# الگوی Factory Method در #C

## تعریف الگوی Factory Method

الگوی **Factory Method** یک الگوی طراحی **خلاقی** (Creational) است که یک رابط برای ایجاد اشیا در یک سوپرکلاس تعریف می‌کند، اما اجازه می‌دهد زیرکلاس‌ها نوع شیءای که ایجاد می‌شود را تغییر دهند. این الگو همچنین به عنوان **الگوی سازنده مجازی** (Virtual Constructor) نیز شناخته می‌شود.

## مشکل اصلی که Factory Method حل می‌کند

1. **وابستگی مستقیم**: وقتی کد به کلاس‌های خاصی وابسته است و نیاز به ایجاد نمونه‌های مختلف دارد.
2. **انعطاف‌پذیری**: وقتی نمی‌دانیم دقیقاً چه نوع شیءای باید ایجاد شود یا ممکن است در آینده تغییر کند.
3. **تست‌پذیری**: وقتی نیاز داریم ایجاد اشیا را برای تست‌های واحد کنترل کنیم.

## ساختار پایه Factory Method

// 1. رابط محصول (Product)

public interface IProduct { string Operation(); }

// 2. پیاده‌سازی‌های مختلف محصول

public class ConcreteProductA : IProduct {

public string Operation() => "Result of ConcreteProductA";

}

public class ConcreteProductB : IProduct {

public string Operation() => "Result of ConcreteProductB";

}

// 3. کلاس Creator (سازنده)

public abstract class Creator {

// Factory Method

public abstract IProduct FactoryMethod();

public string SomeOperation() {

var product = FactoryMethod();

return $"Creator: {product.Operation()}";

}

}

// 4. پیاده‌سازی‌های Creator

public class ConcreteCreatorA : Creator {

public override IProduct FactoryMethod() => new ConcreteProductA();

}

public class ConcreteCreatorB : Creator {

public override IProduct FactoryMethod() => new ConcreteProductB();

}

## نحوه استفاده از الگو

class Program

{

static void Main(string[] args) {

Console.WriteLine("App: Launched with ConcreteCreatorA.");

ClientCode(new ConcreteCreatorA());

Console.WriteLine("App: Launched with ConcreteCreatorB.");

ClientCode(new ConcreteCreatorB());

}

public static void ClientCode(Creator creator) {

Console.WriteLine($"Client: {creator.SomeOperation()}");

}

}

## مزایای Factory Method

1. **جدا کردن کد ایجاد کننده از کد محصول**: کد کلاینت به کلاس‌های محصول خاص وابسته نمی‌شود.
2. **انعطاف‌پذیری**: اضافه کردن محصولات جدید بدون تغییر کد موجود امکان‌پذیر است.
3. **تست‌پذیری**: می‌توان از Mock objects در تست‌ها استفاده کرد.
4. **اصل Single Responsibility**: منطق ایجاد شیء در یک متمرکز می‌شود.
5. **اصل Open/Closed**: می‌توان محصولات جدید را بدون تغییر کد موجود اضافه کرد.

## معایب Factory Method

1. **پیچیدگی**: ممکن است منجر به ایجاد کلاس‌های زیاد شود.
2. **نیاز به زیرکلاس‌سازی**: برای هر نوع محصول جدید نیاز به ایجاد یک زیرکلاس Creator است.
3. **عدم شفافیت**: ممکن است تشخیص اینکه کدام Factory کدام محصول را می‌سازد دشوار باشد.

## مثال کاربردی: سیستم پرداخت

// رابط پرداخت

public interface IPayment {

void ProcessPayment(decimal amount);

}

// پیاده‌سازی‌های پرداخت

public class CreditCardPayment : IPayment {

public void ProcessPayment(decimal amount)

=> Console.WriteLine($"Processing credit card payment: {amount}");

}

public class PayPalPayment : IPayment {

public void ProcessPayment(decimal amount)

=> Console.WriteLine($"Processing PayPal payment: {amount}");

}

// Creator

public abstract class PaymentFactory {

public abstract IPayment CreatePayment();

public void ProcessOrder(decimal amount) {

var payment = CreatePayment();

payment.ProcessPayment(amount);

}

}

// Concrete Creators

public class CreditCardPaymentFactory : PaymentFactory {

public override IPayment CreatePayment() => new CreditCardPayment();

}

public class PayPalPaymentFactory : PaymentFactory {

public override IPayment CreatePayment() => new PayPalPayment();

}

## استفاده در دنیای واقعی در #C

1. **ASP.NET Core Dependency Injection**:

services.AddTransient<IPayment, CreditCardPayment>();

1. **System.Data.Common.DbProviderFactories**:

DbProviderFactory factory = DbProviderFactories.GetFactory("System.Data.SqlClient");

## تفاوت با Abstract Factory

* **Factory Method**: ایجاد یک محصول از طریق ارث‌بری (وراثت)
* **Abstract Factory**: ایجاد خانواده‌ای از محصولات از طریق ترکیب (کمپوزیشن)

## بهترین روش‌های پیاده‌سازی

1. از **رابط‌ها (Interfaces)** برای محصولات استفاده کنید.
2. **پارامتری کردن** Factory Method می‌تواند مفید باشد.
3. برای کاهش تعداد کلاس‌ها می‌توان از **متدهای ساده** به جای زیرکلاس‌ها استفاده کرد.
4. در #C مدرن می‌توان از **Delegateها** به عنوان Factory Method استفاده کرد:

public class PaymentProcessor

{

private readonly Func<IPayment> \_paymentFactory;

public PaymentProcessor(Func<IPayment> paymentFactory) {

\_paymentFactory = paymentFactory;

}

public void ProcessOrder(decimal amount) {

var payment = \_paymentFactory();

payment.ProcessPayment(amount);

}

}

این الگو زمانی مناسب است که یک سیستم باید مستقل از نحوه ایجاد، ترکیب و نمایش اجزای آن باشد یا وقتی کلاس‌ها نمی‌دانند چه کلاس‌هایی باید ایجاد کنند.

-------------------------------------------------------------------------------------------------------