Simple Factory · Factory Method · Abstract Factory

05 Sep 2025

Agenda

- *Crear objetos sin acoplar a clases concretas*
- Motivación: el problema de `**new**` y el acoplamiento
- Simple Factory (idioma)
- Factory Method (GoF)
- Dependency Inversion Principle
- Abstract Factory (GoF)
- Caso Pizza: flujo y diseño
- Comparaciones, ventajas y trade-offs
- Trabajo Práctico + Quiz

Motivación: ¿qué está mal con `**new**`?

- *Separar lo que cambia (creación) de lo que permanece (uso)*
- Cuando ves `**new**`, estás acoplándote a una clase concreta
- Cambios futuros ⇒ reabrir código en varios lugares
- Queremos ‘programar contra abstracciones’, pero hay que instanciar
- ➜ Encapsular la creación para cumplir OCP (Open–Closed)

Identificar lo que varía

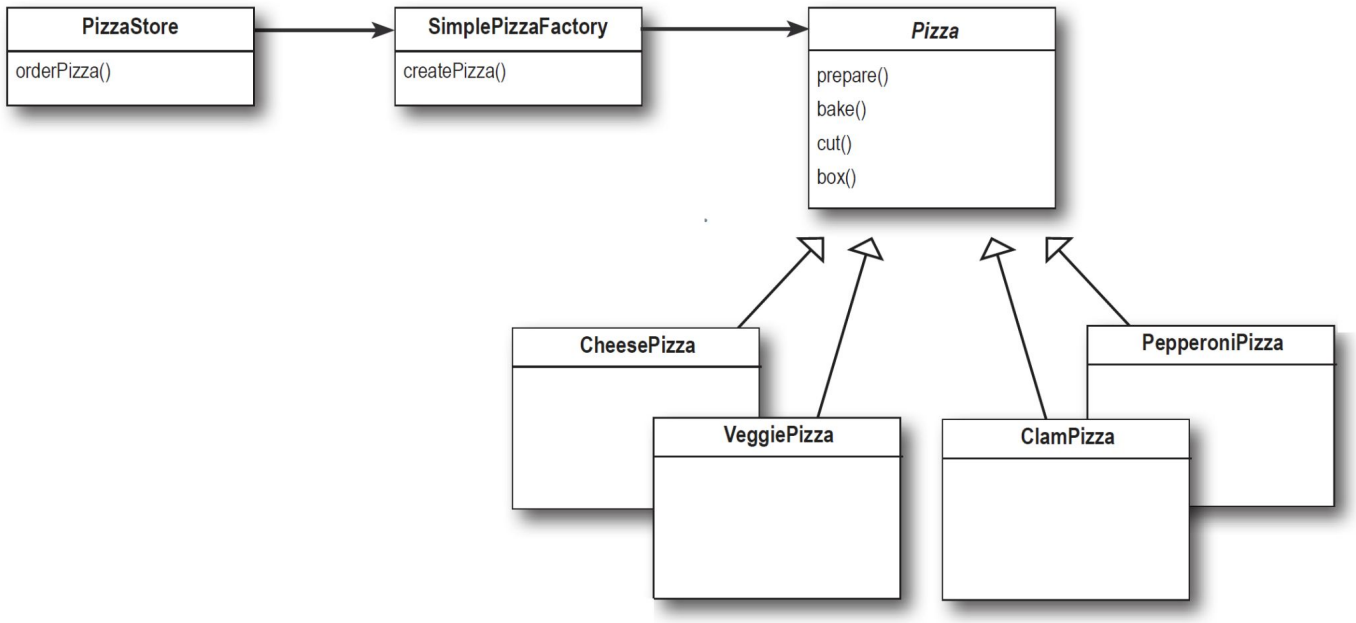
- *‘Encapsulate what varies’ aplicado a instanciación*
- En PizzaStore: recetas/estilos cambian con el tiempo
- El pipeline de preparación (prepare/bake/cut/box) casi no cambia
- ➔ Extraer/encapsular la creación de productos

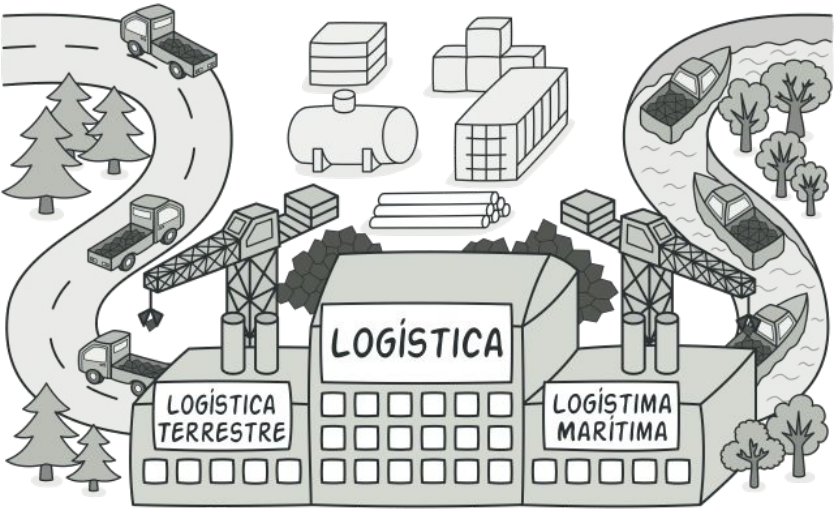
Simple Factory

- **Idea**
  - Centralizar la creación en un objeto/método
  - Clientes piden `create(type)` y reciben un `Product`
  - Permite 1 solo lugar de cambios en instanciación
- **Notas**
  - No es un patrón oficial de la Gang of Four, sino una solución práctica. Es una clase o función que crea objetos basándose en un parámetro, sin usar una jerarquía de clases abstractas. Es más directa y menos formal.
  - Sigue habiendo lógica condicional para elegir concreto

Simple Factory — ejemplo

```
class SimplePizzaFactory:
    def create_pizza(self, kind: str) -> Pizza:
        if kind == "cheese":   return CheesePizza()
        if kind == "veggie":   return VeggiePizza()
        if kind == "clam":     return ClamPizza()
        if kind == "pepperoni":return PepperoniPizza()
        raise ValueError("Tipo inválido")
```





Factory Method

Factory Method es un patrón de diseño creacional que define una interfaz para crear objetos, y permite a las subclases decidir el tipo de objetos que se crearán.

Problema: Logística creciente

App solo camiones

Imagina que estás creando una aplicación de gestión logística. La primera versión de tu aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte de tu código se encuentra dentro de la clase **Camión**.

Demanda de Transporte por mar

La aplicación se vuelve bastante popular y cada día recibes muchas peticiones de empresas de transporte marítimo para que **incorpores** la logística por mar a la aplicación.

Complejidad de cambios

Para añadir barcos a la aplicación habría que hacer **cambios** en toda la base del código. Además, si más tarde decides añadir **otro tipo de transporte** a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.

Solución: Factory Method

1 Reemplazar la Construcción Directa

En lugar de llamar al operador new para construir objetos directamente, se llama a un método de una fábrica especial

2 El Factory Method Retorna Productos

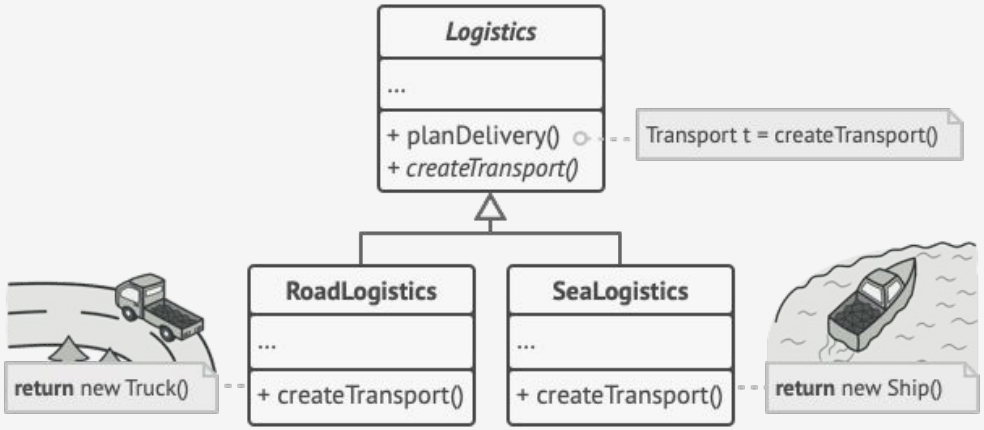
Los objetos devueltos por el método fábrica a menudo se denominan ***productos***.

3 Las Subclases Sobrescriben Factory Method

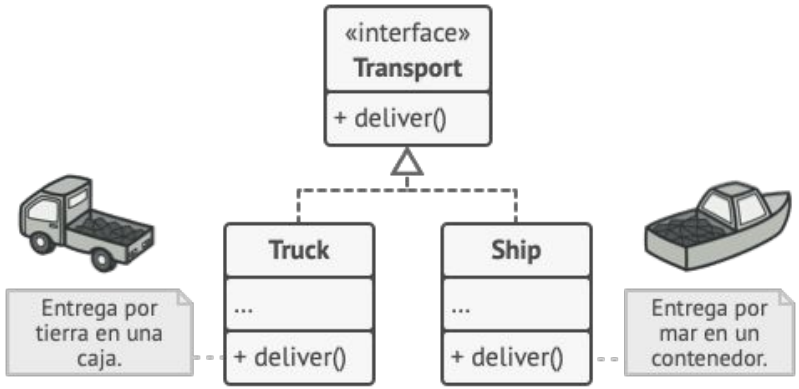
Ahora puedes **sobrescribir** el método fábrica en una subclase y definir la clase concreta de los productos creados por el método.

4 Interfaz Común Requerida

Las subclases sólo pueden devolver productos de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.



Ejemplo: Interfaz Transporte



1

Interfaz Transporte

La clase **Camión** y la clase **Barco** deben implementar la interfaz **Transporte**, que declara un método llamado **entrega.**

2

Truck Implementación

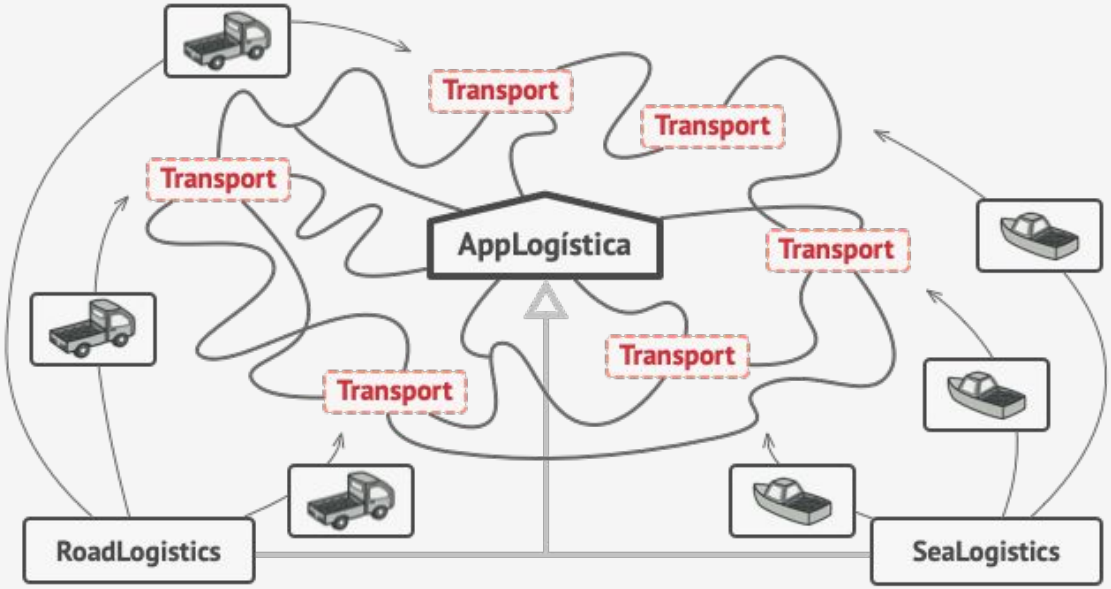
Los camiones entregan su carga por **tierra**.

3

Ship Implementación

Los barcos lo hacen por **mar**.

Interacción en el Código del Cliente



RoadLogistics

El método fábrica dentro de la clase **LogísticaTerrestre** devuelve objetos de tipo camión.

SeaLogistics

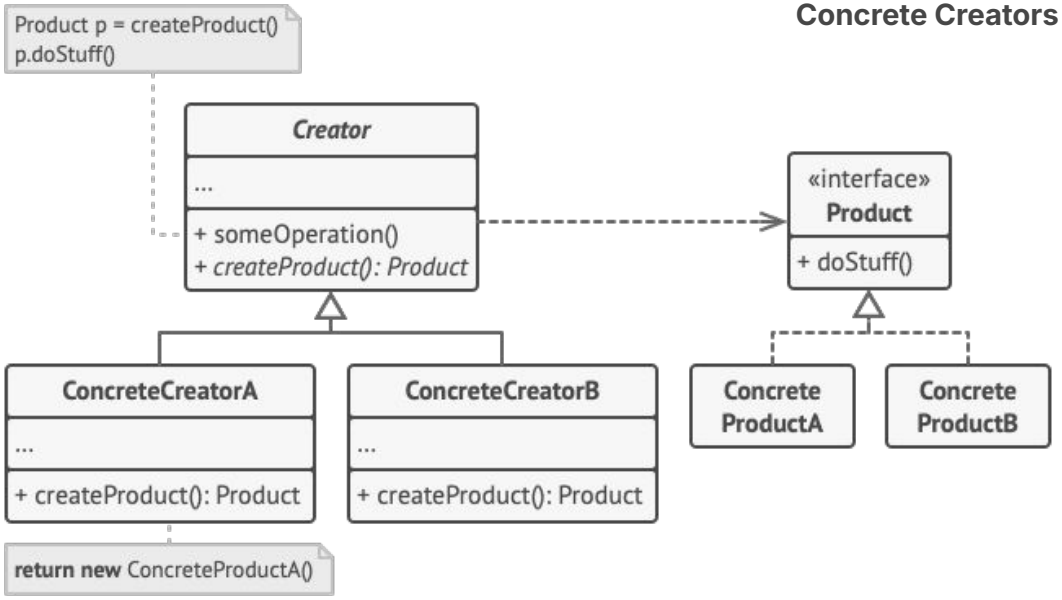
El método fábrica de la clase **LogísticaMarítima** devuelve barcos.

App del Cliente

El código cliente trata todos los productos como la clase abstracta **Transporte**, utilizando el método **entrega** sin conocer su implementación.

Estructura del patrón Factory Method

Product	declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.
Concrete Products	distintas implementaciones de la interfaz de producto.
Creator	declara el método fábrica que devuelve nuevos objetos de producto. El tipo de retorno de este método debe coincidir con la interfaz de producto.
Concrete Creators	sobrescriben el Factory Method base, de modo que devuelva un tipo diferente de producto. .



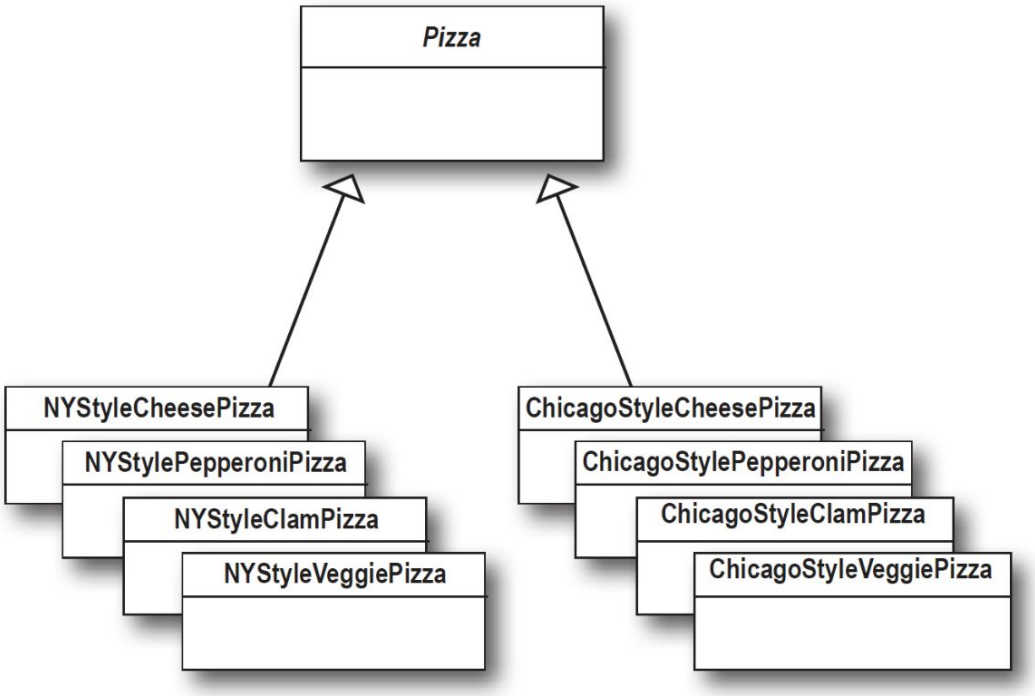
Factory Method

- **Definición (GoF)**
  - Define una interfaz para crear un objeto
  - Deja a las subclases decidir qué clase instanciar
  - El ‘método fábrica’ suele ser abstracto/protegido
- **Estructura**
  - Creator (abstracto) con `factoryMethod()`
  - ConcreteCreator implementa `factoryMethod()`
  - Products comparten interfaz común

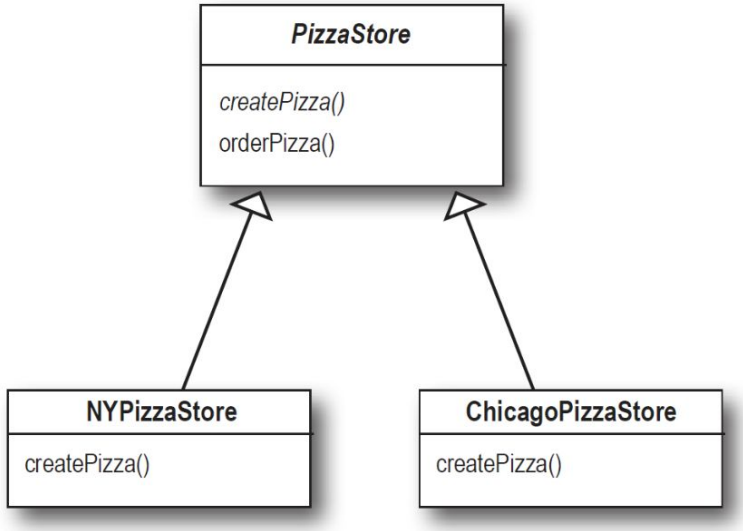
Factory Method — esqueleto (pseudocódigo)

```
class PizzaStore:  # Creator
    def order_pizza(self, kind: str) -> Pizza:
        pizza = self.create_pizza(kind)
        pizza.prepare(); pizza.bake(); pizza.cut(); pizza.box()
        return pizza
    def create_pizza(self, kind: str) -> Pizza:  # factory method
        raise NotImplementedError

class NYPizzaStore(PizzaStore):  # ConcreteCreator
    def create_pizza(self, kind: str) -> Pizza:
        if kind == "cheese": return NYStyleCheesePizza()
        # ... otras variantes

class ChicagoPizzaStore(PizzaStore):  # ConcreteCreator
    def create_pizza(self, kind: str) -> Pizza:
        if kind == "cheese": return ChicagoStyleCheesePizza()
        # ... otras variantes
```

The Product classes

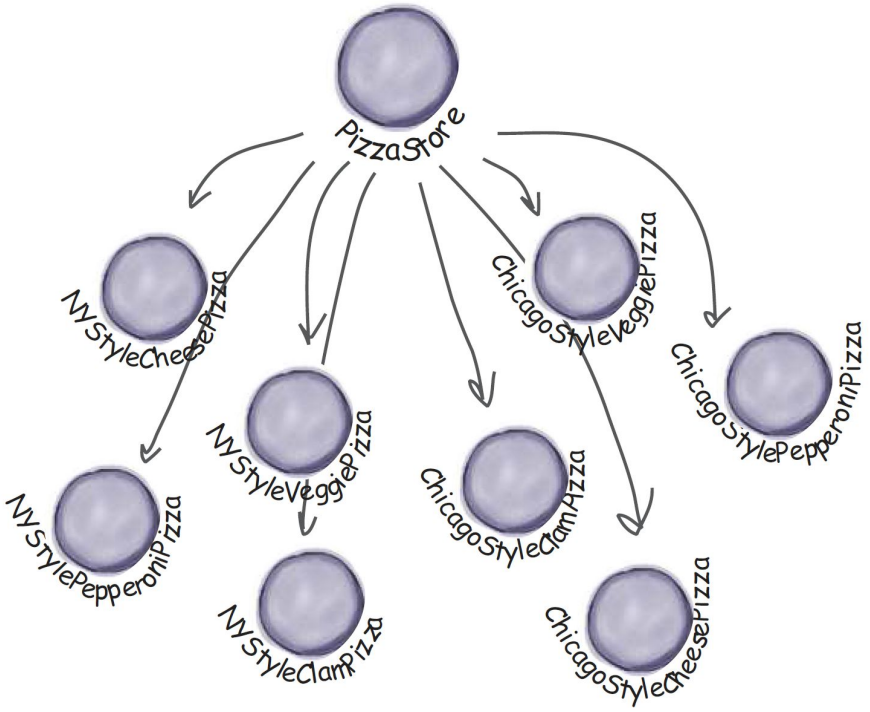


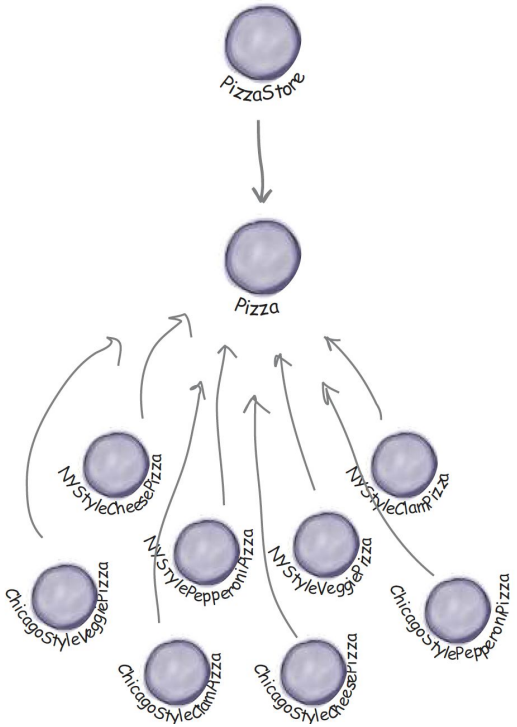
The Creator classes



Dependency Inversion Principle (DIP)

- *Invirtiendo dependencias para reducir fragilidad*
- Depender de abstracciones, no de clases concretas
- Módulos de alto nivel y bajo nivel dependen de la misma abstracción
- Factory Method ayuda a cumplir DIP (reduce dependencias directas)

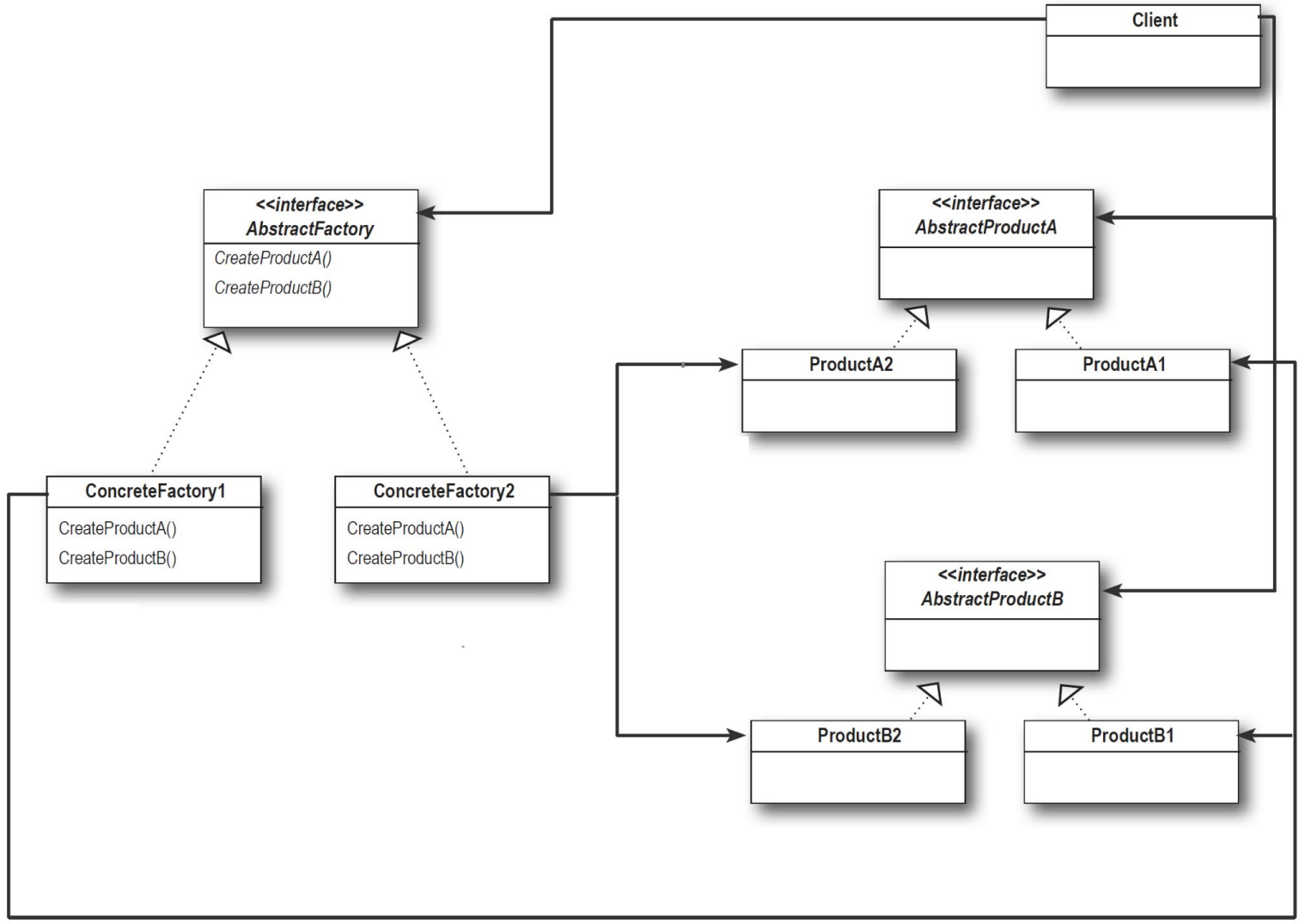




Abstract Factory

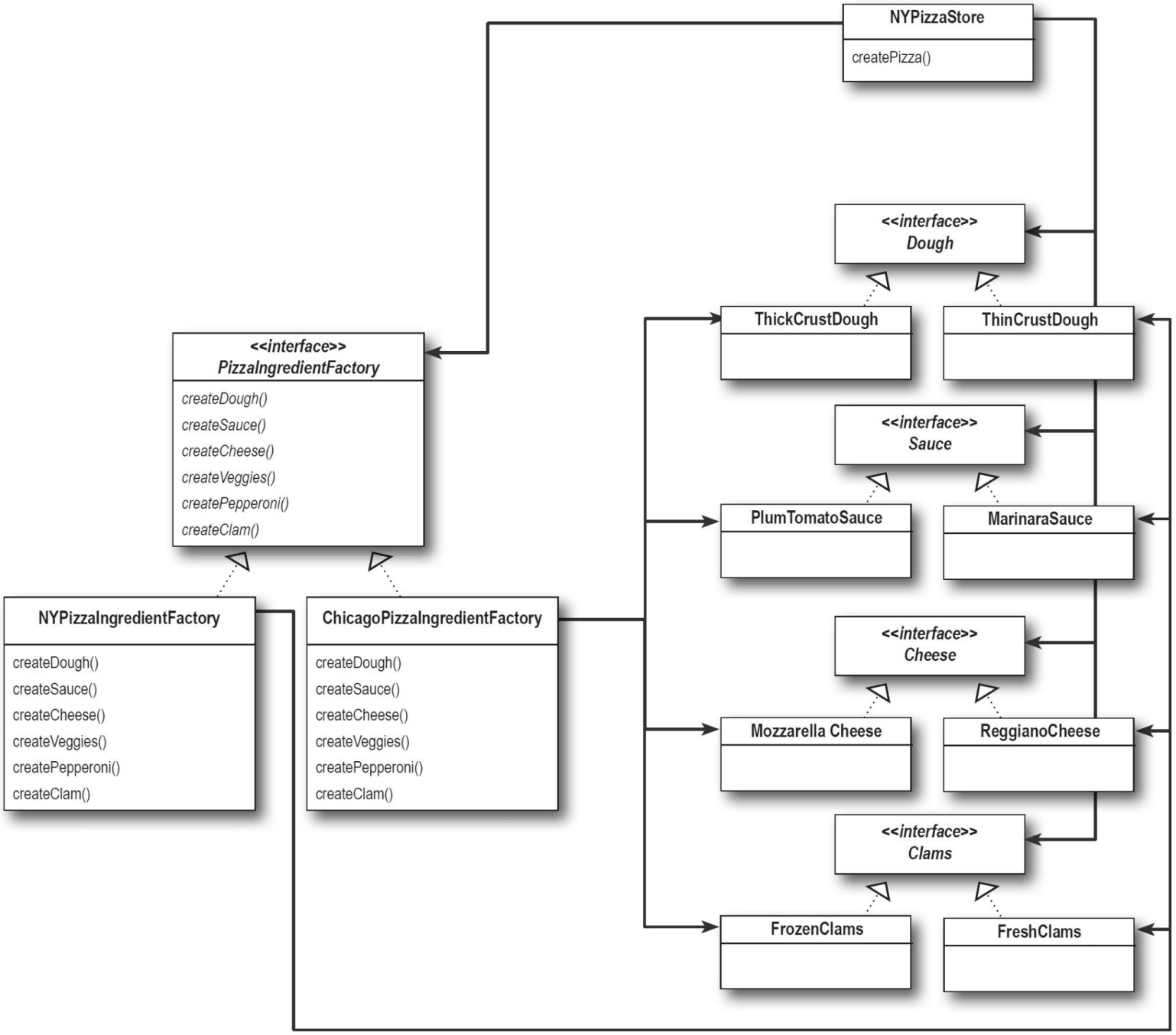
- **Definición (GoF)**
  - Provee una interfaz para crear **familias** de productos relacionados, sin especificar sus clases concretas
- **Cuándo usar**
  - Variantes por plataforma/región/look-and-feel
  - Necesitás consistencia entre productos de una misma familia

Abstract Factory



Abstract Factory — interfaz de ingredientes (pseudocódigo)

```
class PizzaIngredientFactory(Protocol):
    def create_dough(self) -> Dough: ...
    def create_sauce(self) -> Sauce: ...
    def create_cheese(self) -> Cheese: ...
    def create_veggies(self) -> list[Veggie]: ...
    def create_pepperoni(self) -> Pepperoni: ...
    def create_clam(self) -> Clams: ...

class NYPizzaIngredientFactory(PizzaIngredientFactory):
    def create_dough(self): return ThinCrustDough()
    def create_sauce(self): return MarinaraSauce()
    def create_cheese(self): return ReggianoCheese()
    # etc.
```


NYPizzaStore
createPizza()
<<interface>>
Dough
ThickCrustDough
ThinCrustDough
<<interface>>
PizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
<<interface>>
Sauce
PlumTomatoSauce
MarinaraSauce
NYPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
ChicagoPizzaIngredientFactory
createDough()
createSauce()
createCheese()
createVeggies()
createPepperoni()
createClam()
<<interface>>
Cheese
Mozzarella Cheese
ReggianoCheese
<<interface>>
Clams
FrozenClams
FreshClams

Usando la fábrica de ingredientes en `prepare()`

```
class CheesePizza(Pizza):
    def __init__(self, ingredient_factory:
PizzaIngredientFactory):
        self.factory = ingredient_factory
    def prepare(self):
        print(f"Preparing {self.name}")
        self.dough  = self.factory.create_dough()
        self.sauce  = self.factory.create_sauce()
        self.cheese = self.factory.create_cheese()
```

Comparativa rápida

- **Simple Factory**
  - Encapsula instanciación en un único objeto/método
  - Sencillo; buen primer paso; no es patrón GoF
- **Factory Method / Abstract Factory**
  - FM: delega en subclases qué crear; encaja en un framework
  - AF: crea familias de productos; consistencia entre variantes

Ventajas y trade-offs

- *Elegir el nivel de abstracción adecuado*
- ✓ Menos acoplamiento a concretos; cumple OCP y DIP
- ✓ Reutilización de flujo común (order/prepare/bake/cut/box)
- ⚠️ Más clases/jerarquías; puede aumentar complejidad
- ⚠️ Parámetros ‘type-safe’ (usar enums/constantes)

Caso Pizza — flujo de orden

- *El Store queda desacoplado de clases concretas*
- Cliente elige Store (NY/Chicago) ➜ decide estilo
- Store.orderPizza(kind) invoca createPizza(kind)
- Se prepara la pizza y se devuelve al cliente

Trabajo Práctico (TP)

- *Entrega: código + README con decisiones de diseño*
- 1) Implementar Simple Factory (pizzas básicas)
- 2) Migrar a Factory Method (NYPizzaStore/ChicagoPizzaStore)
- 3) Extender con Abstract Factory de ingredientes por región
- 4) Tests: flujos de orden y consistencia de ingredientes

Mini-quiz

- *5 minutos de debate*
- ¿Cuándo usarías Simple Factory vs Factory Method?
- ¿Qué problema resuelve Abstract Factory?
- ¿Cómo ayuda DIP en estos diseños?

Referencias y lecturas

- *Profundizar en variantes parametrizadas y enums*
- Head First Design Patterns, Cap. 4 — The Factory Pattern (pizza, DIP, AF)
- Documentación/guía de la cátedra y ejemplos de clase