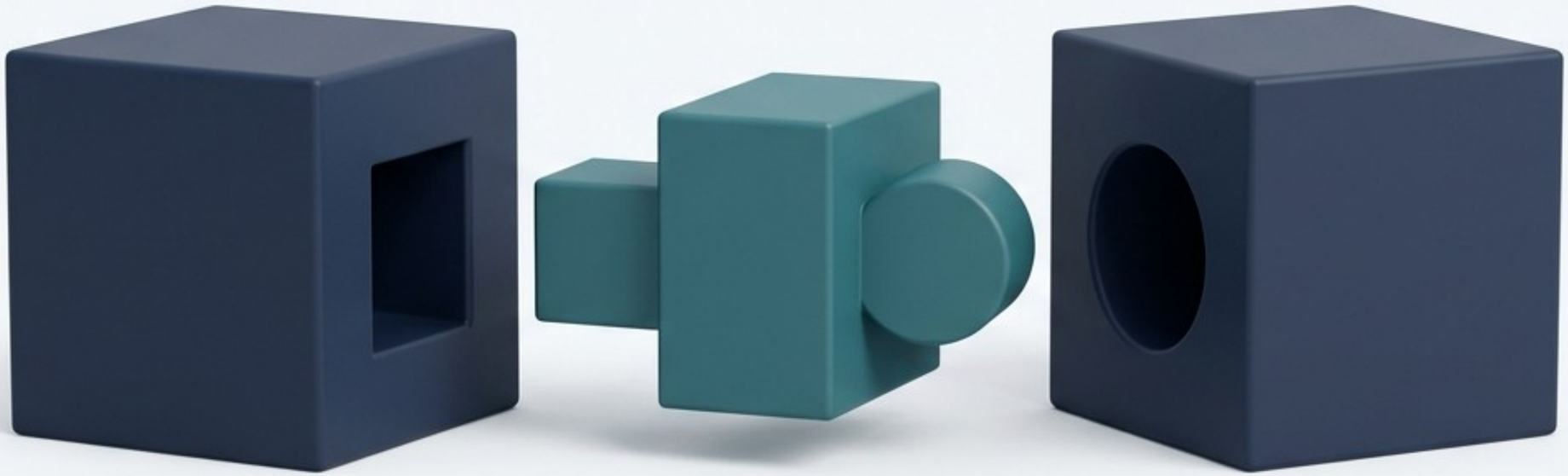
The Adapter Pattern

A practical guide to bridging incompatible systems without modifying source code.



The Integration Dilemma: When Systems Speak Different Languages

You have an existing system—legacy code, a library, or an API. You need to integrate a new component, but their interfaces do not match.

****Example**:**
Your application expects:
`IPayment.Process(amount)`

A new third-party SDK exposes:
`ExecutePayment(value)`

The fundamental challenge is that you cannot change the existing code:
❌ Legacy System
❌ Third-Party Library
❌ Vendor SDK

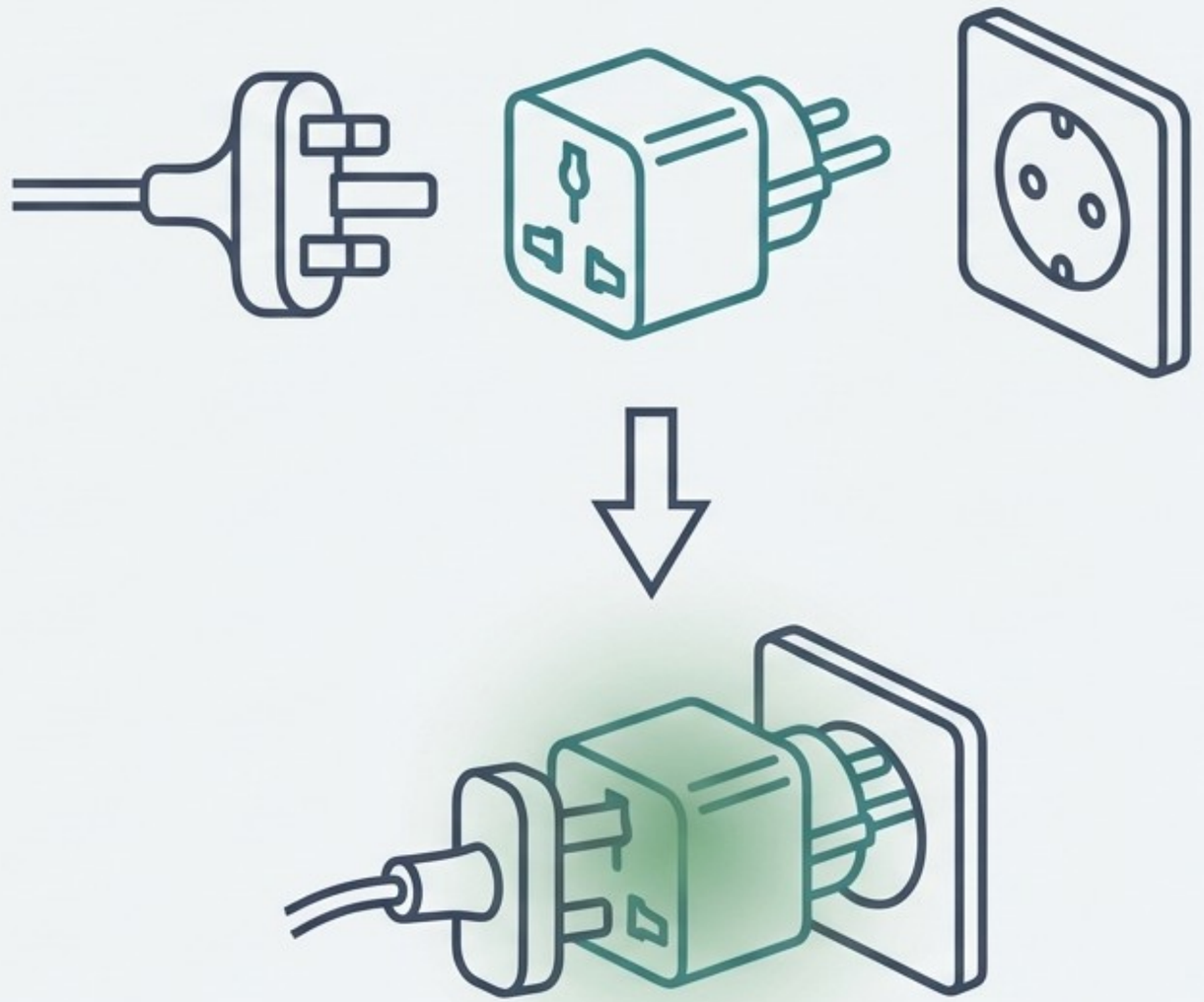
The Solution is as Simple as a Travel Plug Adapter

Analogy Setup
Your laptop has a UK plug.
The wall socket is European.
They don't fit.

The Action
You don't break the wall or replace the laptop. You use a travel adapter that converts the plug shape.

The Outcome
The laptop remains the same.
The wall remains the same.
The adapter makes them compatible.

This is **exactly how the Adapter Pattern works in software.**



The Common Mistake: Modifying Vendor Code Directly

A frequent temptation is to change the third-party classes to fit your system. For example, a developer might say: “Let me just rename or add methods to the vendor’s code…”

```
// Inside the third-party library file
public class ThirdPartyGateway
{
    // Tempting to change this to Process(double amount)
    public void ExecutePayment(double amount) { ... }
}
```

Cannot modify vendor library

It’s often compiled or not permissible.

Breaks future updates

Your changes will be overwritten.

Violates Open/Closed Principle

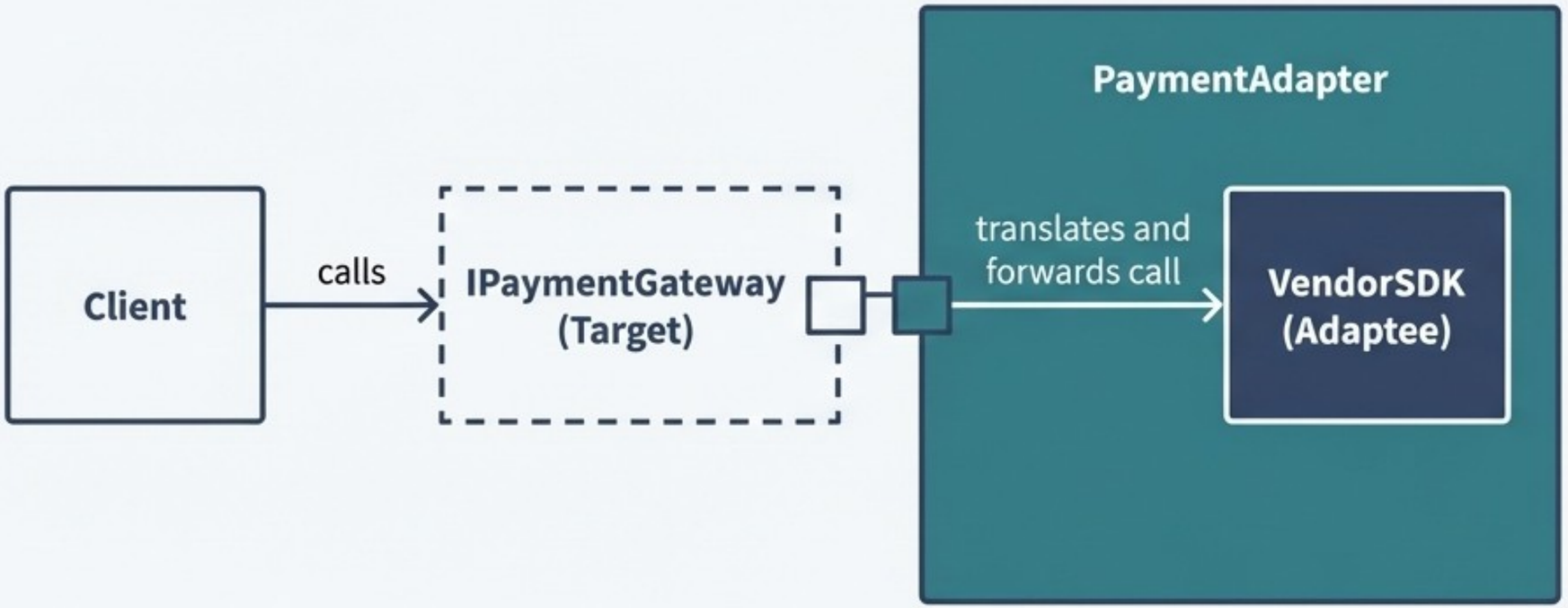
Good systems are open for extension, closed for modification.

High risk

This is unstable and a fundamentally bad practice.

A Better Way: Wrap, Don't Modify

Instead of changing code, we **wrap** the incompatible object inside a new class that exposes the interface our application expects. This new class is the **Adapter**.



Target Interface

The interface your application expects to work with.

Adapter

The new class that implements the Target Interface and translates calls to the incompatible object.

Adaptee

The existing, incompatible class or system (e.g., the vendor SDK).

Putting It into Practice: A Payment Gateway Scenario

The Goal: Integrate a new payment provider, RazorPay, into our application.

Our Application Expects (The Target)

```
public interface IPaymentGateway
{
    void Pay(double amount);
}
```

The Vendor Provides (The Adaptee)

```
public class RazorPaySdk
{
    public void MakePayment(double value) { … }
}
```

The method names and signatures do not match. Direct integration is impossible without a bridge.

Creating the Adapter: The Bridge Between Worlds

The RazorPayAdapter class implements our application's IPaymentGateway interface and internally uses the RazorPaySdk.

```
public class RazorPayAdapter : IPaymentGateway
{
    private readonly RazorPaySdk _sdk;

    public RazorPayAdapter(RazorPaySdk sdk)
    {
        _sdk = sdk;
    }

    public void Pay(double amount)
    {
        _sdk.MakePayment(amount);
    }
}
```

The Adapter implements the interface our app uses.

It holds a private instance of the incompatible SDK.

This is the crucial translation step.

Our app calls Pay(), the adapter calls MakePayment().

The Result: Clean, Decoupled Client Code

The client code remains unchanged. It continues to work with the IPaymentGateway interface, completely decoupled from the concrete implementation of RazorPay.

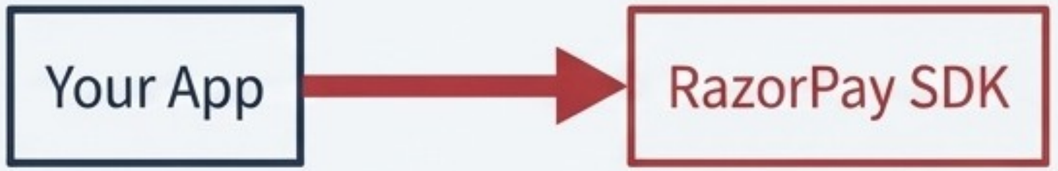
```
// Client creates the Adapter and uses it via
the interface
IPaymentGateway gateway = new
RazorPayAdapter(new RazorPaySdk());

// The client calls its familiar method. No
vendor-specific code here.
gateway.Pay(500);
```

- Client code stays clean and readable.
- The payment gateway is easily replaceable later.
- No hard-coded dependency on a specific vendor.

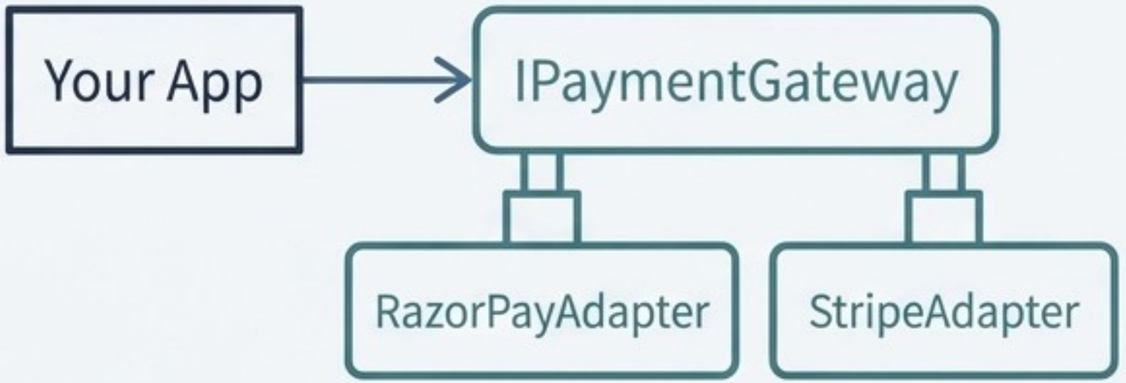
From Tight Coupling to True Flexibility

Without Adapter



Tightly coupled. Difficult to change.

With Adapter



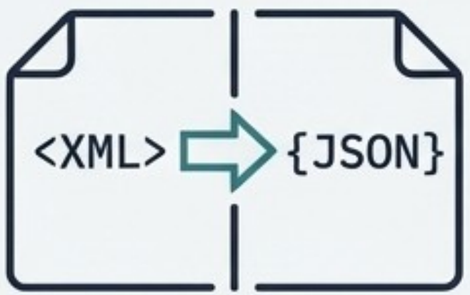
Decoupled. Plug & Play.

Tomorrow, the business decides: “Use Stripe now.”

The solution: Simply create a new `StripeAdapter`.
No core application code changes are needed.

Where You'll Find Adapters in the Wild

The Adapter is not just for payments. It is a fundamental pattern for integration across many domains.



Legacy → New System Integration

An adapter converts data formats, such as old system XML to a new system's JSON.

Third-Party API Wrappers

Companies rarely call SDKs (Stripe, AWS, Azure) directly. They are almost always wrapped in an adapter to isolate the system from vendor specifics.

Database Migrations

An adapter can bridge the gap between an old database repository and a new ORM.

Microservice Communication

Adapters are used between services to translate different Data Transfer Object (DTO) formats.

The Adapter Decision Framework

Use Adapter When...

- Two or more systems must work together.
- You cannot change the existing source code.
- An interface from a new module doesn't match what your system expects.
- You are migrating from a legacy system.
- You need to integrate vendor APIs safely.

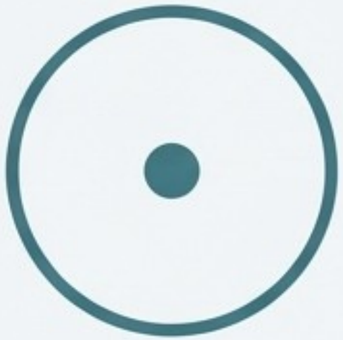
Avoid Adapter When...

- You control both systems and can refactor them to match.
- The mismatch is very simple and a small inline change is clearer.
- Adding an adapter would significantly overcomplicate the logic for a minor gain.

Upholding Core Design Principles

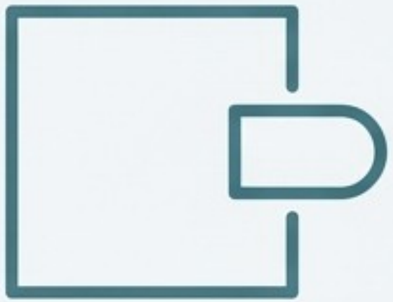
The Adapter Pattern is a practical application of several SOLID principles, leading to more maintainable and robust systems.

SRP (Single Responsibility Principle)



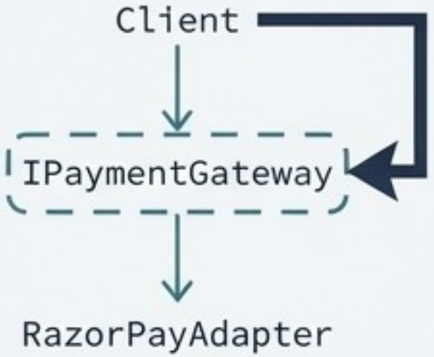
The Adapter has only one job: to convert requests between two interfaces. It doesn't contain business logic.

OCP (Open/Closed Principle)



We extend the system's behaviour with new adapters (e.g., `StripeAdapter`) *without* modifying existing client or interface code.

DIP (Dependency Inversion Principle)



The client depends on an abstraction (`IPaymentGateway`), not on a concrete implementation (`RazorPayAdapter`).

The Adapter Pattern at a Glance

Item	Meaning
Problem	Two essential interfaces do not match.
Solution	Create a new adapter class to bridge the gap.
Goal	Achieve compatibility without modification of existing code.
Benefits	Reusable, flexible, and safe integration of components.
Best For	Legacy systems, third-party APIs, SDKs, and migrations.
Principles	Single Responsibility, Open/Closed, Dependency Inversion.