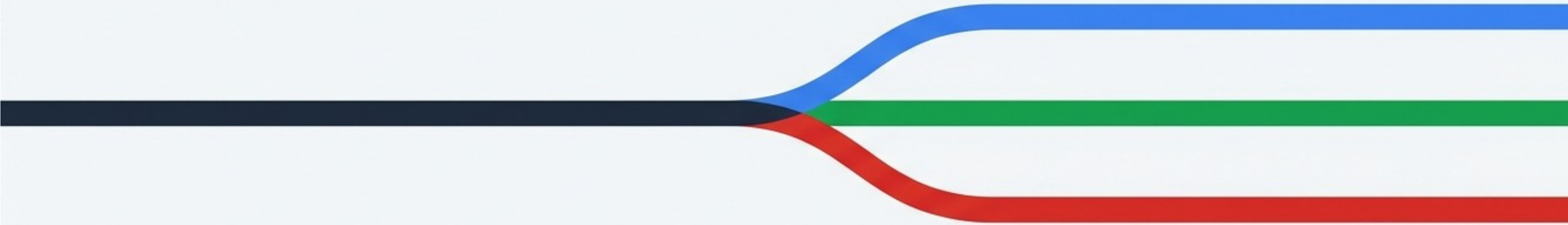


Escaping Tight Coupling with the Factory Method

A step-by-step guide to building flexible, maintainable systems by centralizing object creation.

It All Starts with One “Simple” Line of Code

```
// In our service, we need to process a payment…
var payment = new RazorPayGateway();
```

This looks harmless. But what happens when the business needs change?

The Inevitable Chaos of Hard-Coded Creation

Tomorrow, the business adds new payment gateways:

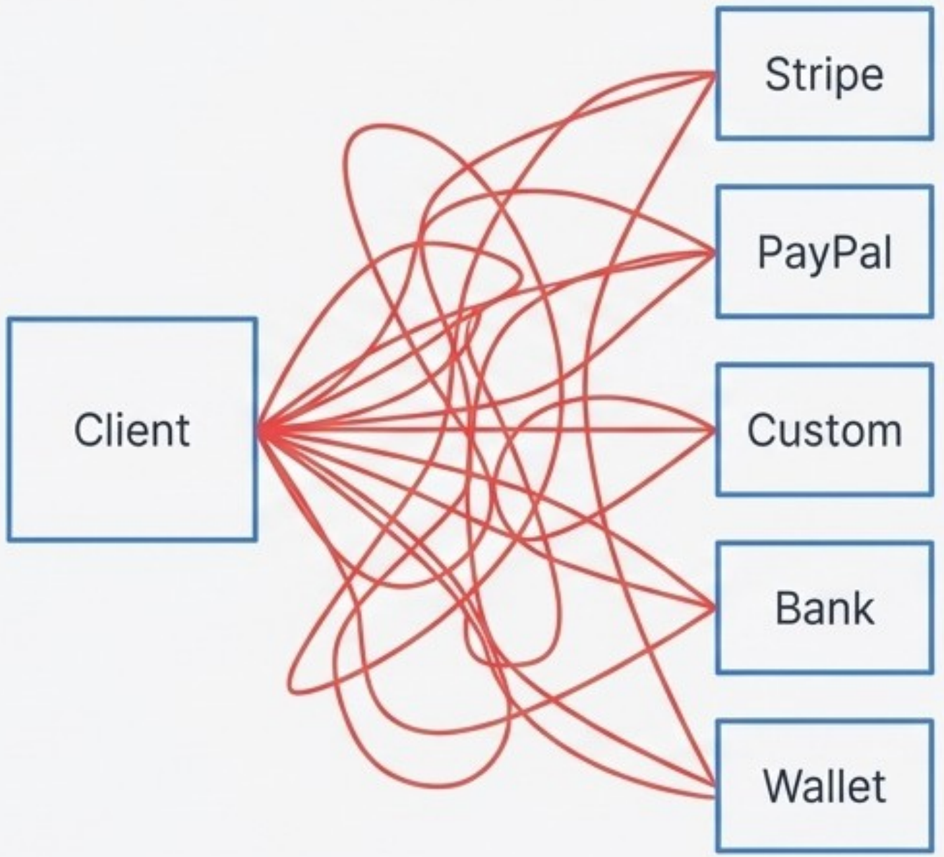
Stripe

PayPal

Custom Gateway

Bank Transfer

Wallet



Now, you must edit code everywhere. This leads to:

- ❌ **Tight Coupling:** Client is bound to concrete classes.
- ❌ **Breaking Changes:** Adding a new gateway means modifying and recompiling existing code.
- ❌ **OCP Violation:** The code is not open for extension but closed for modification.
- ❌ **Difficult Testing:** You can't easily swap in mock objects.
- ❌ **The if/else Nightmare:** Logic becomes a complex, unmanageable mess.

Think of It Like Ordering Coffee at Starbucks

You don’t go behind the counter to brew the coffee yourself.

The barista handles the complexity.

👉 **You ask WHAT you want. The Factory decides HOW to make it.**

The Solution: A 4-Step Journey to Flexible Code

The Factory Method pattern decouples the client from the creation process. We'll implement it in four guided steps.

1. **Create an Abstraction (The Menu):** Define a common interface.
2. **Implement Concrete Classes (The Drinks):** Build the specific payment types.
3. **Create the Factory (The Barista):** Centralize the creation logic.
4. **Refactor the Client (The Order):** Use the factory instead of `new`.

1 Step 1: Define the Contract with an Interface

First, we create a common interface that all our payment gateways will implement. The client will only know about this interface, not the specific classes.

```
// The 'Menu' of available actions
public interface IPayment
{
    void Pay();
}
```

This is the foundation of decoupling. The client now depends on an abstraction (`IPayment`), not a concrete detail (`StripePayment`).

Step 2: Build the Concrete Implementations

Next, we create separate classes for each payment gateway. Each one implements the `IPayment` interface, providing its own specific behavior for the `Pay()` method.

```
// The "Stripe" drink
public class StripePayment : IPayment
{
    public void Pay()
        => Console.WriteLine("Stripe payment done");
}
```

```
// The "Paypal" drink
public class PaypalPayment : IPayment
{
    public void Pay()
        => Console.WriteLine("Paypal payment done");
}
```

Step 3: Create the Factory to Centralize Creation

The factory is where the magic happens. It contains a single method that takes a parameter and decides which concrete class to create and return. All `new` calls are now hidden inside this one place.

```
// The "Barista"
public class PaymentFactory
{
    public static IPayment GetPayment(string method)
    {
        return method switch
        {
            "Stripe" => new StripePayment(),
            "Paypal" => new PaypalPayment(),
            _ => throw new ArgumentException("Invalid method")
        };
    }
}
```

The client no longer uses `new` directly. The responsibility of creation has been delegated entirely to the factory.

Step 4: The Transformed Client Code

Finally, we update the client to use our new factory. The creation logic is gone, leaving clean, intention-revealing code.

Before

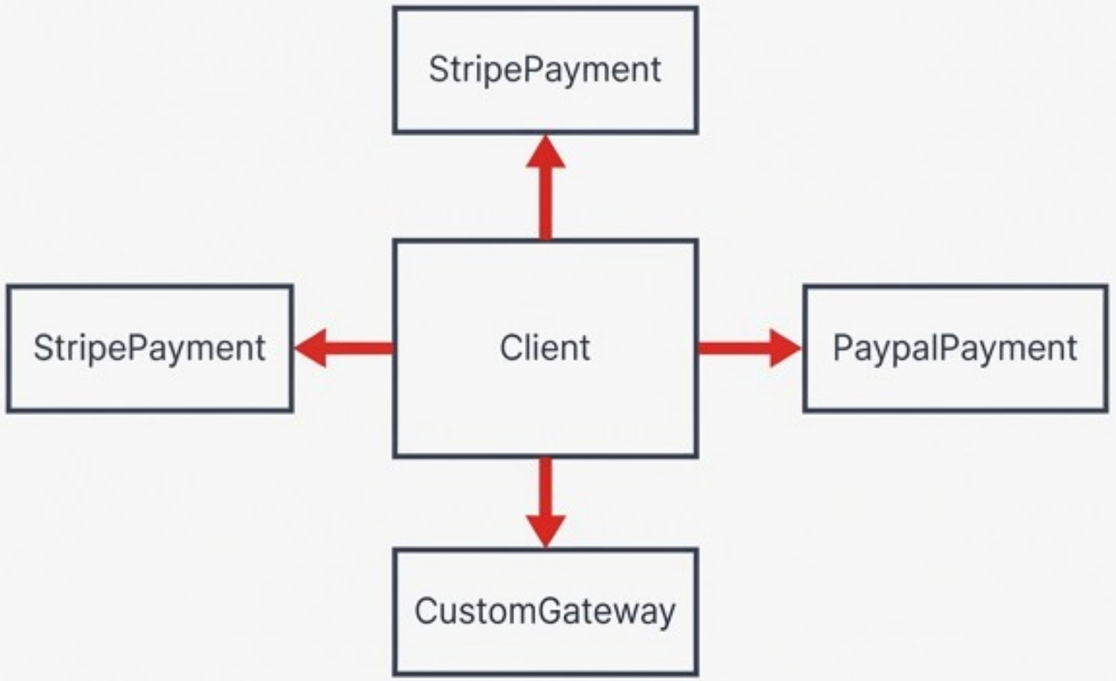
```
// var payment = new StripePayment();
// ...
// if/else hell...
```

After

```
// Just ask the factory for what you want.
var payment = PaymentFactory.GetPayment("Stripe");
payment.Pay();
```

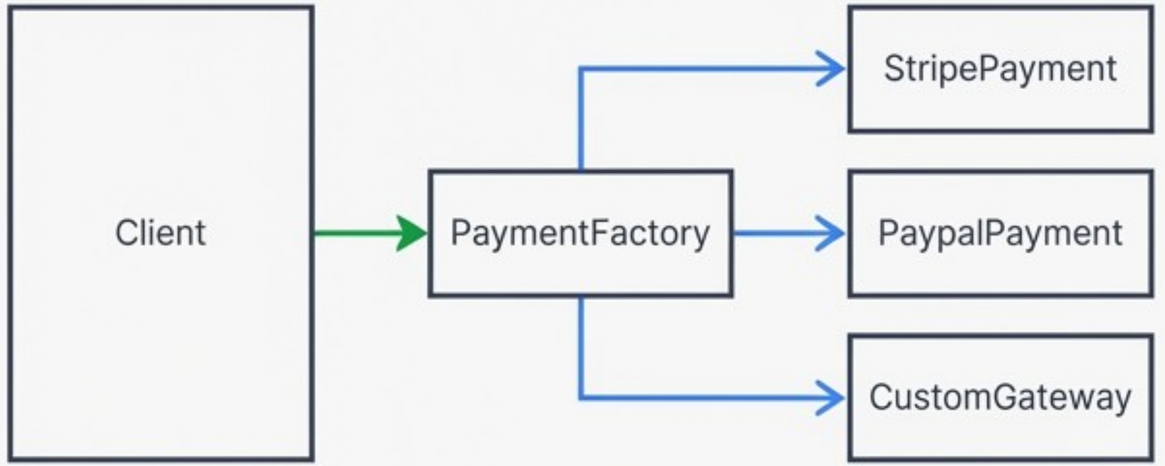
- ✓ Clean & Readable
- ✓ Flexible to Change
- ✓ Easy to Extend

What Just Happened? The Shift from Coupling to Decoupling



Before: Tight Coupling

The client knew about every single payment class. Adding a new one meant changing the client.



After: Loose Coupling

The client only knows about the `IPayment` interface and the factory. It is now completely isolated from the concrete implementations.

Beyond the Switch: A More Scalable, Registration-Based Factory

- A `switch` statement works, but it can grow ugly with dozens of types. We still have to modify the factory to add a new payment method.
- We can create a dynamic factory that allows new types to be registered at runtime.

```
// No switch, fully OCP compliant
public static class PaymentFactory
{
    private static readonly Dictionary<string, Func<IPayment>> _map = new();

    public static void Register(string name, Func<IPayment> creator)
        => _map[name] = creator;

    public static IPayment Create(string name)
        => _map[name]();
}

// Register anywhere, even in different assemblies!
PaymentFactory.Register("Stripe", () => new StripePayment());
```

Now, adding a new payment type requires **zero modification** to the factory code. This is true **Open/Closed Principle** compliance.

The Litmus Test: When to Use the Factory Method

Use this pattern when...

- ✔ Object creation logic is complex or subject to change.
- ✔ A class has multiple variations or subtypes.
- ✔ You want to decouple the client from concrete implementations.
- ✔ You need to centralize the use of the `new` keyword.
- ✔ You want to adhere strictly to the Open/Closed Principle.
- ✔ You need to simplify unit testing by easily substituting mock objects.

A Word of Caution: When a Simple `new` Is Good Enough

Avoid this pattern when...

- ❌ Only one concrete type exists and it's unlikely to change.
- ❌ The object construction process is extremely simple.
- ❌ No future flexibility is required for the object's creation.
- ❌ It adds unnecessary complexity to a small, simple project.

```
// Sometimes, this is perfectly fine. Don't over-engineer.
var user = new User();
```

The Factory Method: A Summary

Item	Meaning
Problem	Hard-coded object creation (`new`) causes tight coupling and rigidity.
Solution	Move creation logic into a dedicated factory method or class.
What Changes	The client code no longer uses `new` and depends on an interface, not concrete classes.
Core Benefits	Creates a flexible, testable, and maintainable system that can grow over time.

Key Principles Applied

Key principles applied for aligh-end asthetic.

OCP — Open/Closed Principle: You can introduce new implementations without modifying existing client or factory code (especially with the registration method).

DIP — Dependency Inversion Principle: High-level modules (the client) do not depend on low-level modules (the concrete implementations). Both depend on abstractions (the interface).