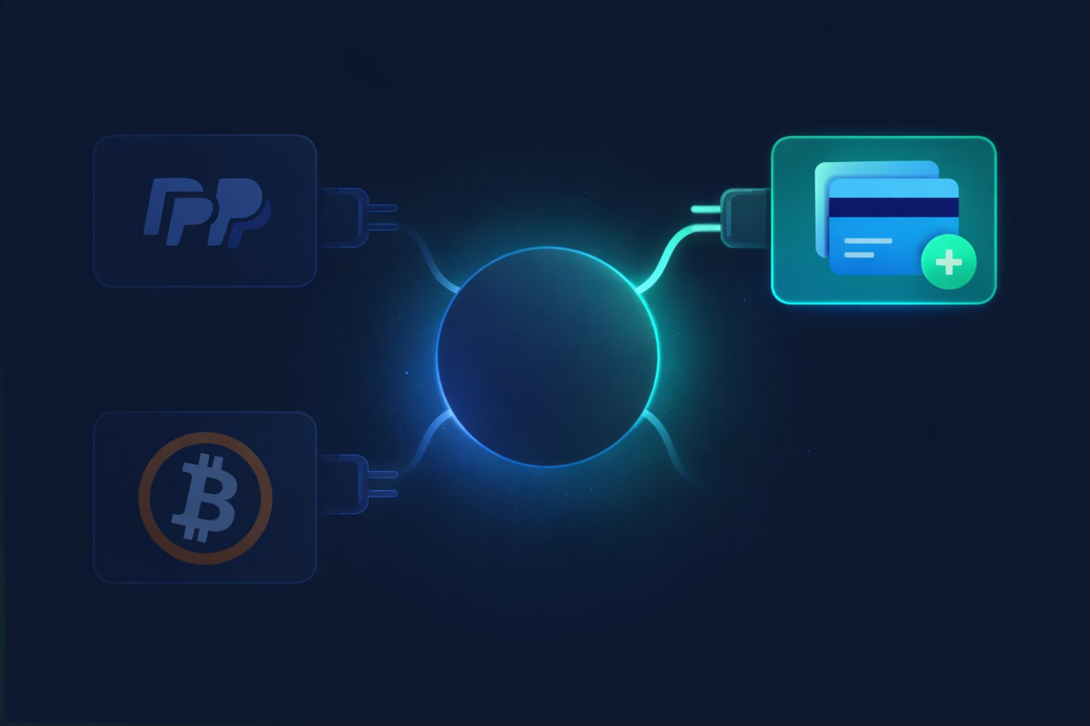
The Strategy Pattern & SRP

Decoupling Algorithms for Maintainable, Extensible Architectures

Defining the Strategy Pattern



Encapsulated Behaviors

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable at runtime.

- Composition over Inheritance.
- Dynamic behavior switching.
- Strong Isolation of Logic.

PART I

The "God Class" Monolith

A Single Responsibility Principle (SRP) Violation

The Single-Method Monolith

C# Code Implementation

```
public class PaymentProcessor {
  public void ProcessPayment( string type,
decimal amt) {
    if (type == "CreditCard") {
      /* 50 lines of Visa/Master logic */
    }
    else if (type == "PayPal") {
      /* API Handshake logic */
    }
    else if (type == "Crypto") {
      /* Blockchain validation */
    }
  }
}
```

SRP Violations

This class has **many reasons to change**, failing the fundamental rule of SRP:

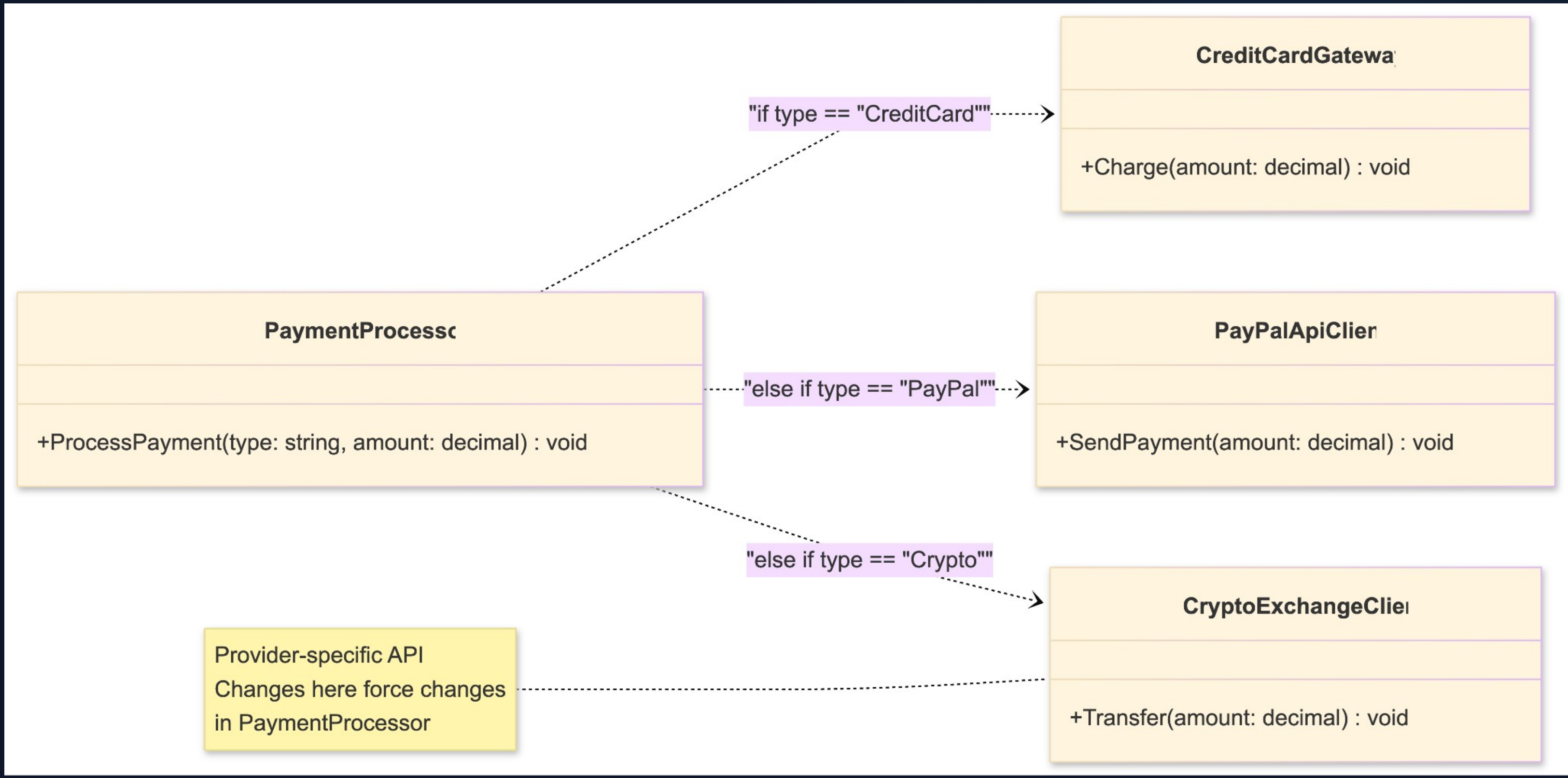
- ✗ Change to PayPal API breaks the file.
- ✗ Adding "Apple Pay" requires editing core logic.
- ✗ Testing Crypto requires mocking PayPal.

Fragile Complexity

The Cost of High Coupling

In a monolithic method, logic for unrelated features is tangled together. This "spaghetti code" creates a fragile system where a change in one area causes unexpected regressions in another.

The more payment methods you add, the more difficult the class becomes to read, maintain, and unit test effectively.


CreditCardGatewa
+Charge(amount: decimal) : void
"if type == "CreditCard""
PaymentProcessc
+ProcessPayment(type: string, amount: decimal) : void
"else if type == "PayPal""
PayPalApiClier
+SendPayment(amount: decimal) : void
"else if type == "Crypto""
CryptoExchangeClie
+Transfer(amount: decimal) : void
Provider-specific API
Changes here force changes
in PaymentProcessor

PART II

Refactoring to Strategy

Enforcing Single Responsibility through Interfaces

The Strategy UML Pattern

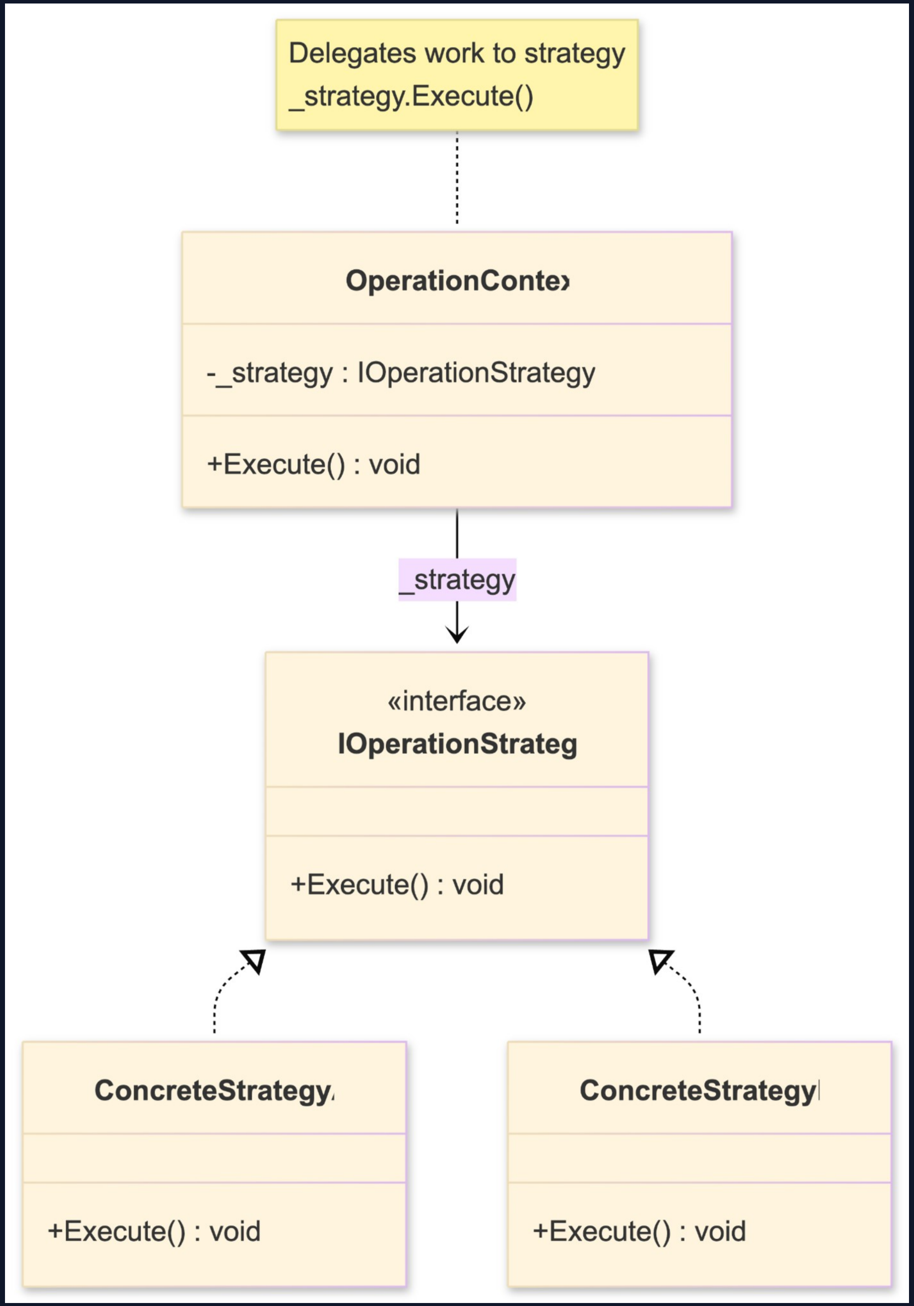
Anatomy of the Pattern

Context: Maintains a reference to the Strategy.

Strategy: The common interface (IPayment).

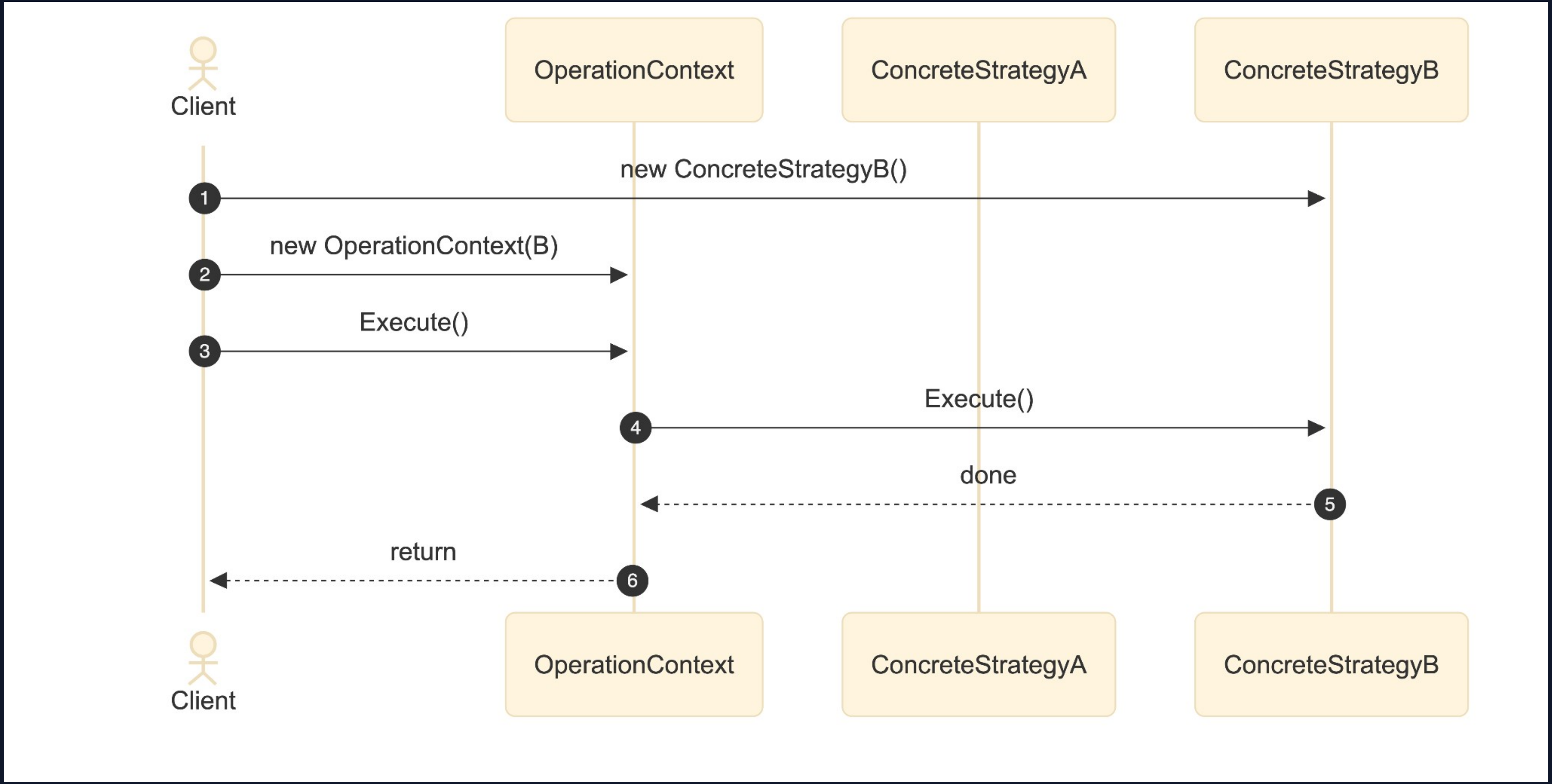
ConcreteStrategy: Individual classes implementing unique logic (e.g., PayPalStrategy).

The client interacts with the Context, oblivious to which ConcreteStrategy is actually executing.



The Strategy UML Pattern

Sequence Diagram



Defining the Contract

1. The Strategy Interface

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

A clean, single-method contract that all algorithms must follow.

2. Concrete Strategies

```
public class PayPalStrategy :
IPaymentStrategy {
  public void ProcessPayment(decimal amt)
{
    // PayPal API Handshake ONLY
  }
}
```

Now, each class has **exactly one reason to change**.

Context: The Runtime Switch

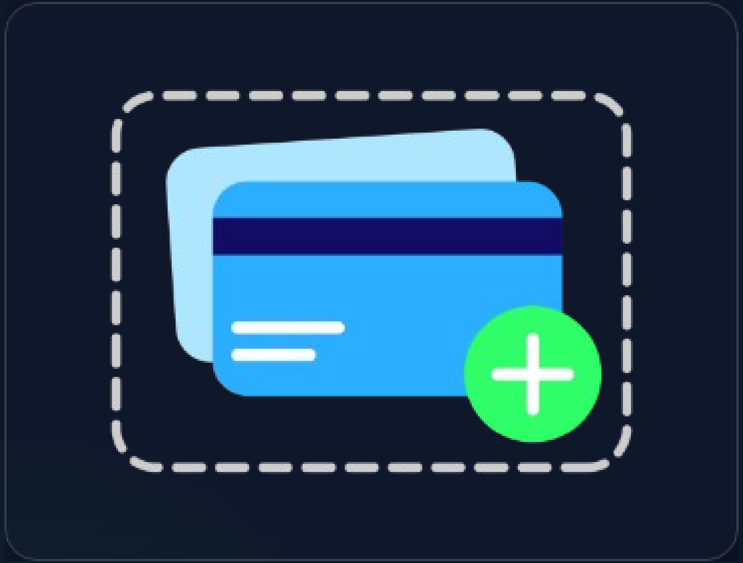
```
public class CheckoutContext {
  private IPaymentStrategy _strategy;

  public void SetStrategy(IPaymentStrategy strategy) {
    _strategy = strategy;
  }

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

- Decouples the Checkout flow from payment logic.
- Algorithms are "plugged in" dynamically.
- Easy to swap strategies mid-session.

Modular Payment Models



Card Strategy

Validation, encryption, and banking gateway integration.

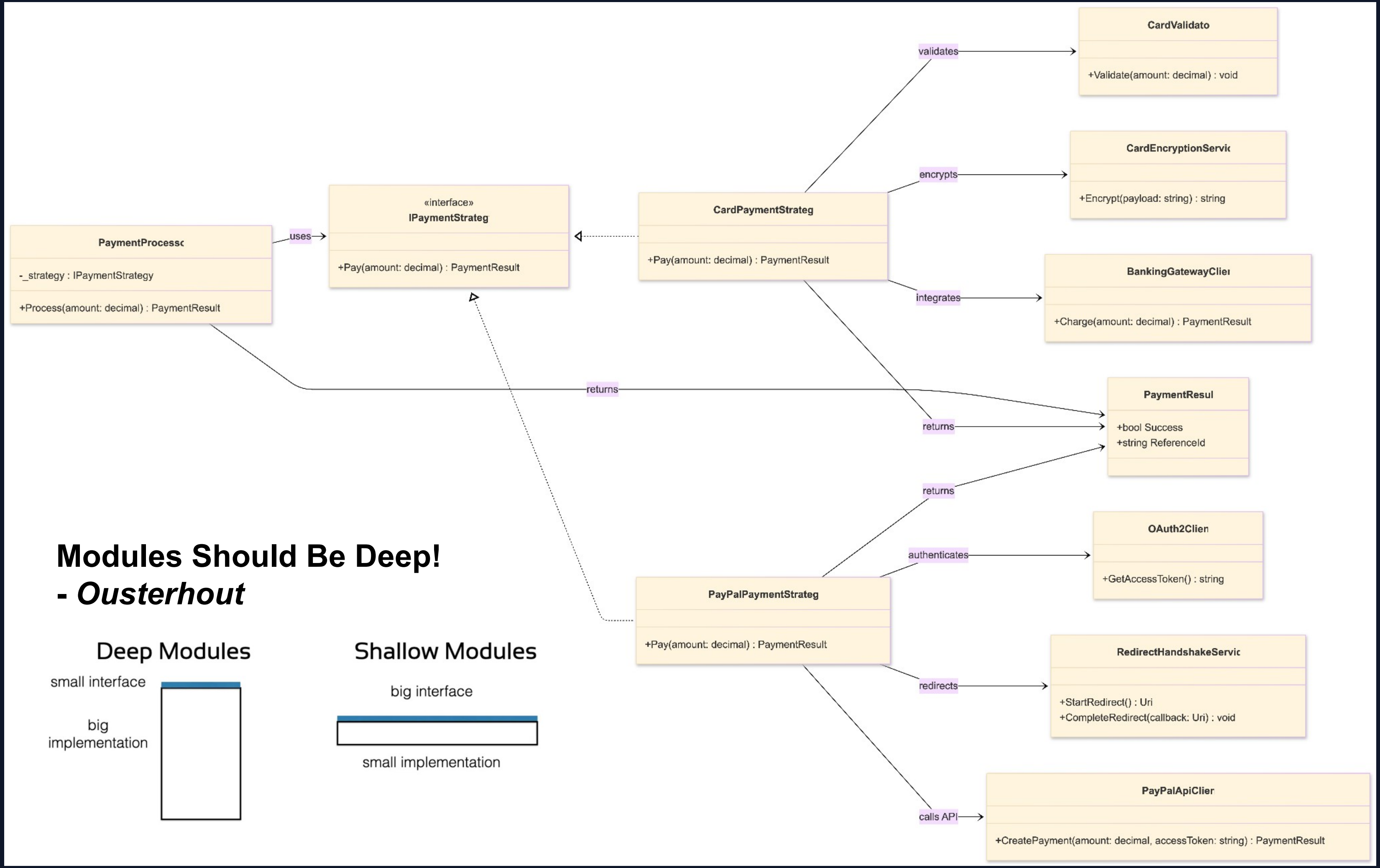
PayPal Strategy

OAuth2 authentication and external redirect handshakes.

Crypto Strategy

Blockchain verification and wallet address confirmation.

With Strategy



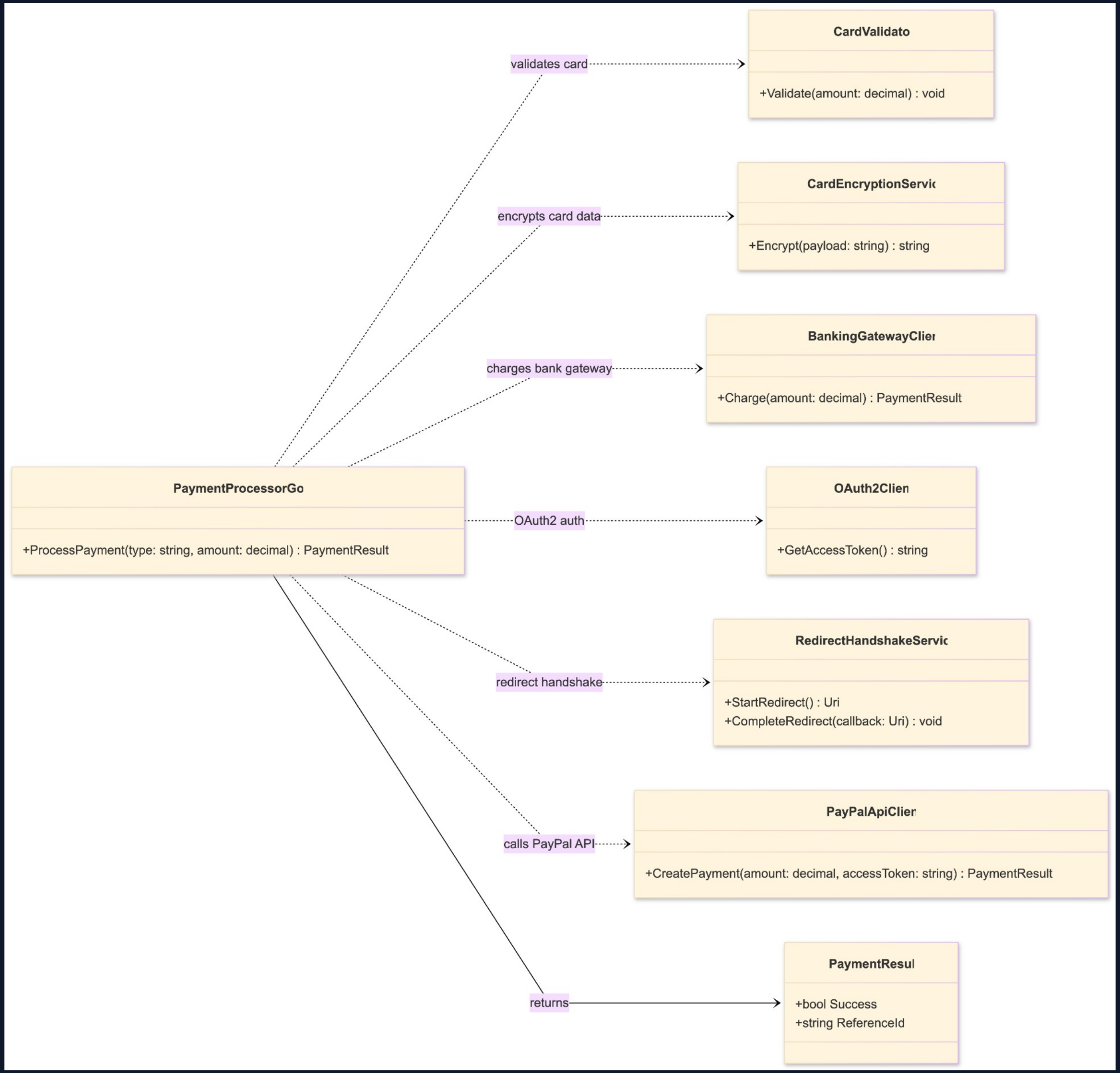
PaymentProcessor receives the payment strategy.

Coupling is limited to the dependencies of each payment strategy.

Each class has a single reason to change.

New processors can be added without changes to existing processors.

Without Strategy



CardValidato
+Validate(amount: decimal) : void
validates card
CardEncryptionServic
+Encrypt(payload: string) : string
encrypts card data
BankingGatewayClier
+Charge(amount: decimal) : PaymentResult
charges bank gateway
PaymentProcessorGo
+ProcessPayment(type: string, amount: decimal) : PaymentResult
OAuth2 auth
OAuth2Clien
+GetAccessToken() : string
redirect handshake
RedirectHandshakeServic
+StartRedirect() : Uri
+CompleteRedirect(callback: Uri) : void
calls PayPal API
PayPalApiClier
+CreatePayment(amount: decimal, accessToken: string) : PaymentResult
returns
PaymentResul
+bool Success
+string ReferenceId

All payment models live in PaymentProcessor

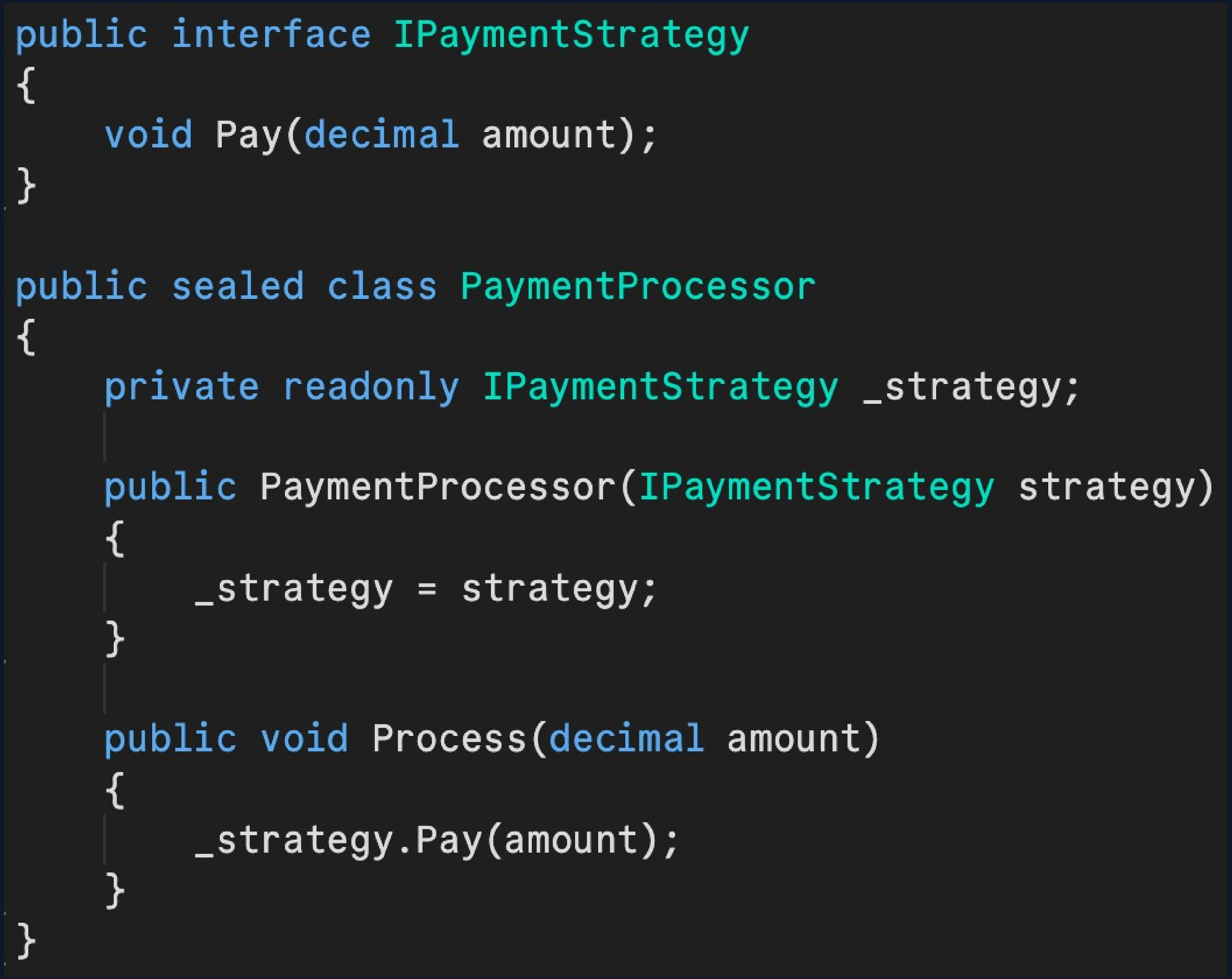
High coupling: dependencies created on every provider + helper (today and all future)

Many reasons to change:

- new payment type
- API change (PayPal/gateway/crypto)
- validation/encryption rules

Hard to test: must mock unrelated systems

C# Demo



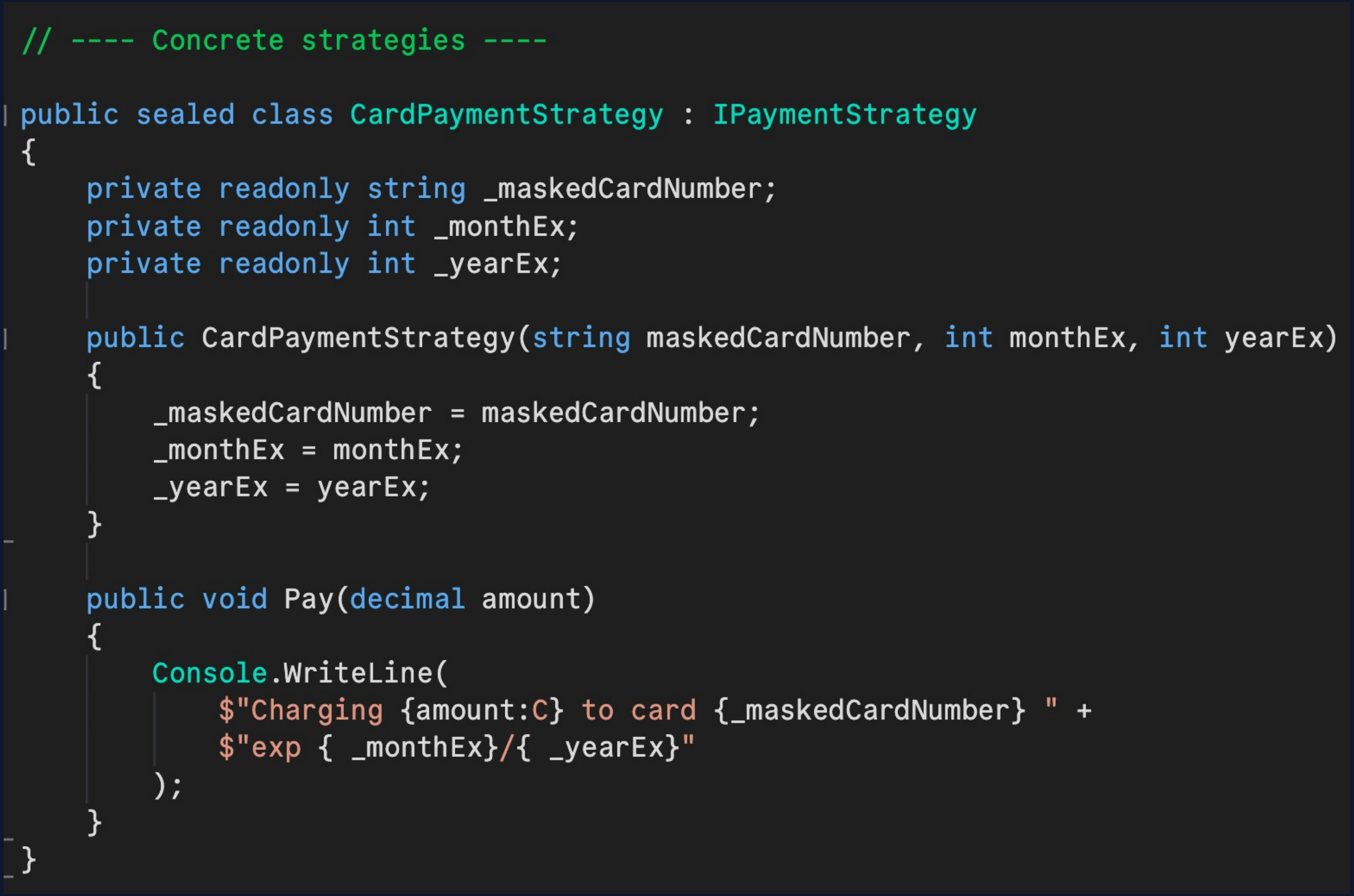
```csharp
public interface IPaymentStrategy
{
    void Pay(decimal amount);
}

public sealed class PaymentProcessor
{
    private readonly IPaymentStrategy _strategy;

    public PaymentProcessor(IPaymentStrategy strategy)
    {
        _strategy = strategy;
    }

    public void Process(decimal amount)
    {
        _strategy.Pay(amount);
    }
}
```

C# Demo



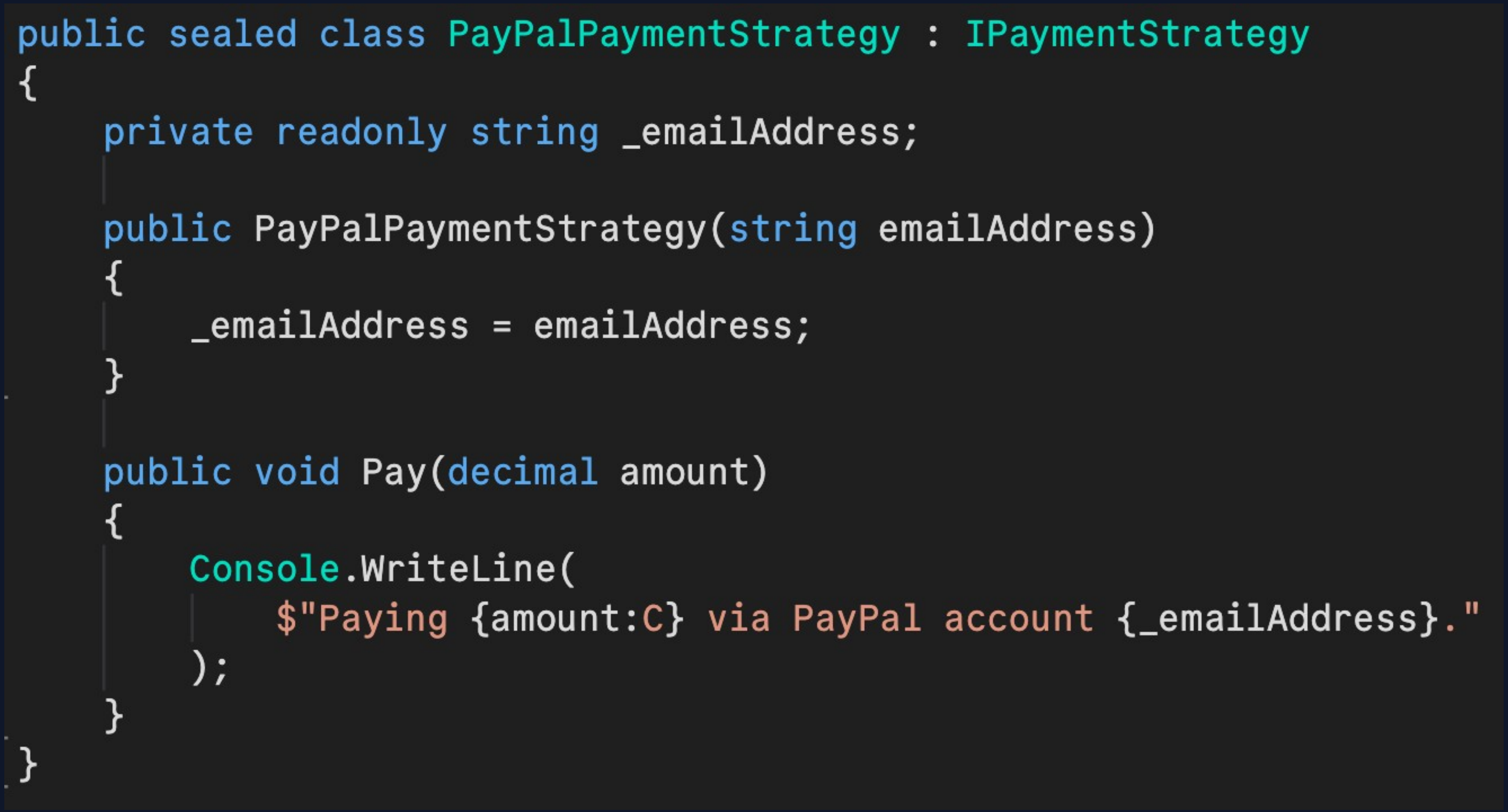
```
// ---- Concrete strategies ----

public sealed class CardPaymentStrategy : IPaymentStrategy
{
    private readonly string _maskedCardNumber;
    private readonly int _monthEx;
    private readonly int _yearEx;

    public CardPaymentStrategy(string maskedCardNumber, int monthEx, int yearEx)
    {
        _maskedCardNumber = maskedCardNumber;
        _monthEx = monthEx;
        _yearEx = yearEx;
    }

    public void Pay(decimal amount)
    {
        Console.WriteLine(
            $"Charging {amount:C} to card {_maskedCardNumber} " +
            $"exp { _monthEx}/{ _yearEx}"
        );
    }
}
```

C# Demo

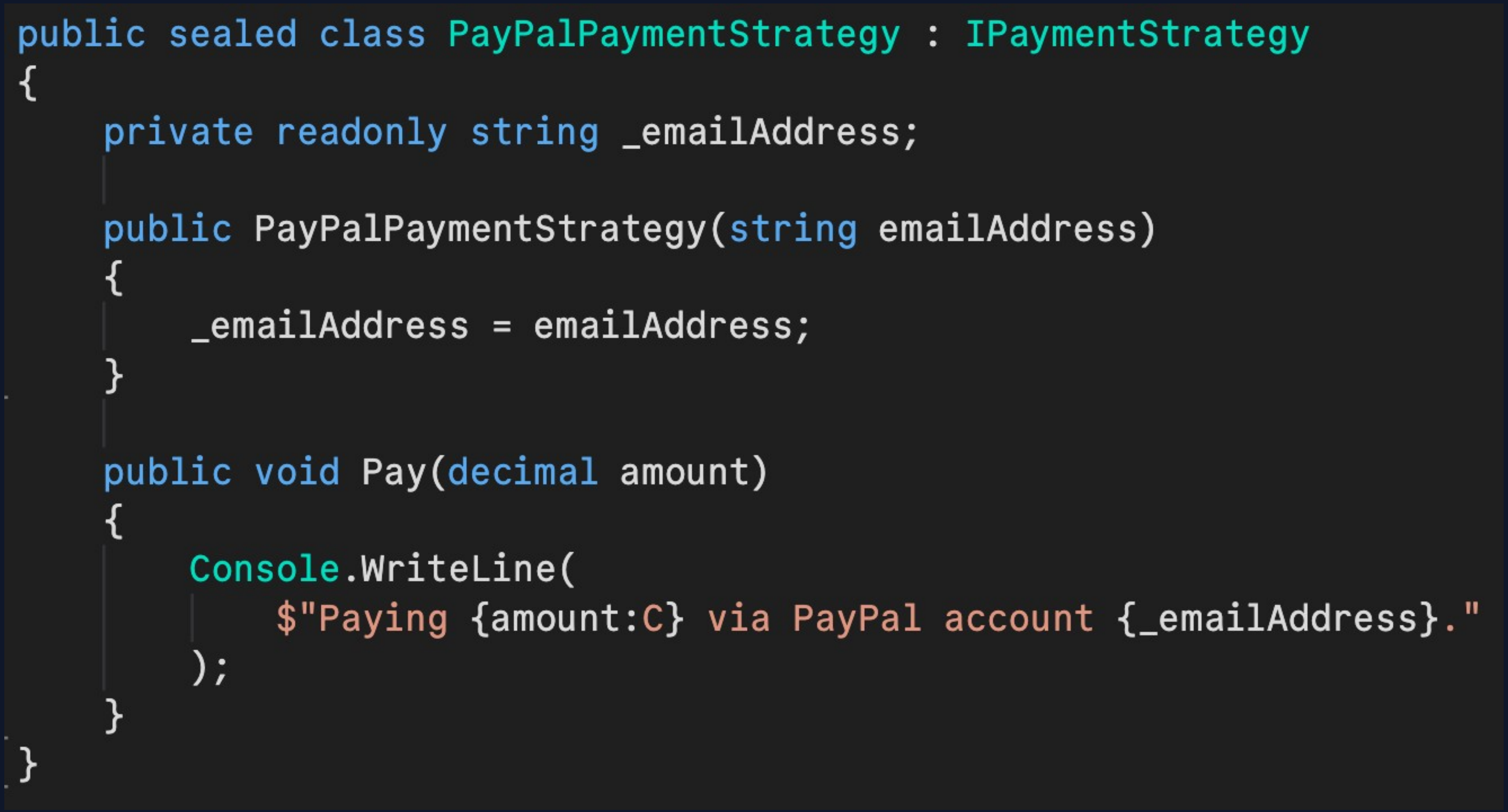


```csharp
public sealed class PayPalPaymentStrategy : IPaymentStrategy
{
    private readonly string _emailAddress;

    public PayPalPaymentStrategy(string emailAddress)
    {
        _emailAddress = emailAddress;
    }

    public void Pay(decimal amount)
    {
        Console.WriteLine(
            $"Paying {amount:C} via PayPal account {_emailAddress}."
        );
    }
}
```

C# Demo

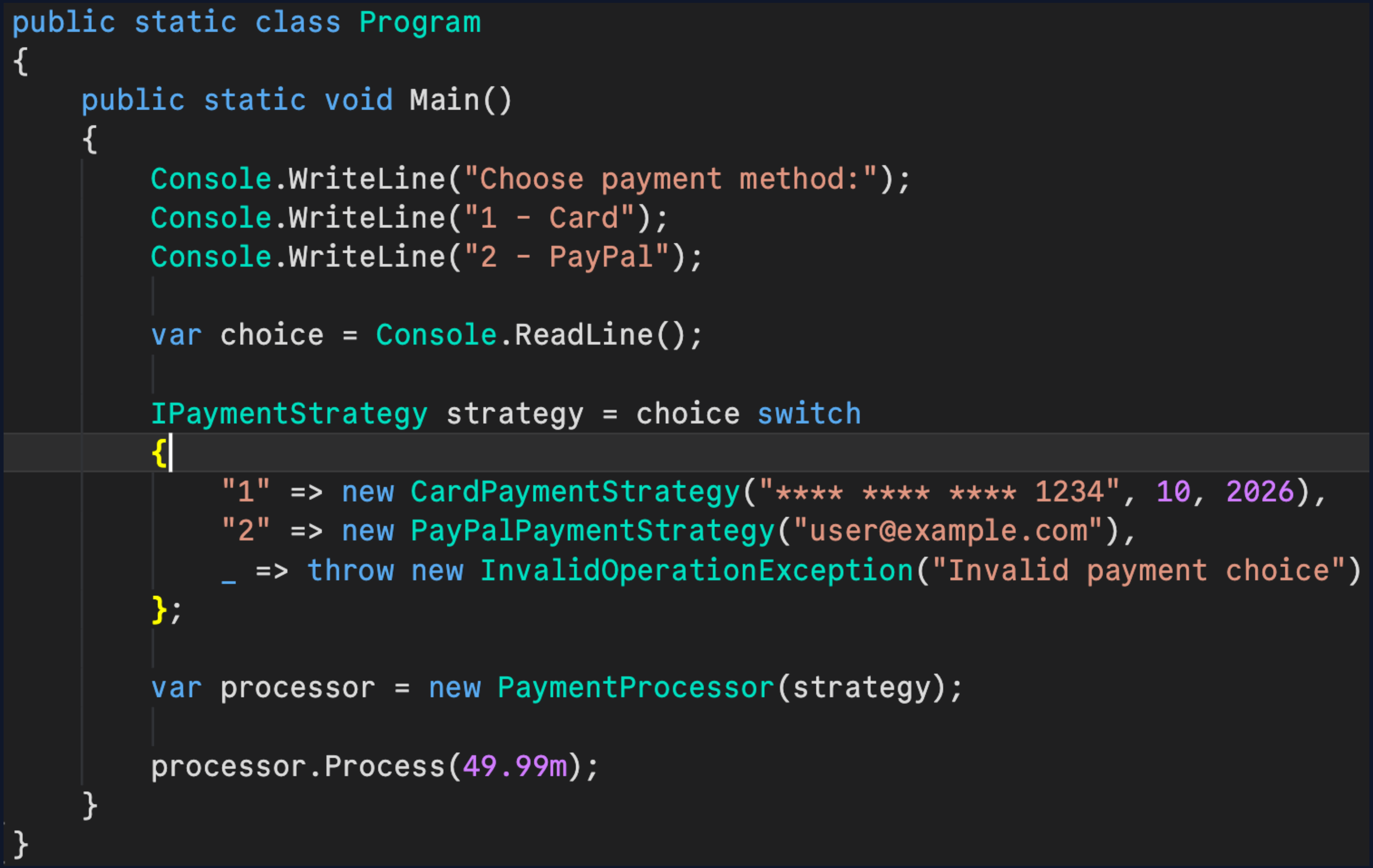


```csharp
public sealed class PayPalPaymentStrategy : IPaymentStrategy
{
    private readonly string _emailAddress;

    public PayPalPaymentStrategy(string emailAddress)
    {
        _emailAddress = emailAddress;
    }

    public void Pay(decimal amount)
    {
        Console.WriteLine(
            $"Paying {amount:C} via PayPal account {_emailAddress}."
        );
    }
}
```

C# Demo



```csharp
public static class Program
{
    public static void Main()
    {
        Console.WriteLine("Choose payment method:");
        Console.WriteLine("1 - Card");
        Console.WriteLine("2 - PayPal");

        var choice = Console.ReadLine();

        IPaymentStrategy strategy = choice switch
        {
            "1" => new CardPaymentStrategy("**** **** **** 1234", 10, 2026),
            "2" => new PayPalPaymentStrategy("user@example.com"),
            _ => throw new InvalidOperationException("Invalid payment choice")
        };

        var processor = new PaymentProcessor(strategy);

        processor.Process(49.99m);
    }
}
```

C# Demo

```
Choose payment method: 1
1 - Card
2 - PayPal
Charging $49.99 to card **** **** **** 1234 exp 10/2026
```

Architecture Comparison

Criteria	God Class (Monolith)	Strategy Pattern
SRP Compliance	None (Many reasons to change)	Full (One reason to change per class)
Testability	Hard (Must mock every payment type)	Easy (Test algorithms in isolation)
Extensibility	Hard (Modify existing method)	Easy (Add new strategy class)
Runtime Switching	Manual logic branches	Native polymorphism