Patrón de Diseño Factory Method: Creación Flexible de Objetos

El patrón de diseño Factory Method es una poderosa herramienta dentro del paradigma de la programación orientada a objetos. Como patrón creacional, su objetivo principal es encapsular la lógica de creación de objetos, delegando esta responsabilidad a las subclases. Esto permite a un sistema crear objetos sin especificar la clase concreta del objeto que se creará, promoviendo así la flexibilidad y la mantenibilidad.

Al aplicar este patrón, nos adherimos a principios de diseño fundamentales como el Principio de Abierto/Cerrado (O/C), lo que facilita la extensión del sistema con nuevos tipos de productos sin modificar el código existente del cliente. Esto contrasta con enfoques más rígidos que a menudo requieren el uso de múltiples condicionales (como sentencias `if/else` o `switch`) para instanciar clases, lo que puede llevar a un código difícil de mantener y extender.

¿Cómo Funciona el Factory Method? Estructura y Roles Clave

El patrón Factory Method se basa en cuatro componentes principales que trabajan en conjunto para lograr una creación de objetos flexible y desacoplada:

Creator (Creador)

Declara el método fábrica, que abstractamente se encarga de devolver un objeto del tipo `Product`. Puede contener una implementación predeterminada de este método o dejarlo abstracto.

ConcreteCreator (Creador Concreto)

Extiende la clase `Creator` y sobrescribe el `factory_method` para producir un `ConcreteProduct` específico. Cada creador concreto es responsable de instanciar un tipo particular de producto.

Product (Producto)

Define la interfaz o clase abstracta común para los objetos que serán creados por el método fábrica. Todos los productos concretos deben implementar esta interfaz o heredar de esta clase base.

ConcreteProduct (Producto Concreto)

Son las implementaciones específicas de la interfaz `Product`. Cada `ConcreteProduct` proporciona una implementación concreta de las operaciones declaradas en `Product`.

Ejemplo: Factory Method en Acción

Imaginemos que necesitamos un sistema para gestionar la entrega de paquetes, que puede ser por camión o por barco, y en el futuro quizás por avión. Usando Factory Method, podemos crear los objetos de transporte de manera flexible:

```
from abc import ABC, abstractmethod

# Product: Interfaz de transporte
class Transport(ABC):
    @abstractmethod
    def deliver(self) -> str:
        pass

# ConcreteProduct: Camión
class Truck(Transport):
    def deliver(self) -> str:
        return "Entrega por tierra en camión."

# ConcreteProduct: Barco
class Ship(Transport):
    def deliver(self) -> str:
        return "Entrega por mar en barco."

# Creator: Clase base para la fábrica de transporte
class Logistics(ABC):
    @abstractmethod
    def create_transport(self) -> Transport:
        pass

    def plan_delivery(self) -> str:
        transport = self.create_transport()
        result = f"Logística: Preparando la entrega. {transport.deliver()}"
        return result

# ConcreteCreator: Fábrica de logística terrestre
class RoadLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Truck()

# ConcreteCreator: Fábrica de logística marítima
class SeaLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Ship()

# Uso del patrón
def client_code(logistics: Logistics):
    print(f"Cliente: No conozco la clase de logística, pero funciona.\n"
          f"{logistics.plan_delivery()}")

print("Aplicación: Lanzando logística terrestre.")
client_code(RoadLogistics())

print("\nAplicación: Lanzando logística marítima.")
client_code(SeaLogistics())
```

Este ejemplo demuestra cómo el cliente interactúa con la interfaz `Logistics` sin necesidad de saber si la entrega se realizará por `Truck` o `Ship`. Si se añade un nuevo método de transporte (ej., `AirLogistics` y `Plane`), solo se necesitaría añadir nuevas subclases sin modificar el código existente de la logística base o del cliente.