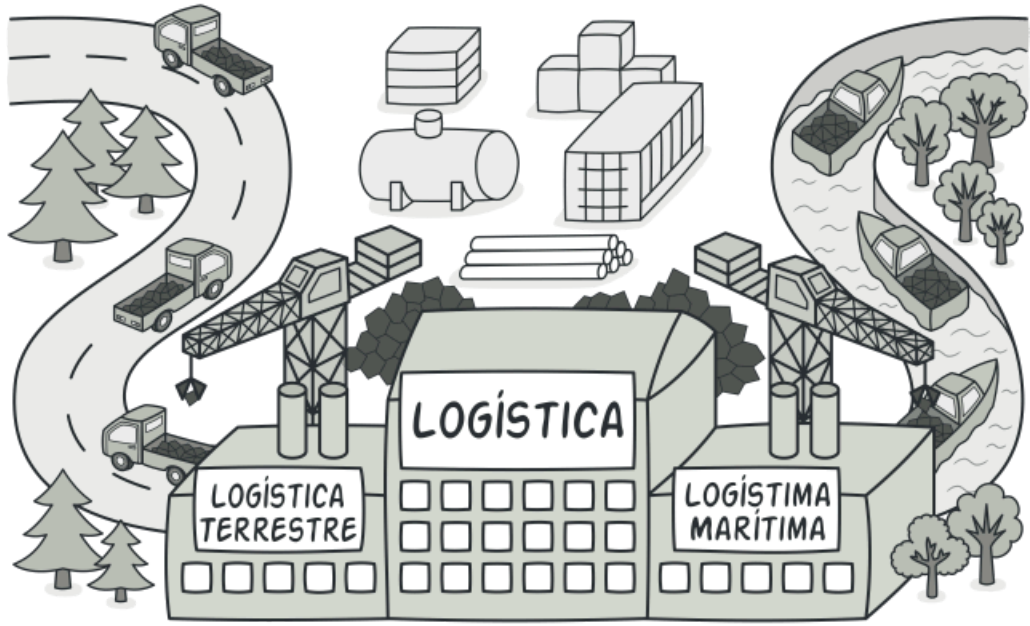
Patrón: Factory Method

Patrón creacional (POO)

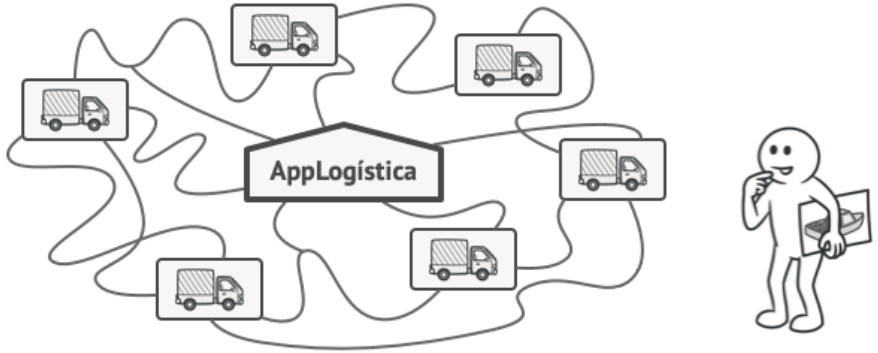
Fuente principal:
https://refactoring.guru/es/design-patterns/factory-method

Propósito

Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



Imágenes extraídas de Refactoring.Guru



Problema

Imagina una aplicación de gestión logística. Inicialmente solo maneja transporte en **Camión**. Al volverse popular, necesitas añadir transporte **Marítimo**.

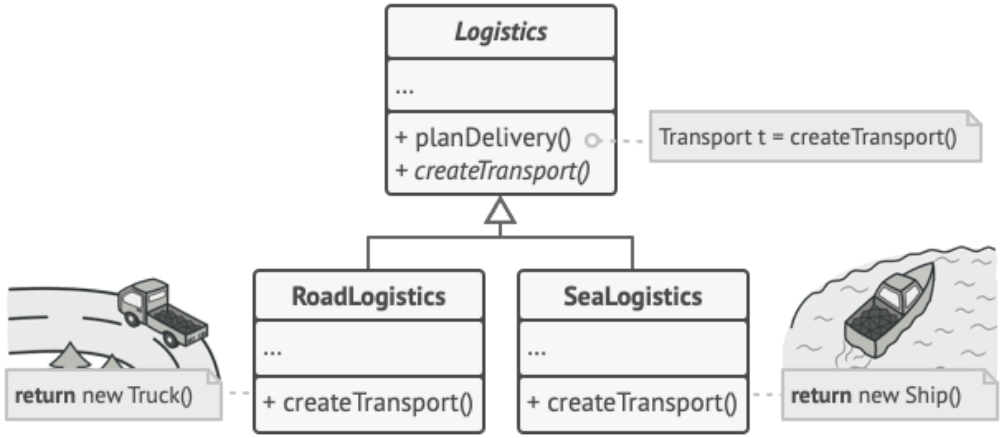
- El código actual está acoplado a la clase `Camión`.
- Añadir `Barco` requeriría cambios en todo el código.
- Plagado de condicionales.

Imágenes extraídas de Refactoring.Guru

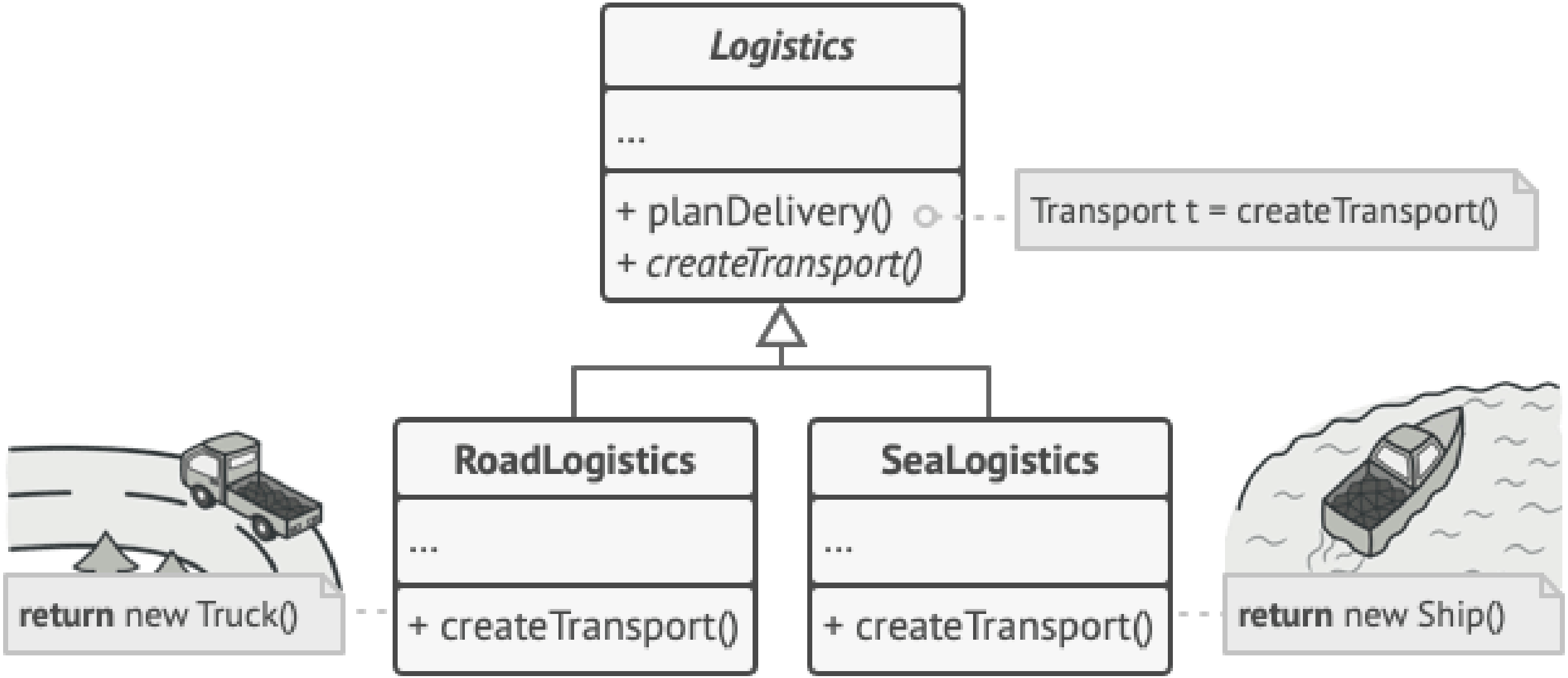
Solución

El patrón sugiere invocar a un **método fábrica** en lugar de llamar al operador `new` directamente.

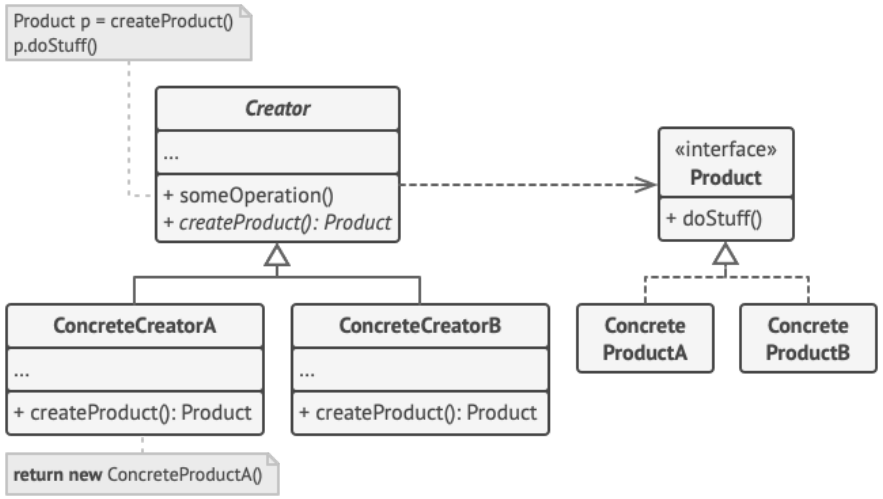
- Los objetos se siguen creando con `new`, pero desde el método fábrica.
- Las subclases pueden alterar la clase de los objetos devueltos.



Imágenes extraídas de Refactoring.Guru



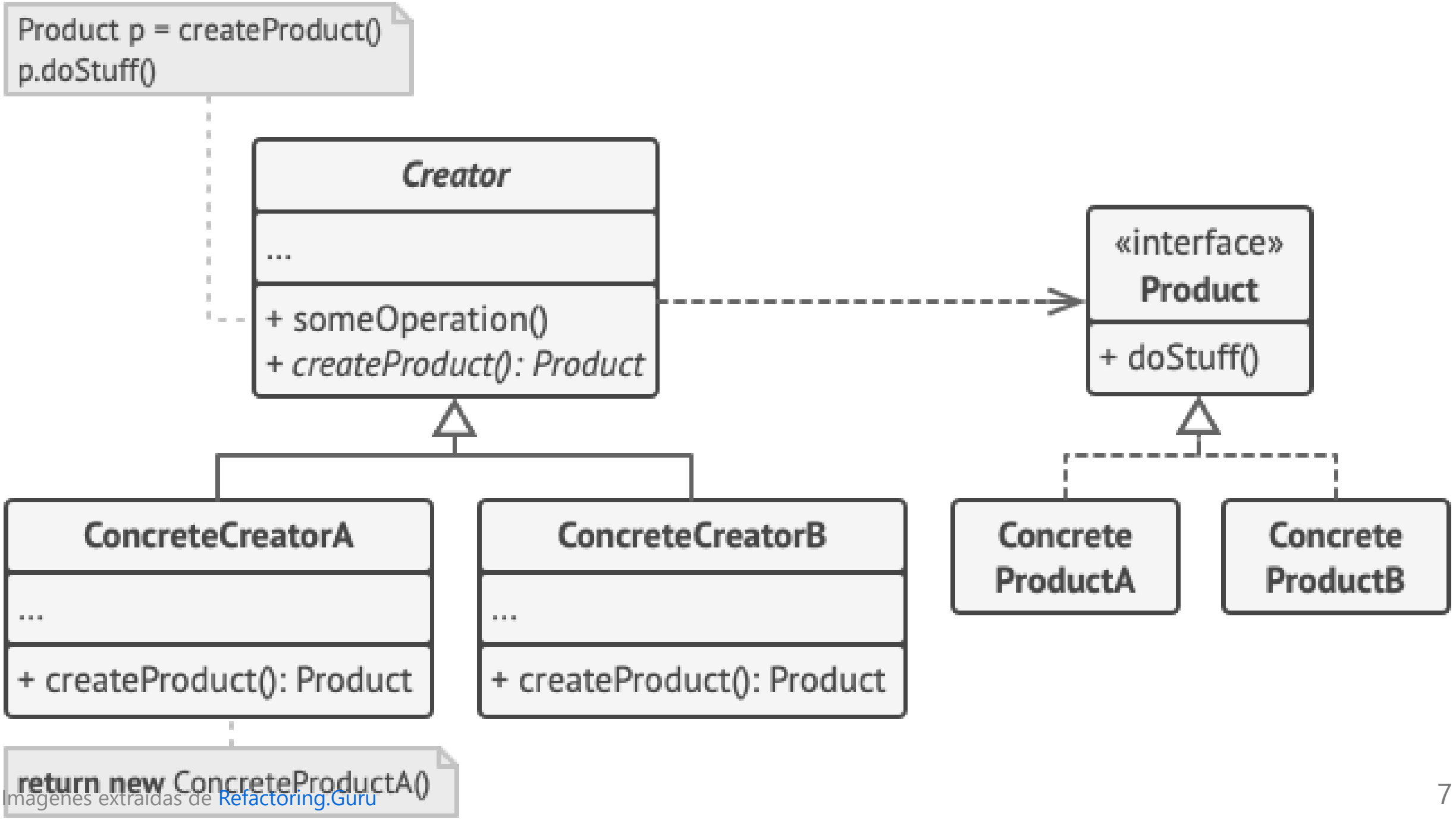
Imágenes extraídas de Refactoring.Guru



Estructura

1. **Producto**: Interfaz común (`Transporte`).
2. **Productos Concretos**: Implementaciones (`Camion` , `Barco`).
3. **Creador**: Declara el método fábrica (`Logistica`).
4. **Creadores Concretos**: Sobrescriben el método fábrica (`LogisticaTerrestre` , `LogisticaMaritima`).

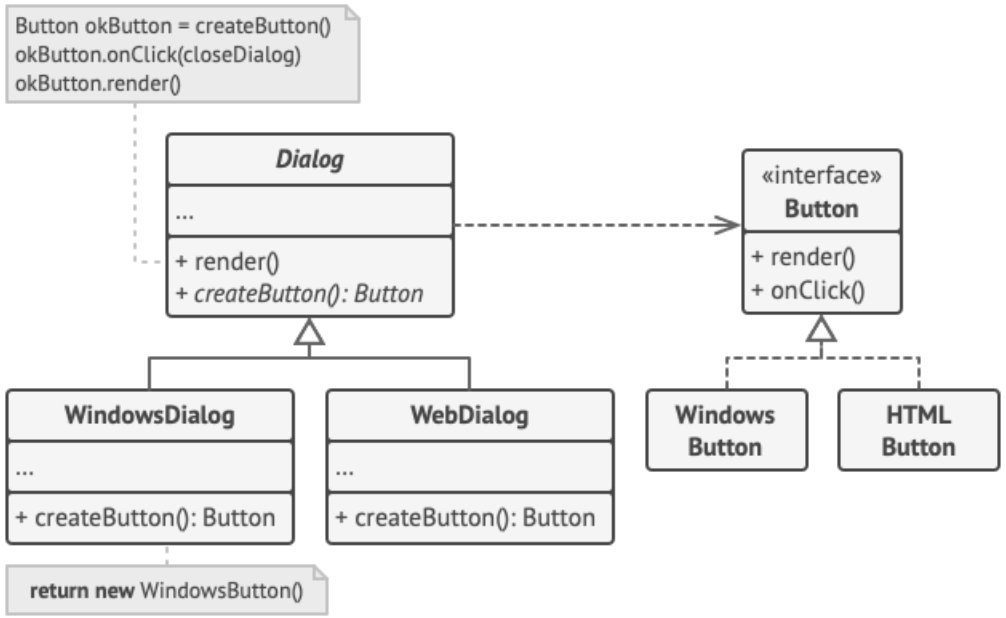
Imágenes extraídas de Refactoring.Guru



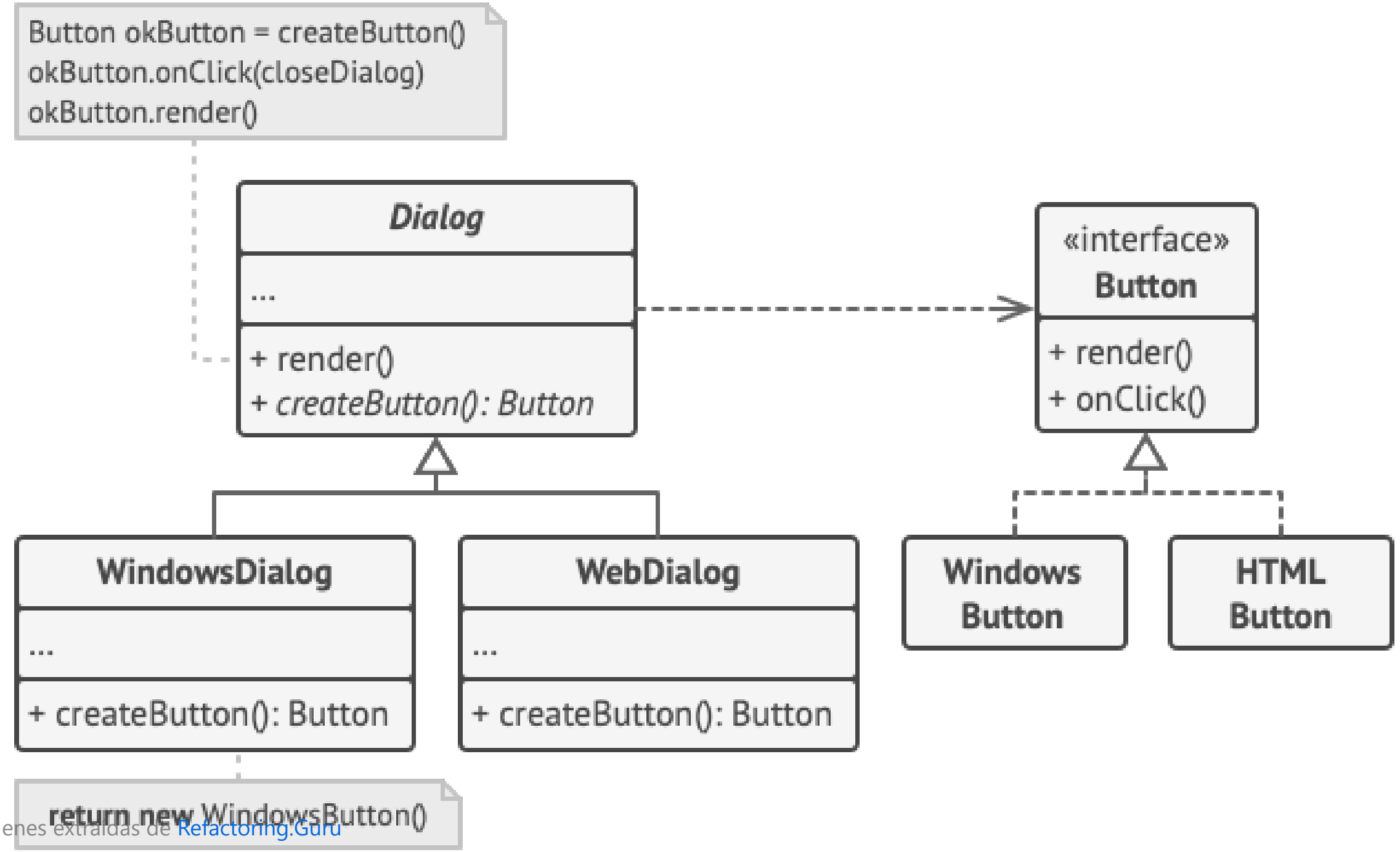
Imágenes extraídas de Refactoring.Guru

Aplicabilidad

- Cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.
- Cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos.
- Cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes.



Imágenes extraídas de Refactoring.Guru



Imágenes extraídas de Refactoring.Guru

Pros y contras

Pros:

- Evita acoplamiento fuerte entre creador y productos concretos.
- **Principio de responsabilidad única**: Código de creación en un solo lugar.
- **Principio de abierto/cerrado**: Nuevos productos sin romper código existente.

Contras:

- El código puede complicarse al introducir muchas subclases nuevas (una para cada producto concreto).

Código Java

El cliente trabaja con la abstracción `Logistica` y `Transporte`.

```
// Creador
public abstract class Logistica {
    public abstract Transporte crearTransporte();

    public void planificarEntrega() {
        Transporte t = crearTransporte();
        t.entregar();
    }
}

// Creador Concreto
public class LogisticaTerrestre extends Logistica {
    @Override
    public Transporte crearTransporte() {
        return new Camion();
    }
}
```

Código Java (Productos)

```java
// Producto (Interfaz)
public interface Transporte {
    void entregar();
}

// Producto Concreto
public class Camion implements Transporte {
    @Override
    public void entregar() {
        System.out.println("Entrega por tierra en una caja.");
    }
}
```