

Ingeniería de Software

17 -Patrones de Diseño III: Creacionales

IIC2143-3

Josefa España

jpespana@uc.cl

Clase “patrocinada” por Refactoring Guru



Patrones de Diseño

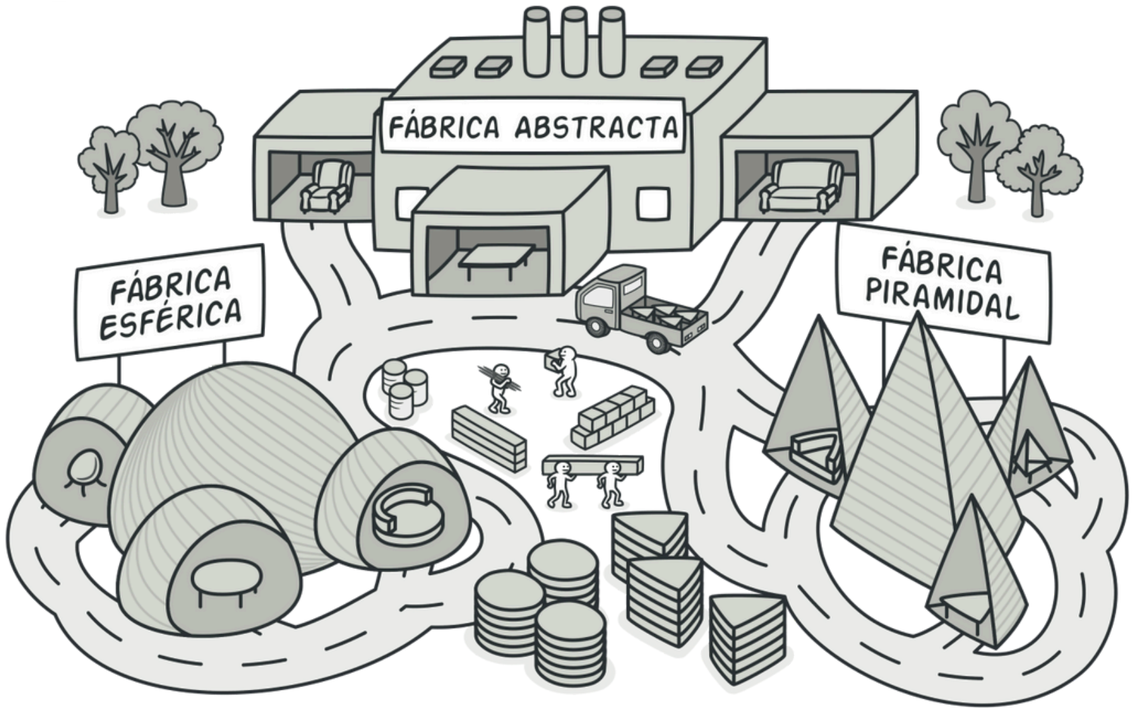
Patrones Creacionales

- Abstract factory
- Builder
- Factory
- Singleton

Abstract Factory

Propósito

Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



Abstract Factory

Motivación - Problema

- Simulador de tienda de muebles:
  - Familia de productos relacionados: Silla + Sofá + Mesita.
  - Estilos de productos: Moderna, Victoriana, ArtDeco.



Familias de productos y sus variantes.

Abstract Factory

Motivación - Problema

- ¿Cómo crear productos individuales que combinen entre ellos?
- **Requisito**: No cambiar el código existente al añadir nuevos productos o familias.

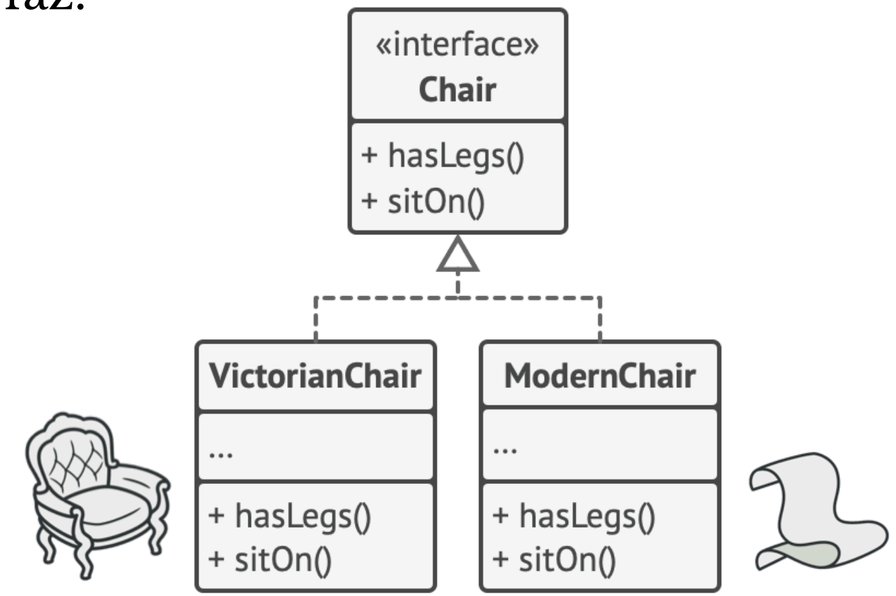


Un sofá de estilo moderno no combina con unas sillas de estilo victoriano.

Abstract Factory

Motivación - Solución

- Declarar una interfaz para cada producto diferente de la familia de productos.
- Todas las variantes de producto seguirán esta interfaz.

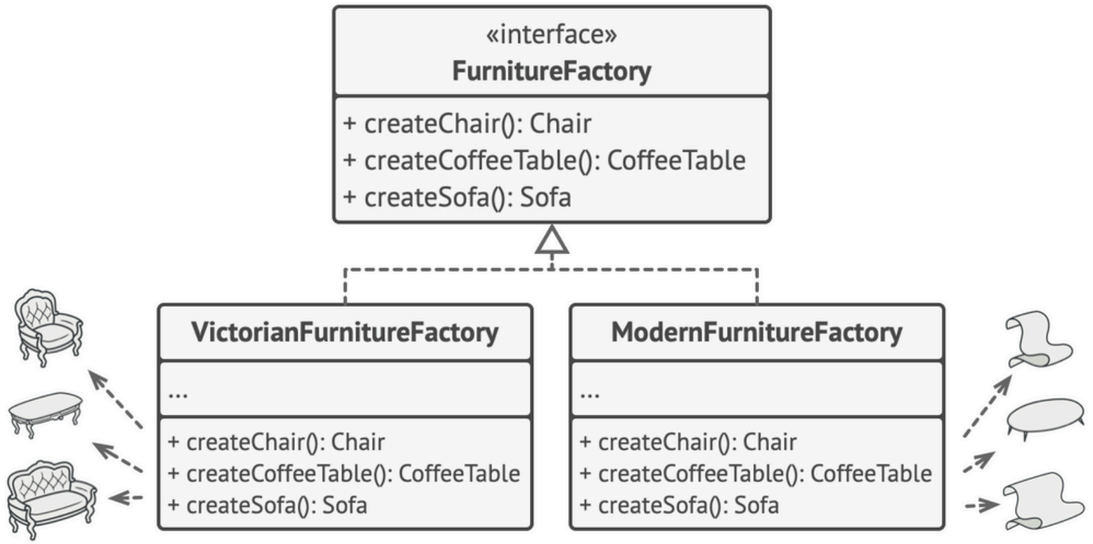


Todas las variantes del mismo objeto deben moverse a una única jerarquía de clase.

Abstract Factory

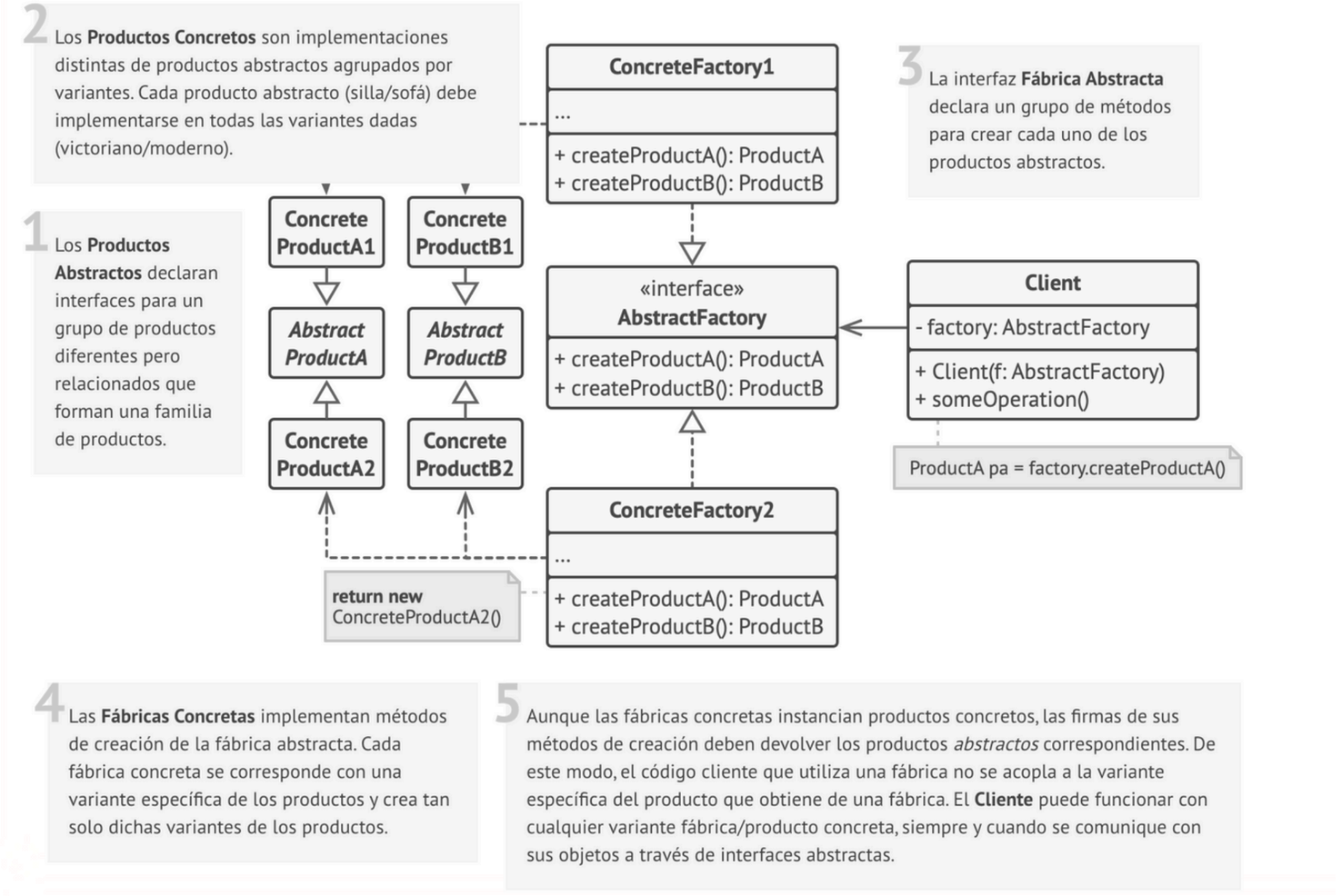
Motivación - Solución

- Luego declaramos la fábrica abstracta: Una interfaz con una lista de métodos de creación para todos los productos parte de la familia de productos.
- Los métodos devuelven productos abtractos.



Cada fábrica concreta se corresponde con una variante específica del producto.

Abstract Factory



Abstract Factory

Pros y contras

- ✓ Puedes tener la certeza de que los productos que obtienes de una fábrica son compatibles entre sí.
- ✓ Evitas un acoplamiento fuerte entre productos concretos y el código cliente.
- ✓ *Principio de responsabilidad única*. Puedes mover el código de creación de productos a un solo lugar, haciendo que el código sea más fácil de mantener.
- ✓ *Principio de abierto/cerrado*. Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente.
- ✗ Puede ser que el código se complique más de lo que debería, ya que se introducen muchas nuevas interfaces y clases junto al patrón.

Abstract Factory

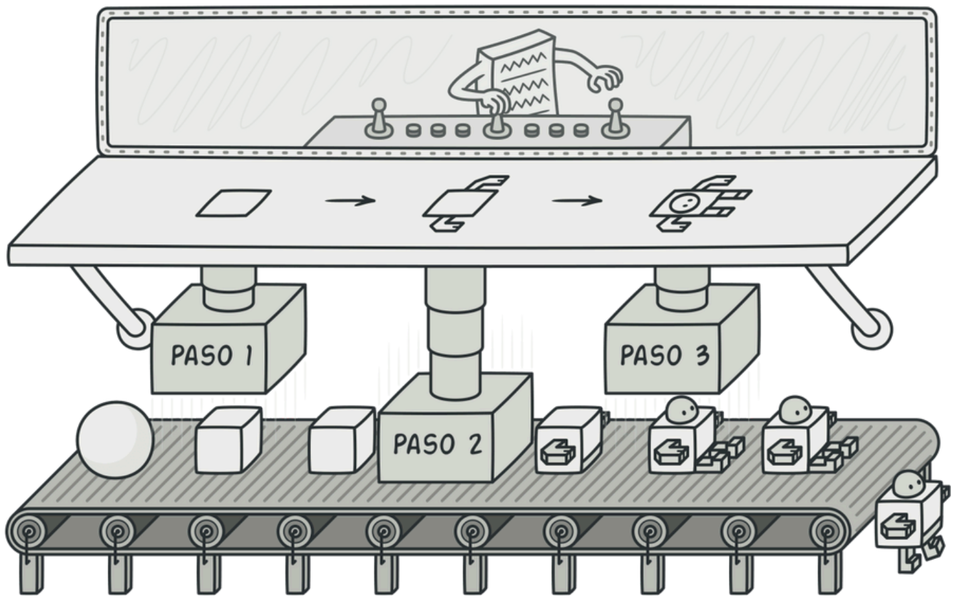
Ejemplo

- Ejemplo en el zip de esta clase.

Builder

Propósito

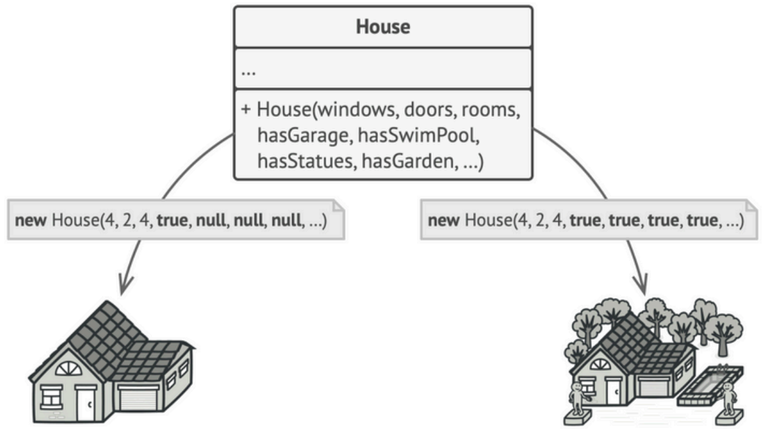
Builder es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.



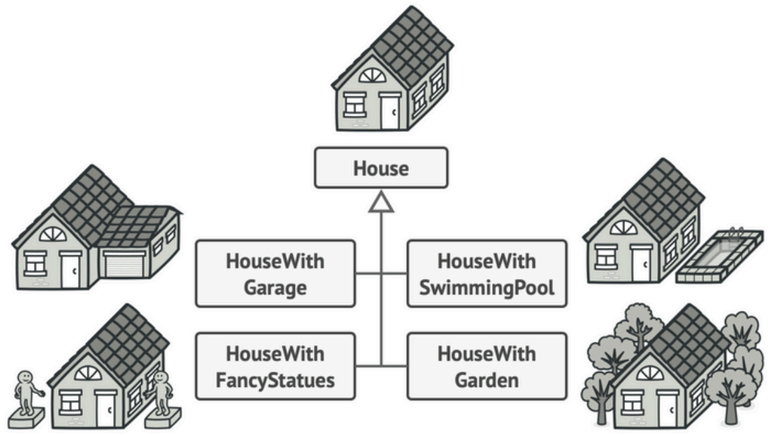
Builder

Motivación - Problema

- Objeto complejo que requiere una inicialización con muchos parámetros y objetos anidados.
- ¿Qué pasa si queremos diferentes tipos de casas?
  - Nuestro código tendrá muchas subclases o parámetros.



Un constructor con un montón de parámetros tiene su inconveniente: no todos los parámetros son necesarios todo el tiempo.

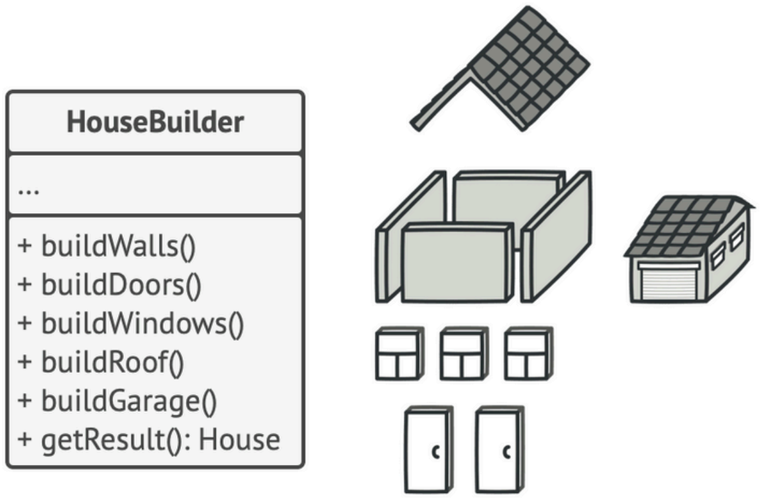


Crear una subclase por cada configuración posible de un objeto puede complicar demasiado el programa.

Builder

Motivación - Solución

- Sacar el código de construcción del objeto de su propia clase y colocarlo dentro de objetos independientes, llamados constructores.



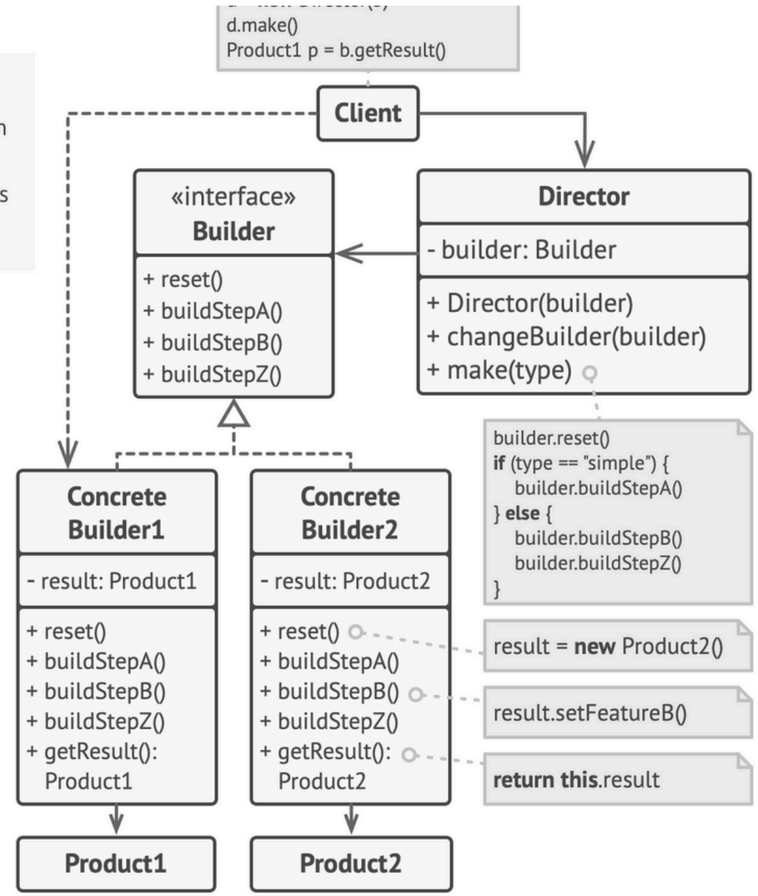
El patrón Builder te permite construir objetos complejos paso a paso. El patrón Builder no permite a otros objetos acceder al producto mientras se construye.

Builder

1 La interfaz **Constructora** declara pasos de construcción de producto que todos los tipos de objetos constructores tienen en común.

2 Los **Constructores Concretos** ofrecen distintas implementaciones de los pasos de construcción. Los constructores concretos pueden crear productos que no siguen la interfaz común.

3 Los **Productos** son los objetos resultantes. Los productos construidos por distintos objetos constructores no tienen que pertenecer a la misma jerarquía de clases o interfaz.



4 La clase **Directora** define orden en el que se invoca los pasos de construcción lo que puedes crear y reutilizar configuraciones específicas de los produc

5 El **Cliente** debe asociar u de los objetos constructo con la clase directora. Normalmente, se hace un sola vez mediante los parámetros del construct la clase directora, que uti el objeto constructor para resto de la construcción. obstante, existe una solu alternativa para cuando e cliente pasa el objeto constructor al método de producción de la clase directora. En este caso, puedes utilizar un constr diferente cada vez que produzcas algo con la cla directora.

Builder

⚖ Pros y contras

- ✓ Puedes construir objetos paso a paso, aplazar pasos de la construcción o ejecutar pasos de forma recursiva.
- ✓ Puedes reutilizar el mismo código de construcción al construir varias representaciones de productos.
- ✓ *Principio de responsabilidad única*. Puedes aislar un código de construcción complejo de la lógica de negocio del producto.
- ✗ La complejidad general del código aumenta, ya que el patrón exige la creación de varias clases nuevas.

Builder

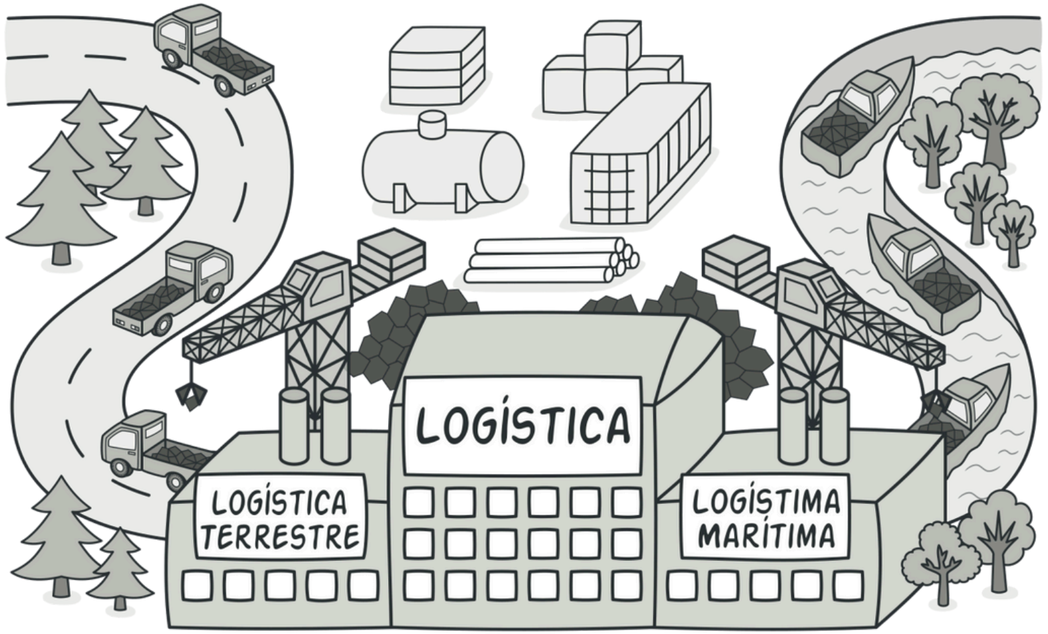
Ejemplo

- Ejemplo en el zip de esta clase.

Factory

Propósito

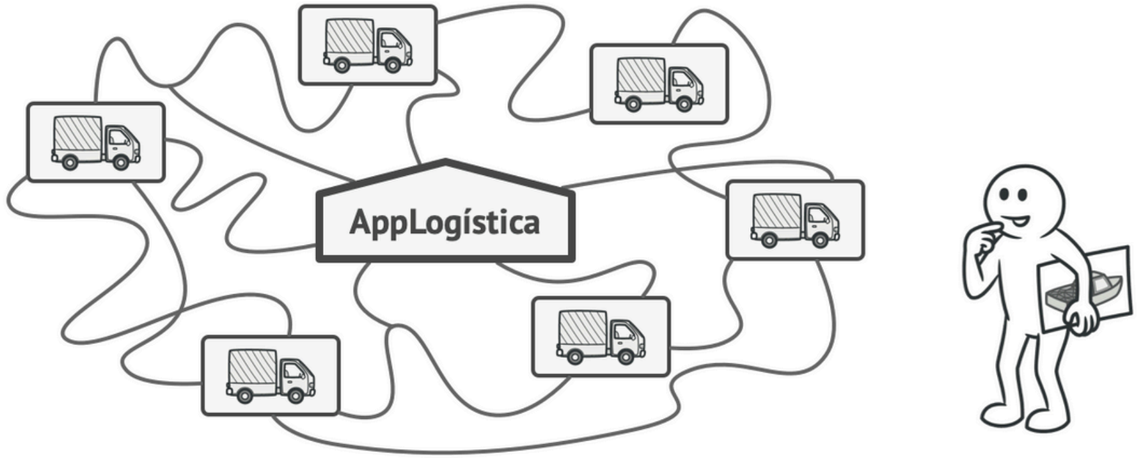
Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



Factory

Motivación - Problema

- Aplicación de gestión lógica: Actualmente solo haces gestión de transporte en camión.
- ¿Cómo agregar un nuevo transporte a tu lógica sin enredar el código?

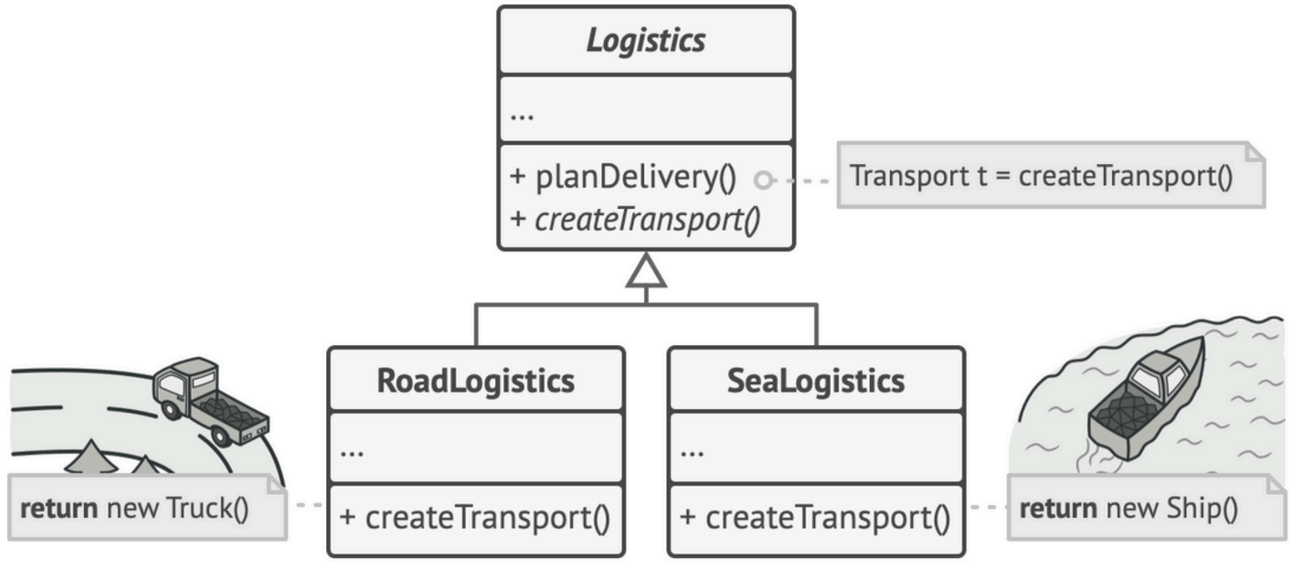


Añadir una nueva clase al programa no es tan sencillo si el resto del código ya está acoplado a clases existentes.

Factory

Motivación - Solución

- En vez de utilizar el operador new para construir objetos, utilizar un método “fabrica”.

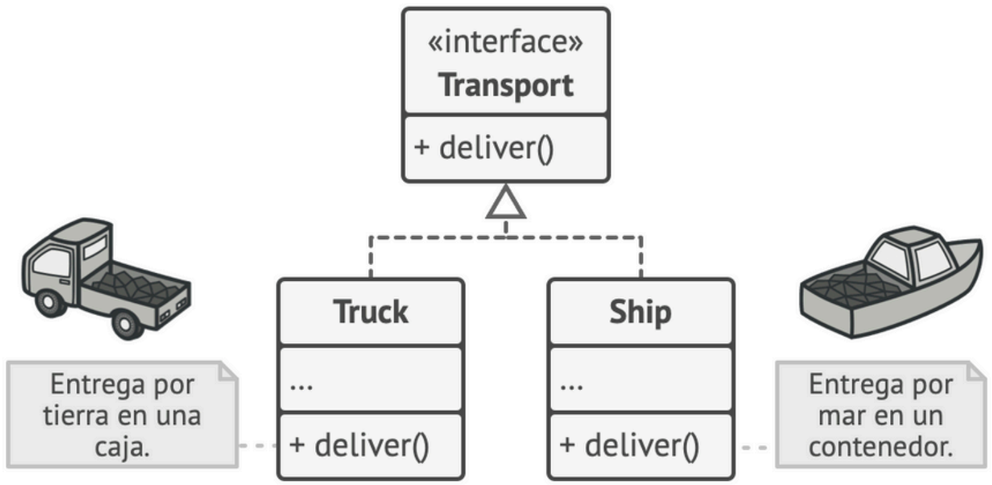


Las subclases pueden alterar la clase de los objetos devueltos por el método fábrica.

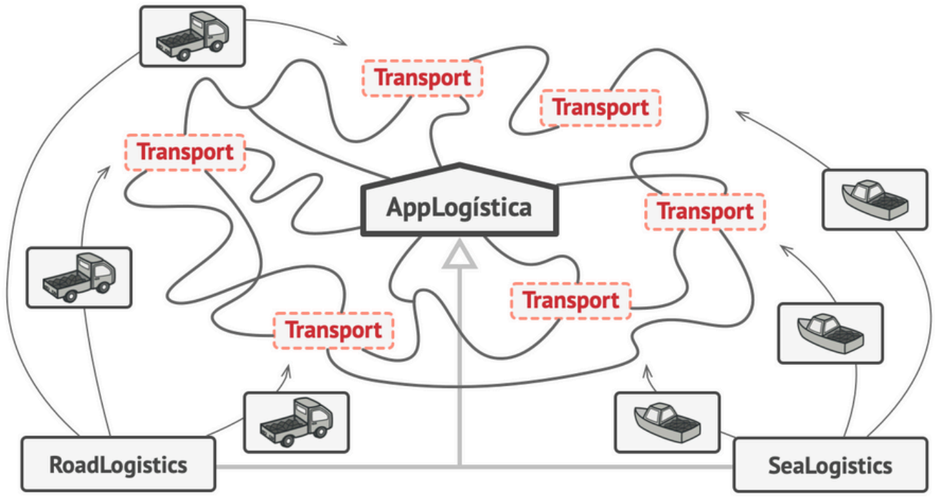
Factory

Motivación - Solución

- Todos los productos deben seguir la misma interfaz, para que el cliente pueda utilizarlos de manera indiferente.



Todos los productos deben seguir la misma interfaz.



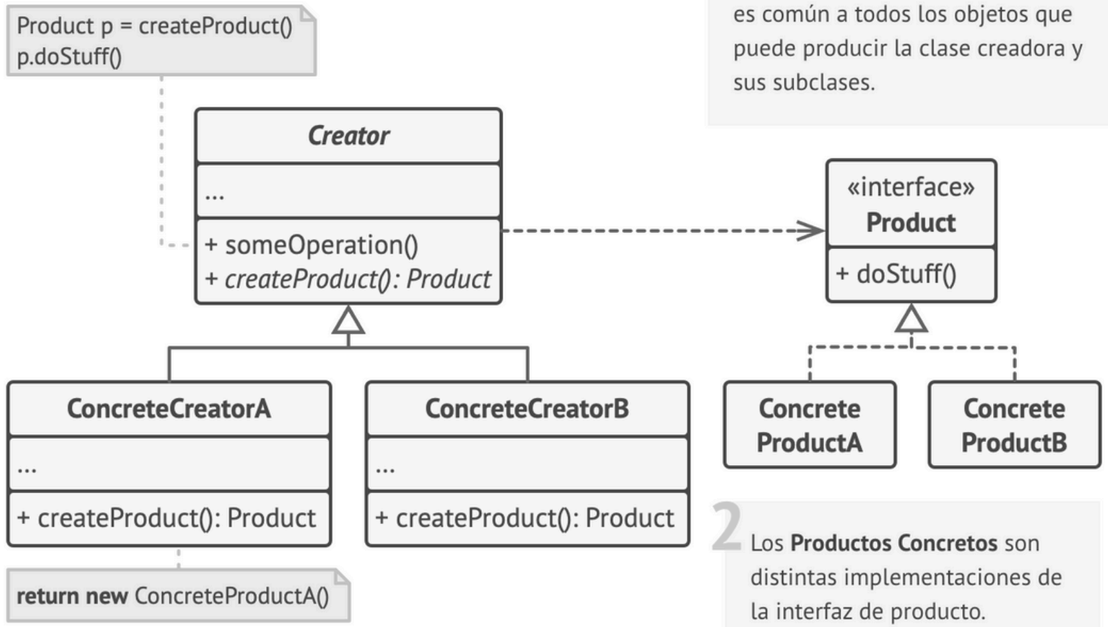
Siempre y cuando todas las clases de producto implementen una interfaz común, podrás pasar sus objetos al código cliente sin descomponerlo.

Factory

3 La clase **Creadora** declara el método fábrica que devuelve nuevos objetos de producto. Es importante que el tipo de retorno de este método coincida con la interfaz de producto.

Puedes declarar el patrón Factory Method como abstracto para forzar a todas las subclases a implementar sus propias versiones del método. Como alternativa, el método fábrica base puede devolver algún tipo de producto por defecto.

Observa que, a pesar de su nombre, la creación de producto **no** es la principal responsabilidad de la clase creadora. Normalmente, esta clase cuenta con alguna lógica de negocios central relacionada con los productos. El patrón Factory Method ayuda a desacoplar esta lógica de las clases concretas de producto. Aquí tienes una analogía: una gran empresa de desarrollo de software puede contar con un departamento de formación de programadores. Sin embargo, la principal función de la empresa sigue siendo escribir código, no preparar programadores.



1 El **Producto** declara la interfaz, que es común a todos los objetos que puede producir la clase creadora y sus subclases.

2 Los **Productos Concretos** son distintas implementaciones de la interfaz de producto.

4 Los **Creadores Concretos** sobrescriben el Factory Method base, de modo que devuelva un tipo diferente de producto.

Observa que el método fábrica no tiene que **crear** nuevas instancias todo el tiempo. También puede devolver objetos existentes de una memoria caché, una agrupación de objetos, u otra fuente.

Factory

Ejemplo

- Ejemplo en el zip de esta clase.

Factory

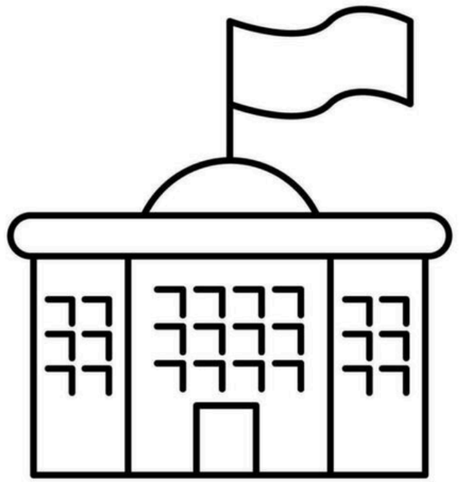
⚖ Pros y contras

- ✓ Evitas un acoplamiento fuerte entre el creador y los productos concretos.
- ✓ *Principio de responsabilidad única*. Puedes 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*. Puedes incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.
- ✗ Puede ser que el código se complique, ya que debes incorporar una multitud de nuevas subclases para implementar el patrón. La situación ideal sería introducir el patrón en una jerarquía existente de clases creadoras.

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.

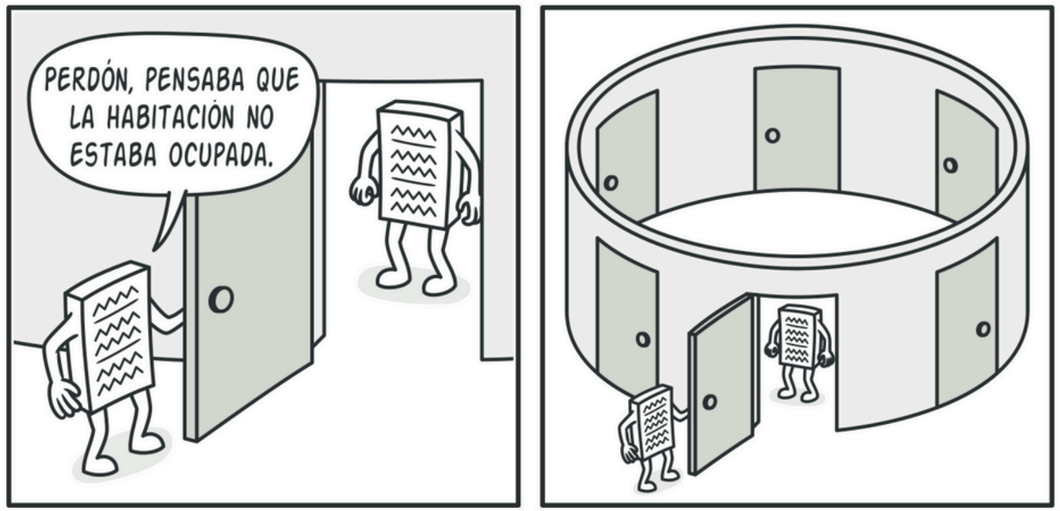


GOVERNMENT

Singleton

Motivación - Problema

- Pensemos que estamos programando un partido de fútbol. Solo queremos que exista la instancia de una pelota.
  - ¿Cómo limitamos que exista solo una instancia?



Puede ser que los clientes ni siquiera se den cuenta de que trabajan con el mismo objeto todo el tiempo.

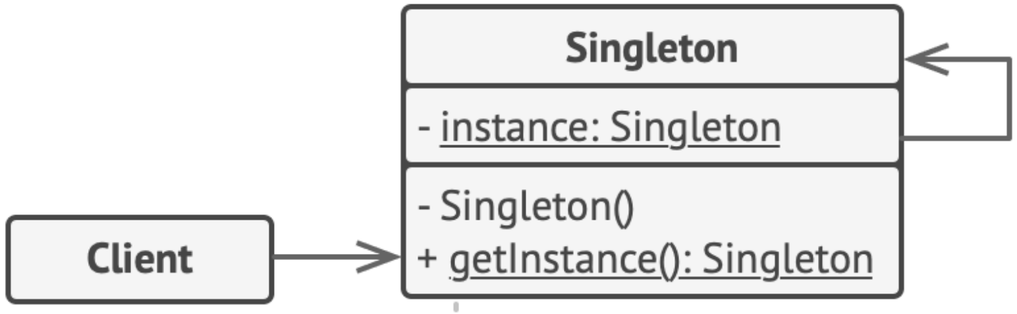
Singleton

Motivación - Solución

- **Para garantizar que una clase tenga una única instancia:**
  - Hacemos privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.
  - Creamos un método de creación estático que actúa como constructor.

- **Proporcionar un punto de acceso global a la instancia:**
  - La primera vez que llamemos al método de creación, se usará el método “new” y guardará el objeto en un atributo. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Singleton



1 La clase **Singleton** declara el método estático `obtenerInstancia` que devuelve la misma instancia de su propia clase.

El constructor del Singleton debe ocultarse del código cliente. La llamada al método `obtenerInstancia` debe ser la única manera de obtener el objeto de Singleton.

```
if (instance == null) {
    // Nota: si estás creando una aplicación
    // que soporte el multihilo, debes
    // colocar un bloqueo de hilo aquí.
    instance = new Singleton()
}
return instance
```

Singleton

Pros y contras

- ✓ Puedes tener la certeza de que una clase tiene una única instancia.
- ✓ Obtienes un punto de acceso global a dicha instancia.
- ✓ El objeto Singleton solo se inicializa cuando se requiere por primera vez.
- ✗ Vulnera el *Principio de responsabilidad única*. El patrón resuelve dos problemas al mismo tiempo.
- ✗ El patrón Singleton puede enmascarar un mal diseño, por ejemplo, cuando los componentes del programa saben demasiado los unos sobre los otros.
- ✗ El patrón requiere de un tratamiento especial en un entorno con múltiples hilos de ejecución, para que varios hilos no creen un objeto Singleton varias veces.
- ✗ Puede resultar complicado realizar la prueba unitaria del código cliente del Singleton porque muchos *frameworks* de prueba dependen de la herencia a la hora de crear objetos simulados (mock objects). Debido a que la clase Singleton es privada y en la mayoría de los lenguajes resulta imposible sobrescribir métodos estáticos, tendrás que pensar en una manera original de simular el Singleton. O, simplemente, no escribas las pruebas. O no utilices el patrón Singleton.

Singleton

Ejemplo

- Ejemplo en el zip de esta clase.

Bibliografía

Enlaces útiles

- Abstract factory: https://refactoring.guru/es/design-patterns/abstract-factory
- Builder: https://refactoring.guru/es/design-patterns/builder
- Factory: https://refactoring.guru/es/design-patterns/factory-method
- Singleton: https://refactoring.guru/es/design-patterns/singleton

¿Consultas?



Ingeniería de Software

17 -Patrones de Diseño III: Creacionales

IIC2143-3

Josefa España

jpespana@uc.cl