

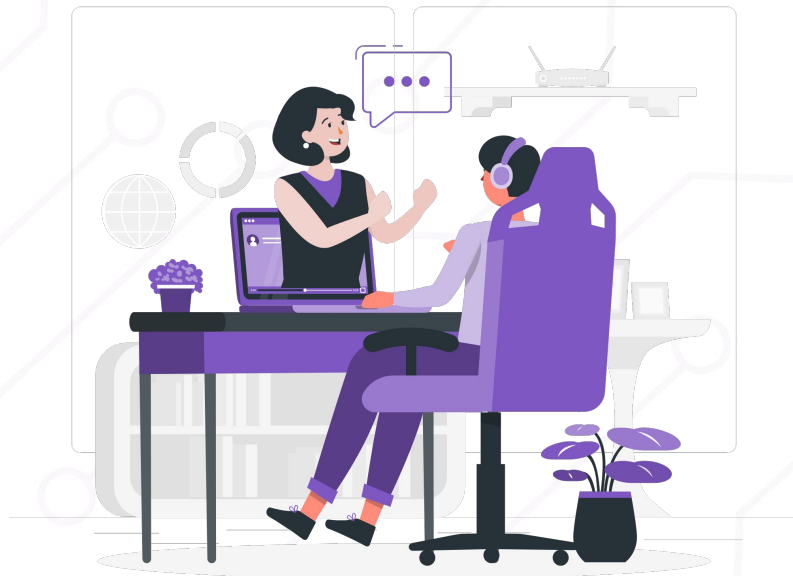
# PATRONES DE DISEÑO

**DEV.F**  
DESARROLLAMOS(PERSONAS);

dev

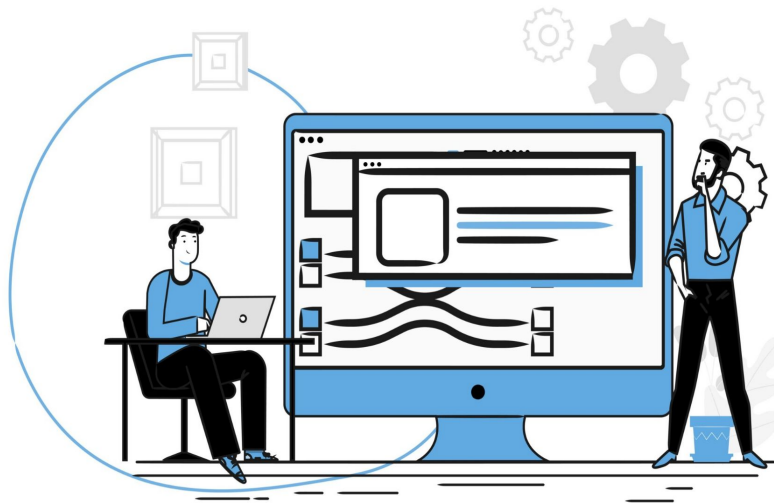
# Objetivo de la sesión

- En la sesión de hoy veremos que son los patrones de diseño.
- Entender bien cuál es la funcionalidad de los patrones de diseño.
- Veremos algunos de los principales patrones de diseño.



# ¿Qué son los patrones de diseño?

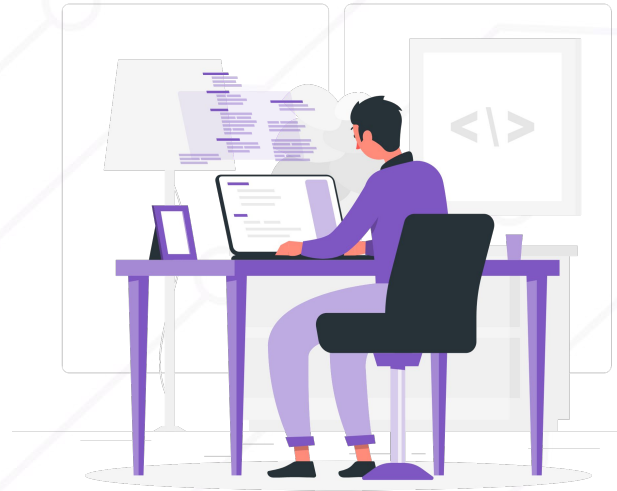
Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el **diseño de software**. Son como planos prefabricados que se pueden personalizar **para resolver un problema de diseño recurrente en tu código**.



# TOMA NOTA

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular.

Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

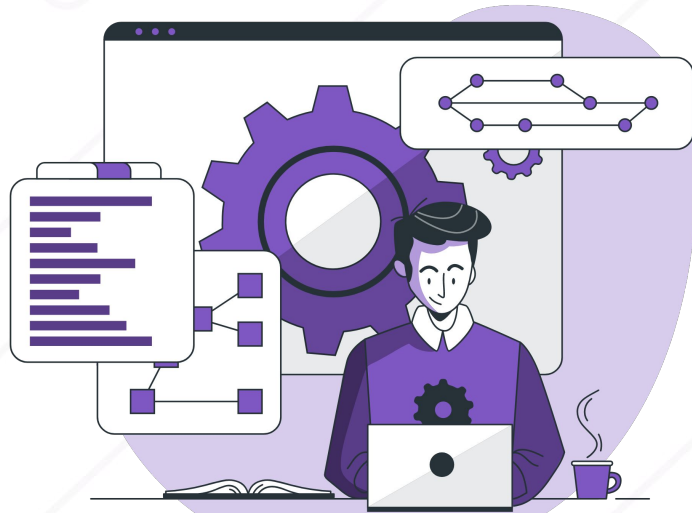


# PATRONES DE DISEÑO

A menudo los patrones se confunden con algoritmos porque ambos **conceptos describen soluciones típicas a problemas conocidos.**

Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución. **El código del mismo patrón aplicado a dos programas distintos puede ser diferente.**





Una analogía de un algoritmo sería una receta de cocina: ambos cuentan con pasos claros para alcanzar una meta. Por su parte, un patrón **es más similar a un plano**, ya que puedes observar cómo son su resultado y sus funciones, **pero el orden exacto de la implementación depende de ti.**

# Clasificación de los patrones de diseño

## CREACIONALES

Soluciona la creación de objetos, hace un sistema independiente de cómo sus objetos son creados.



## ESTRUCTURALES

Describe cómo los objetos se componen para formar estructuras complejas.



## COMPORTAMIENTO

Establece soluciones relacionados con el comportamiento de la aplicación respecto a la interacción entre objetos y clases.



# Prototype Pattern

**DEV.F**  
DESARROLLAMOS(PERSONAS);

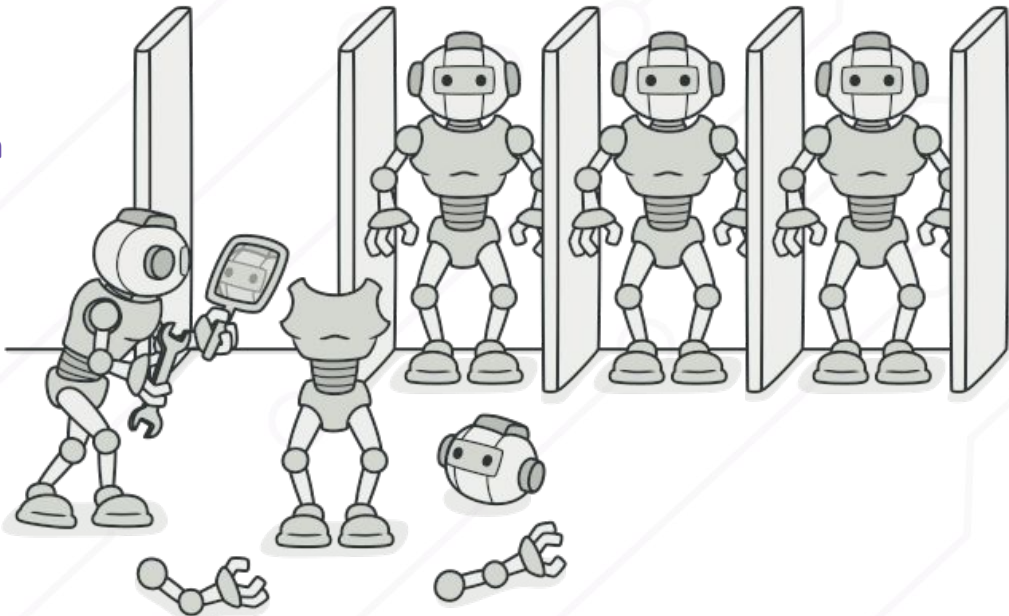
dev



# PROTOTYPE

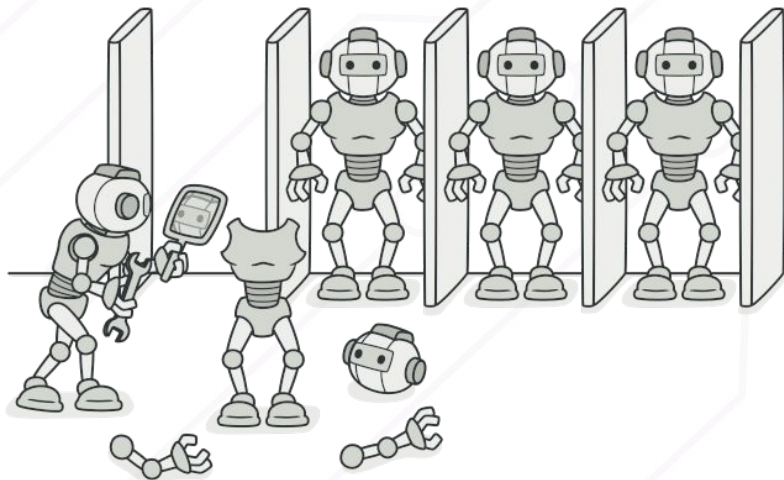
## Propósito

Prototype es un patrón de diseño creacional que nos permite **copiar objetos existentes sin que el código dependa de sus clases.**



## ☹ Problema

Digamos que tienes un objeto y quieres crear una copia exacta de él. ¿Cómo lo harías?





## ☹ Problema

En primer lugar, **debes crear un nuevo objeto de la misma clase.** Después debes recorrer todos los campos del objeto original y copiar sus valores en el nuevo objeto.

¡Bien! Pero hay una trampa. **No todos los objetos se pueden copiar de este modo,** porque algunos de los campos del objeto pueden **ser privados e invisibles** desde fuera del propio objeto.



## 😊 Solución

El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite **clonar un objeto sin acoplar el código a la clase de ese objeto**. Normalmente, dicha interfaz contiene un único método `clonar`.



# Observer Pattern

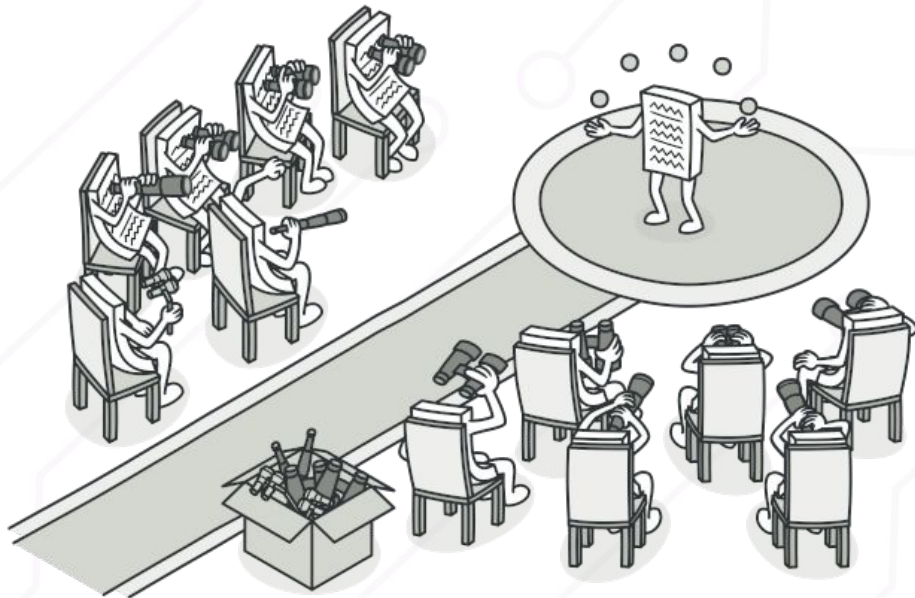
**DEV.F**  
DESARROLLAMOS(PERSONAS);

dev

# OBSERVER

## Propósito

Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

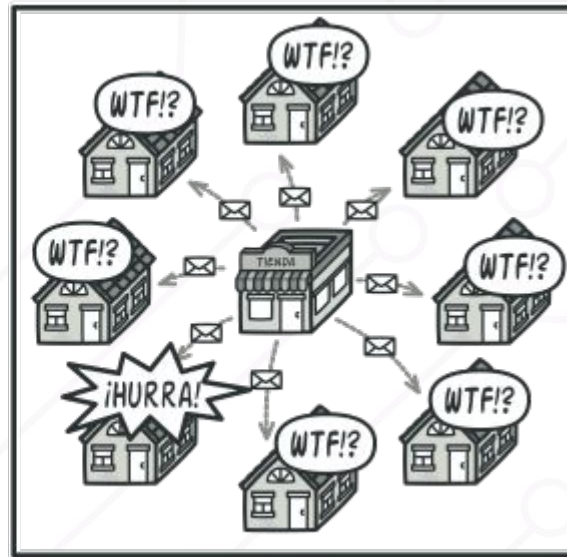


## 🙄 Problema

Imagina que tienes dos tipos de objetos: **un objeto Cliente** y **un objeto Tienda**. El cliente está muy interesado en una marca particular de producto (digamos, un nuevo modelo de iPhone) que estará disponible en la tienda muy pronto.

El cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.

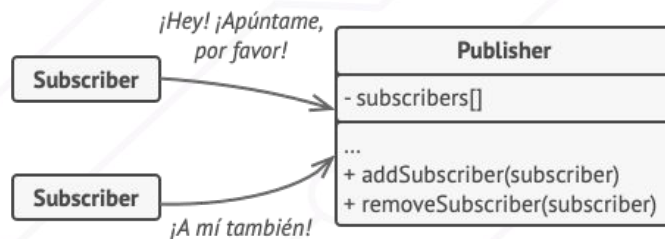




Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados en los nuevos productos.

## 😊 Solución

El objeto que tiene un estado interesante suele denominarse *sujeto*, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos *notificador* (en ocasiones también llamado *publicador*). El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan *suscriptores*.



# Factory Pattern

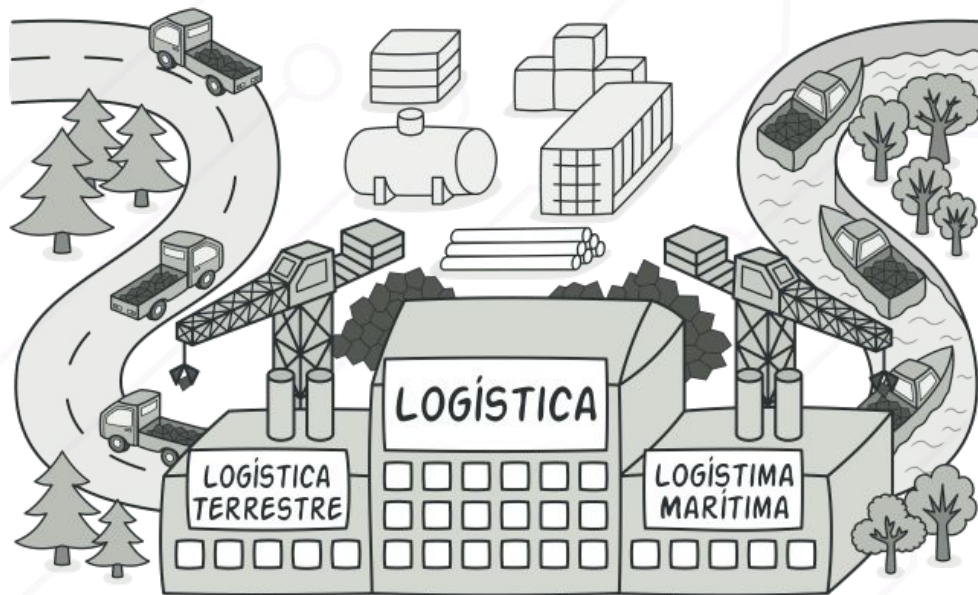
**DEV.F**  
DESARROLLAMOS(PERSONAS);

dev

# FACTORY

## 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.





## 😞 Problema

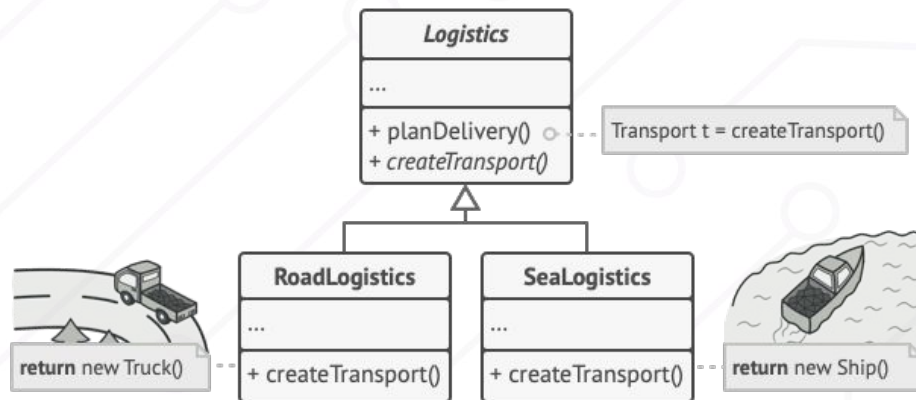
Imagina que estás creando una aplicación de gestión logística. La primera versión de tu aplicación sólo es capaz de manejar el transporte en camión, por lo que **la mayor parte de tu código se encuentra dentro de la clase `Camión`**.

Al cabo de un tiempo, tu aplicación se vuelve bastante popular. Cada día recibes decenas de peticiones de empresas de transporte marítimo para que incorpores la logística por mar a la aplicación.



## 😊 Solución

El patrón Factory Method sugiere que, en lugar de llamar al operador `new` para construir objetos directamente, se invoque a un método **fábrica especial**. No te preocupes: los objetos se siguen creando a través del operador `new`, pero **se invocan desde el método fábrica**. Los objetos devueltos por el método fábrica a menudo se denominan *productos*.



# Singleton Pattern

**DEV.F**  
DESARROLLAMOS(PERSONAS);

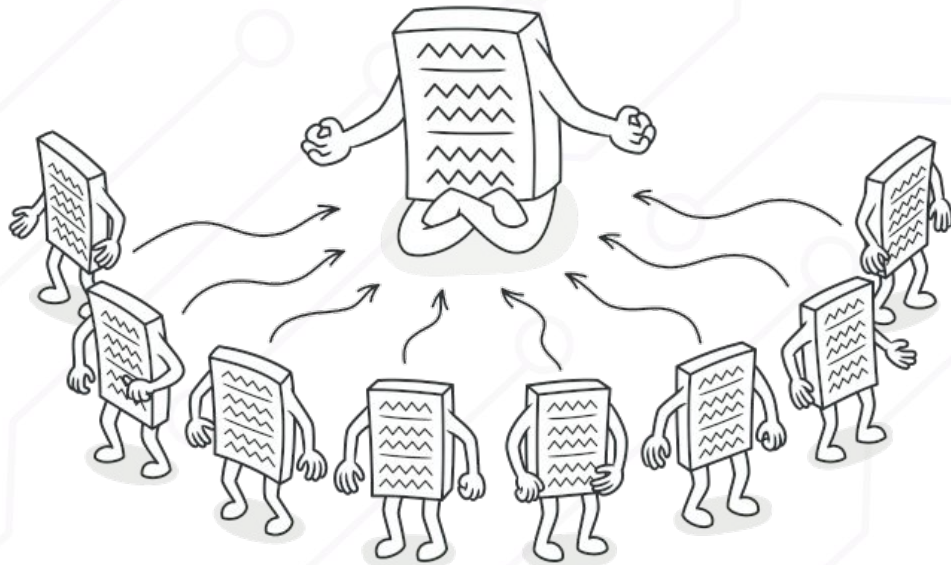
dev



# SINGLETON

## Propósito

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que **una clase tenga una única instancia**, a la vez que proporciona un punto de acceso global a dicha instancia.



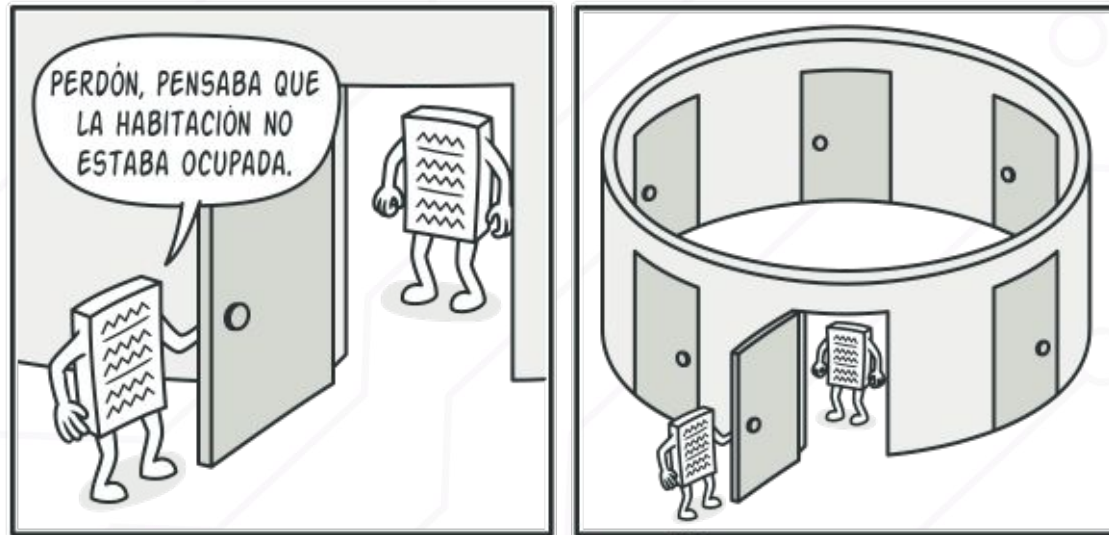


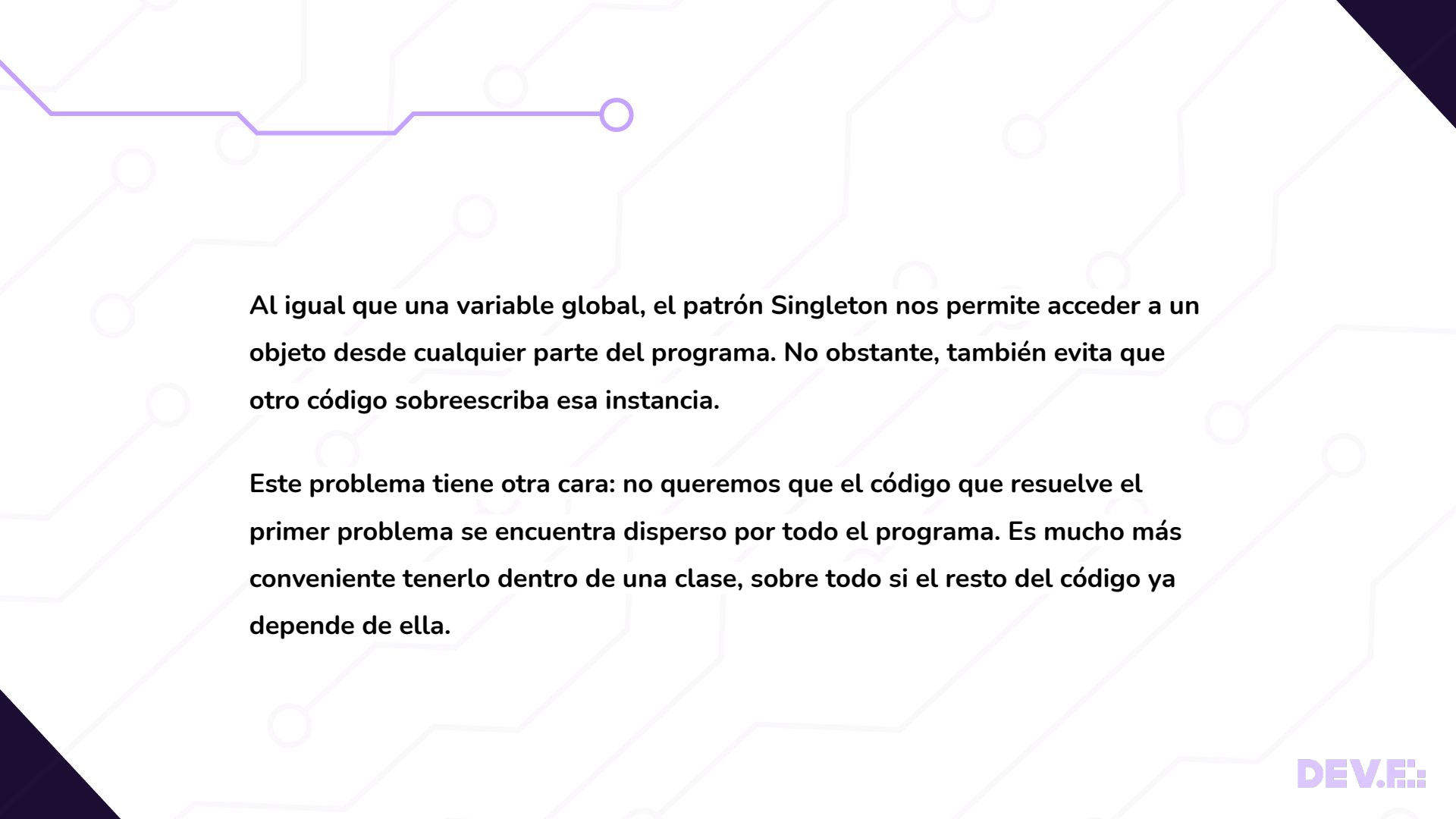
## 🙄 Problema

El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.

Funciona así: imagina que has creado un objeto y al cabo de un tiempo decides crear otro nuevo. **En lugar de recibir un objeto nuevo, obtendrás el que ya habías creado.**

Ten en cuenta que este comportamiento es imposible de implementar con un constructor normal, **ya que una llamada al constructor siempre debe devolver un nuevo objeto por diseño.**





Al igual que una variable global, el patrón Singleton nos permite acceder a un objeto desde cualquier parte del programa. No obstante, también evita que otro código sobrescriba esa instancia.

Este problema tiene otra cara: no queremos que el código que resuelve el primer problema se encuentra disperso por todo el programa. Es mucho más conveniente tenerlo dentro de una clase, sobre todo si el resto del código ya depende de ella.