INTRODUCCIÓN

SOLUCIÓN

CÓDIGO

SOLUCIÓN

EJEMPLO

Factory Method

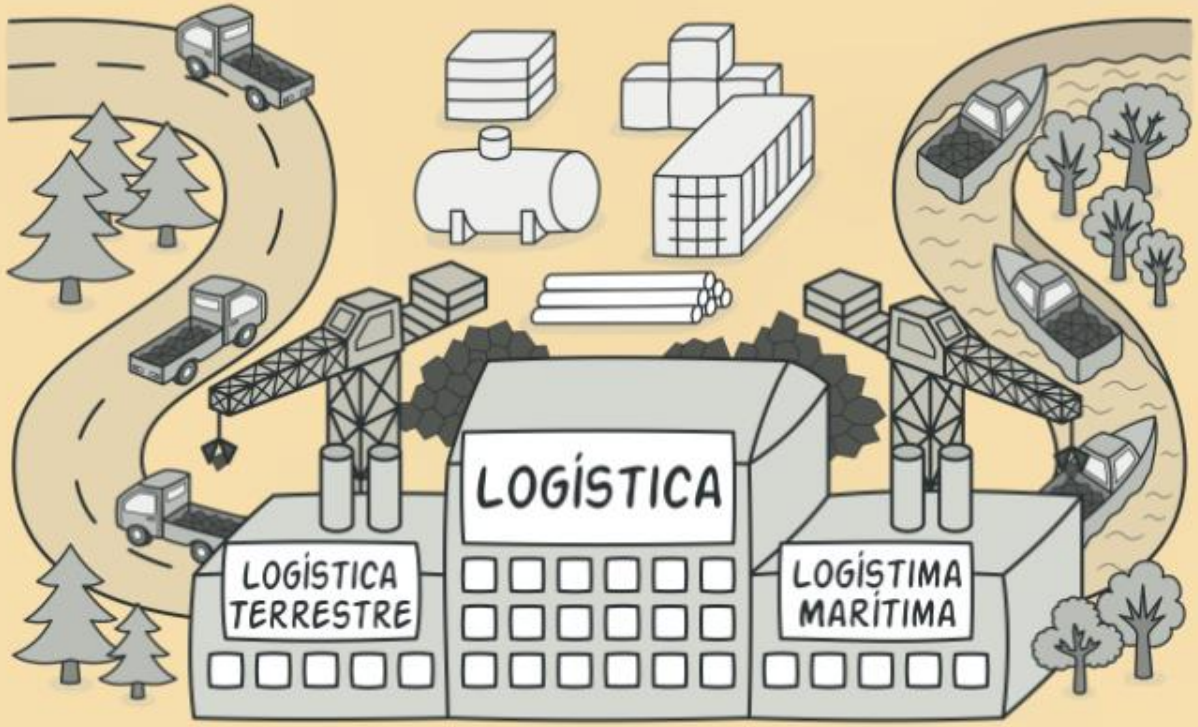
PATRONES DE DISEÑO

PROBLEMA

PROS Y CONTRAS

Introducción

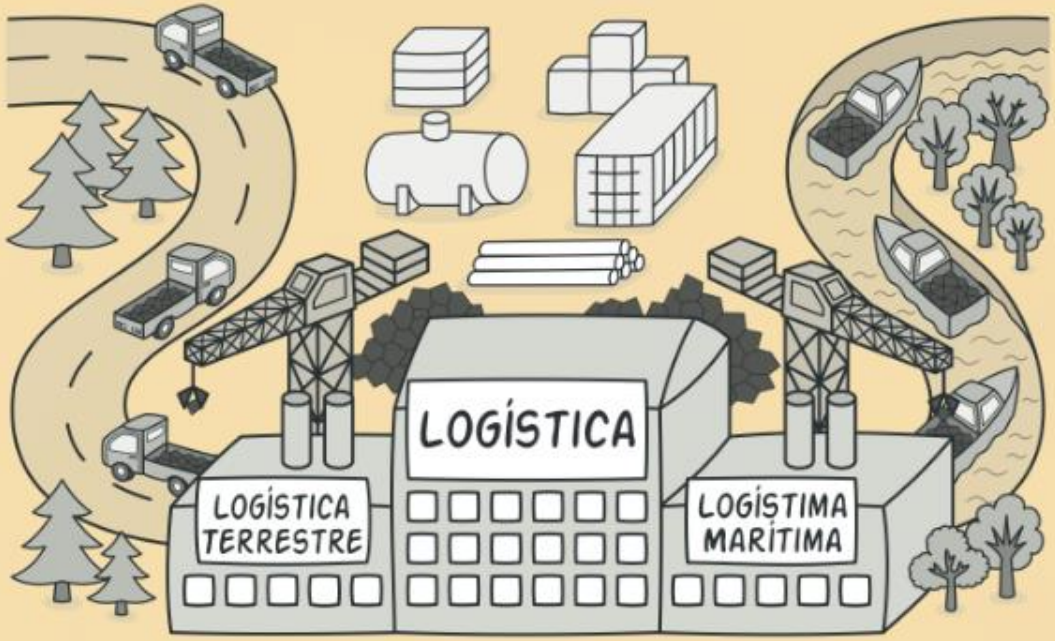
Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



Problema

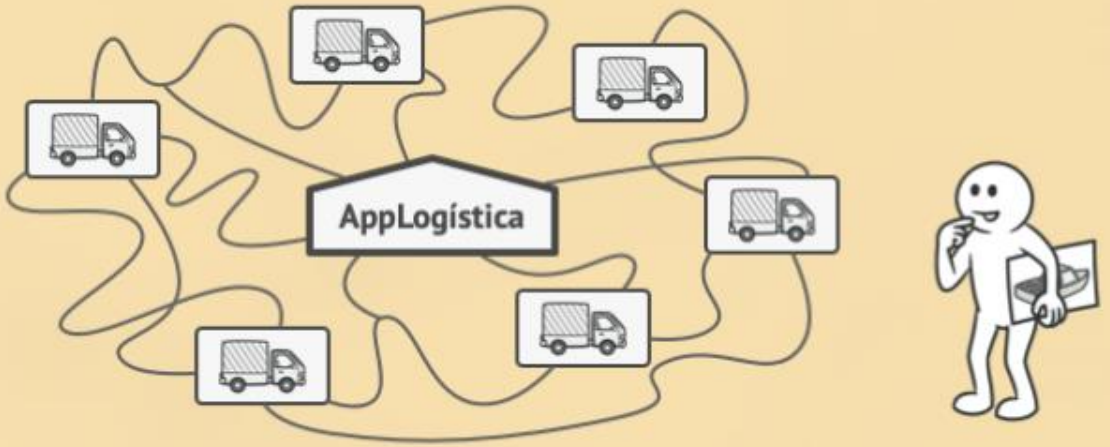
Imaginemos que estamos creando una aplicación de gestión logística. La primera versión de la aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte del código se encuentra dentro de la clase Camión.

Al cabo de un tiempo, la aplicación se vuelve bastante popular. Empezamos a recibir peticiones para que incorporemos la logística por mar a la aplicación.



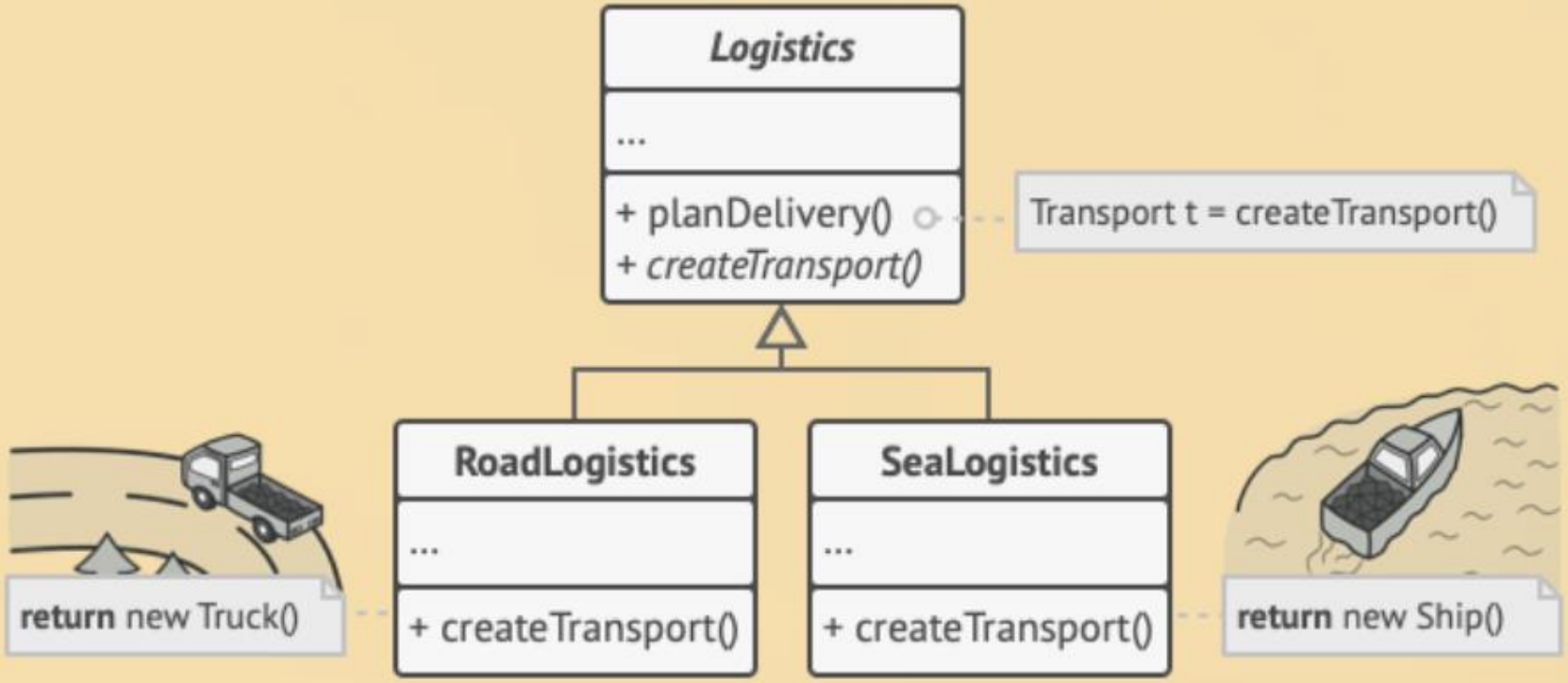
Lo cual es excelente noticia. Pero, ¿qué pasa con el código? En este momento, la mayor parte del código está acoplado a la clase Camión. Para añadir barcos a la aplicación habría que hacer cambios en toda la base del código. Además, si más tarde decidimos añadir otro tipo de transporte a la aplicación, probablemente tendremos que volver a hacer todos estos cambios.

Al final acabaremos con un código bastante sucio, plagado de condicionales que cambian el comportamiento de la aplicación dependiendo de la clase de los objetos de transporte.



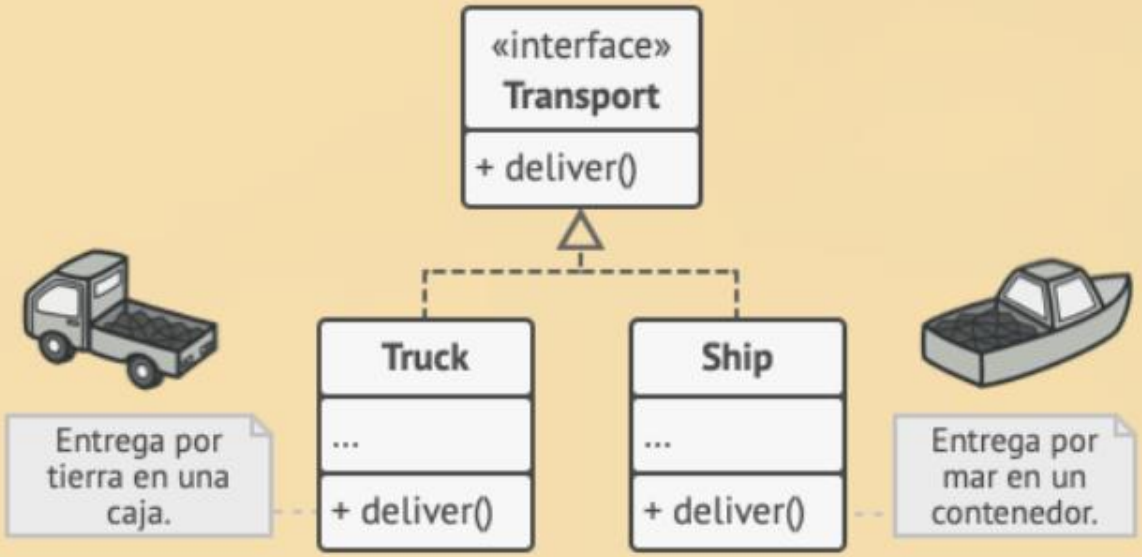
Solución

El patrón Factory Method sugiere que, en lugar de construir objetos directamente, se invoque a un método fábrica especial. Es decir, los objetos se siguen creando, pero se invocan desde el método fábrica. Los objetos devueltos por el método fábrica a menudo se denominan productos.

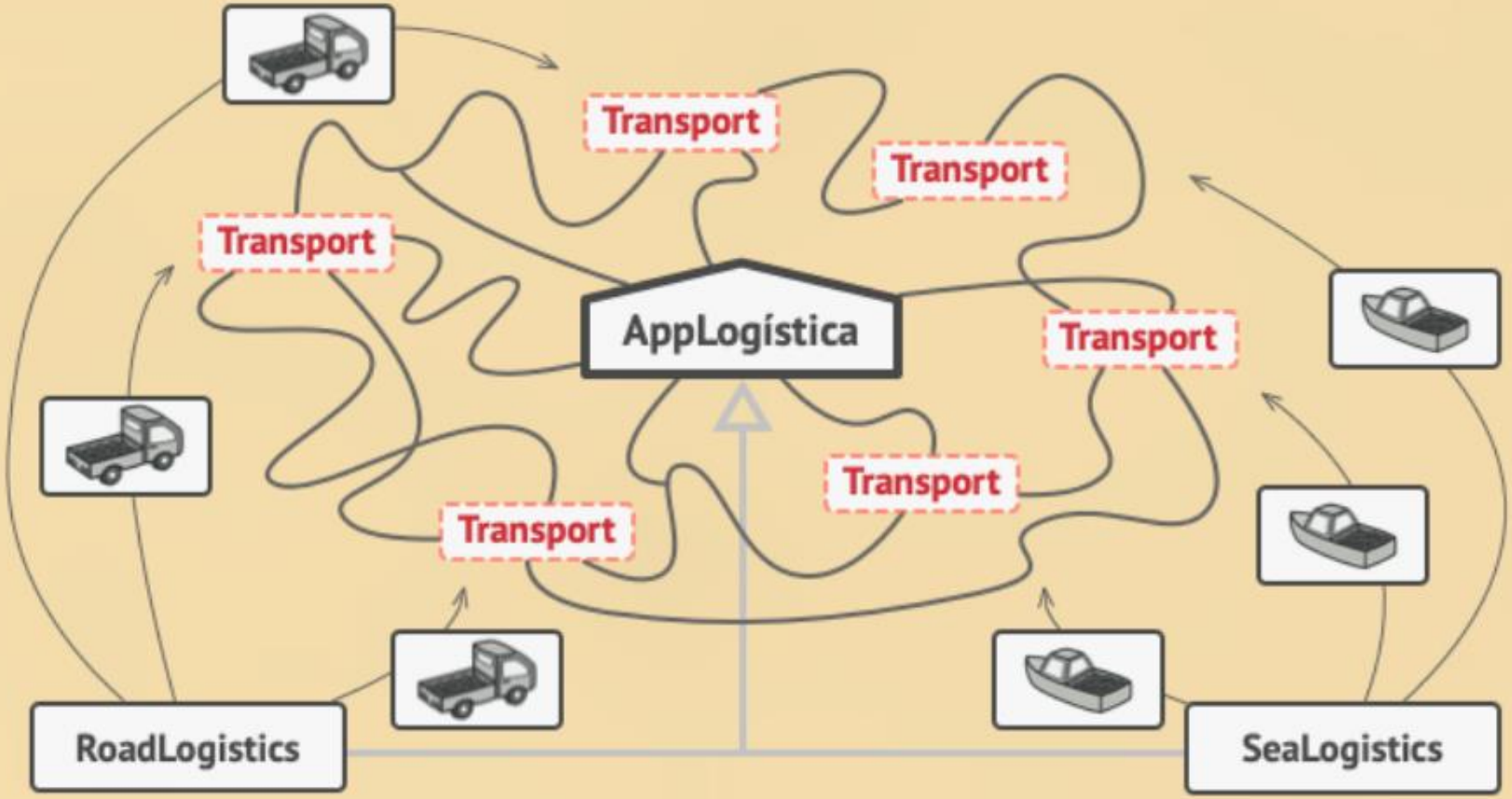


A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, ahora podemos sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

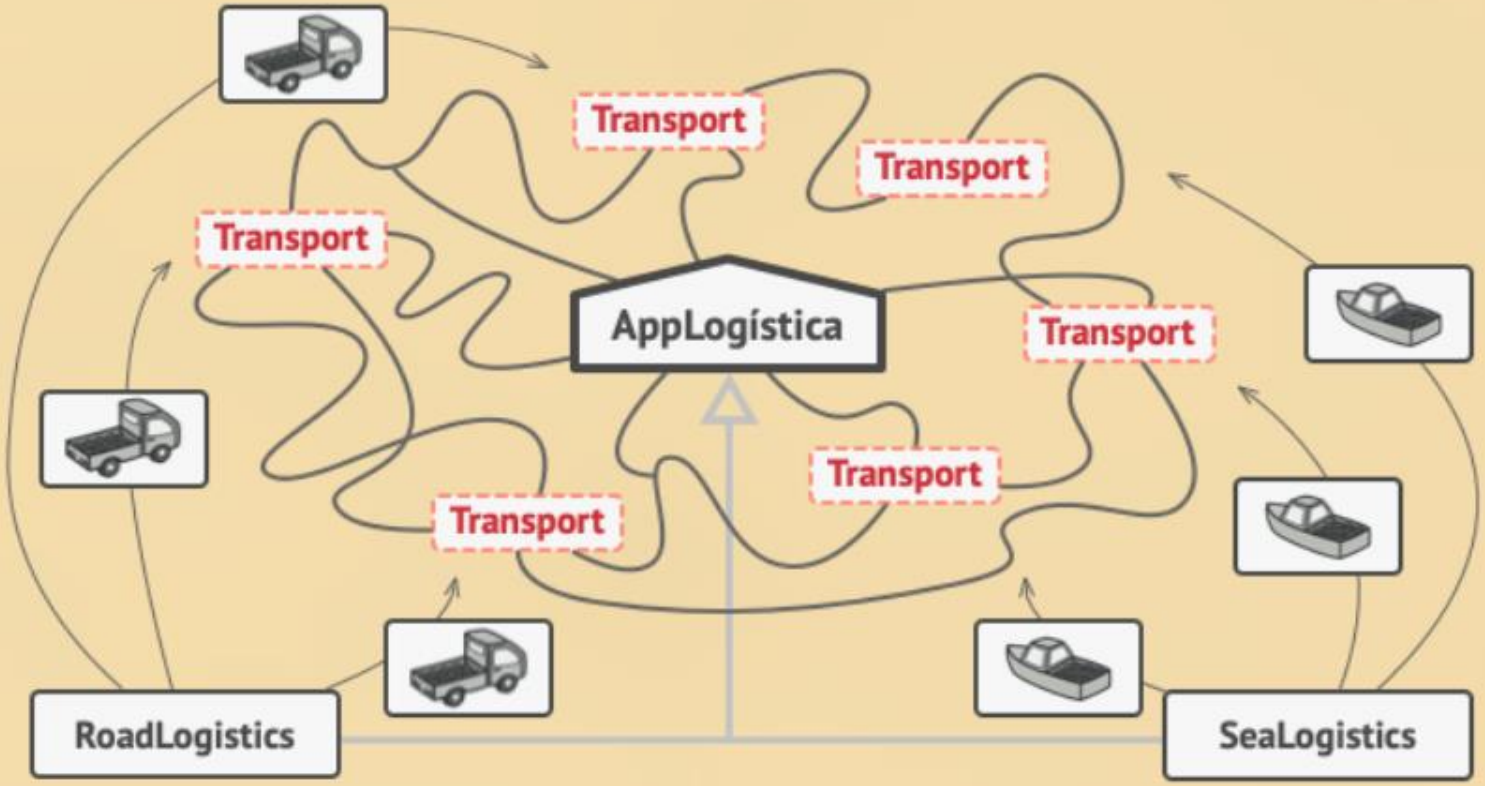
No obstante, hay una pequeña limitación: las subclases sólo pueden devolver productos de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.



Por ejemplo, tanto la clase ***Camión*** como la clase ***Barco*** deben implementar la interfaz ***Transporte***, que declara un método llamado ***entrega***. Cada clase implementa este método de forma diferente: los camiones entregan su carga por tierra, mientras que los barcos lo hacen por mar. El método fábrica dentro de la clase ***LogísticaTerrestre*** devuelve objetos de tipo camión, mientras que el método fábrica de la clase ***LogísticaMarítima*** devuelve barcos.



El código que utiliza el método fábrica (a menudo denominado código cliente) no encuentra diferencias entre los productos devueltos por varias subclases, y trata a todos los productos como la clase abstracta Transporte. El cliente sabe que todos los objetos de transporte deben tener el método entrega, pero no necesita saber cómo funciona exactamente.



PROS

Evitamos un acoplamiento fuerte entre el creador y los productos concretos.

Principio de responsabilidad única: Podemos mover el código de creación de producto a un lugar del programa, haciendo que el código sea más fácil de mantener.

Principio de abierto/cerrado: Podemos incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.

CONTRA

Puede ser que el código se complique, ya que debemos incorporar una multitud de nuevas subclases para implementar el patrón.

```python
from abc import ABC, abstractmethod

# Paso 1: Interfaz de producto (Transporte)
class Transporte(ABC):
    @abstractmethod
    def entrega(self):
        pass

# Paso 2: Clases de producto concretas (Camion y Barco)
class Camion(Transporte):
    def entrega(self):
        return "Entrega por tierra"

class Barco(Transporte):
    def entrega(self):
        return "Entrega por mar"

# Paso 3: Clase creadora abstracta (Logistica)
class Logistica(ABC):
    @abstractmethod
    def crear_transporte(self):
        pass

# Paso 4: Clases creadoras concretas (Terrestre y Maritima)
class LogisticaTerrestre(Logistica):
    def crear_transporte(self):
        return Camion()

class LogisticaMaritima(Logistica):
    def crear_transporte(self):
        return Barco()

# Cliente
def main():
    logistica = LogisticaTerrestre()  # O LogisticaMaritima()
    transporte = logistica.crear_transporte()
    print(transporte.entrega())

if __name__ == "__main__":
    main()
```

Factory Method

¡GRACIAS!

Alumna: María Belén Bertola