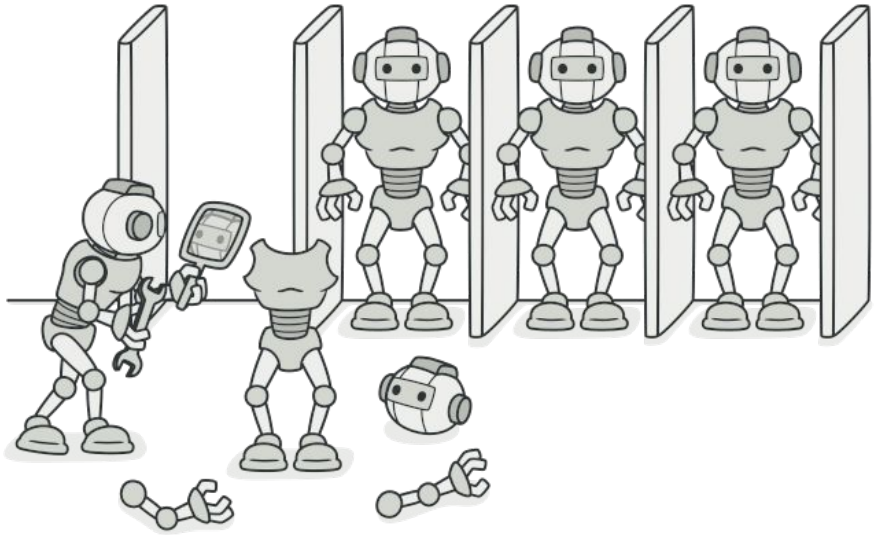
Patrones de Diseño

-Miguel Angel Deza Cuela
-Josue Gabriel Sumare Uscca
-Katherine Nikole Béjar Román
-Yanira Angie Suni Quispe

Prototype

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



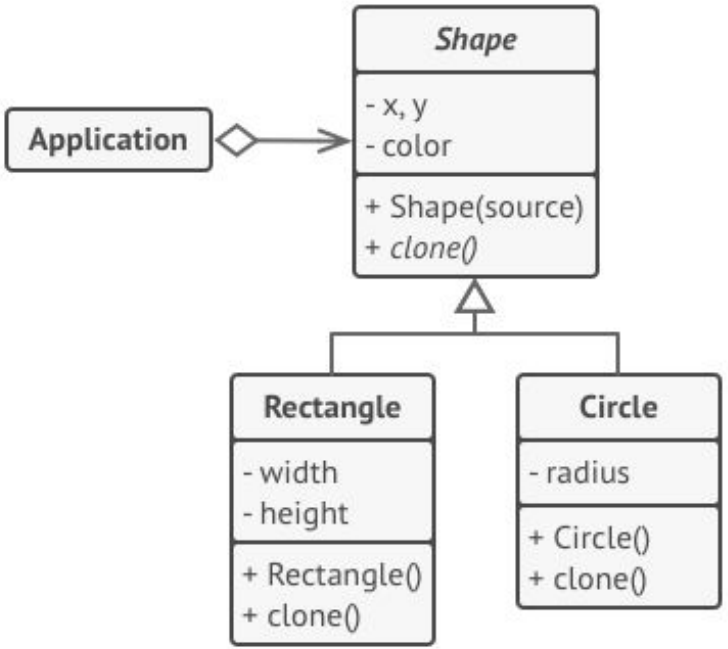
No siempre *es posible copiar un objeto “desde fuera”.*

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`.

Un objeto que soporta la clonación se denomina *prototipo*. Cuando tus objetos tienen decenas de campos y miles de configuraciones posibles, la clonación puede servir como alternativa a la creación de subclases.

Los prototipos prefabricados pueden ser una alternativa a las subclases.

En este ejemplo, el patrón Prototype nos permite producir copias exactas de objetos geométricos sin acoplar el código a sus clases.



Puntos Clave

- **Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.**
- El patrón Prototype proporciona al código cliente una interfaz general para trabajar con todos los objetos que soportan la clonación. Esta interfaz hace que el código cliente sea independiente de las clases concretas de los objetos que clona.
- **Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.**

Relaciones con otros patrones

- Muchos diseños empiezan utilizando el Factory Method (menos complicado y más personalizable mediante las subclases) y evolucionan hacia Abstract Factory, Prototype, o Builder (más flexibles, pero más complicados)
  .
- Las clases del Abstract Factory a menudo se basan en un grupo de métodos de fábrica, pero también puedes utilizar Prototype para escribir los métodos de estas clases.

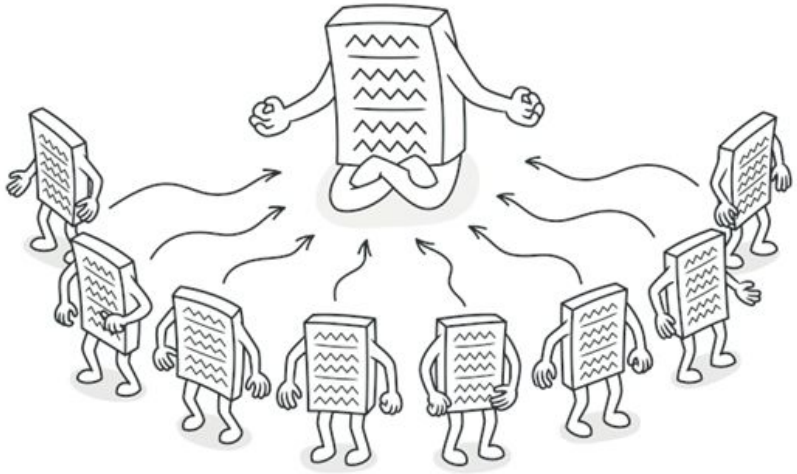
- Prototype puede ayudar cuando necesitas guardar copias de Comandos en un historial.

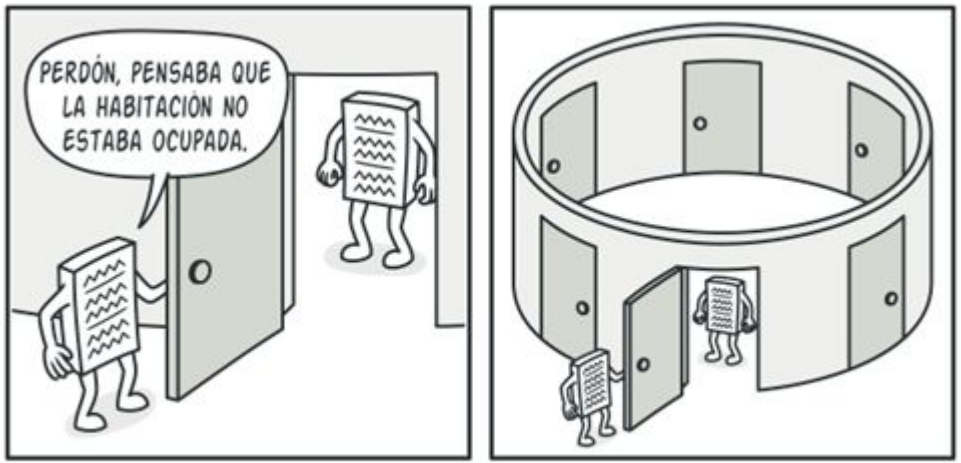
Ventajas de su Uso

- Puedes clonar objetos sin acoplarlos a sus clases concretas.
- Puedes evitar un código de inicialización repetido clonando prototipos prefabricados.
- Puedes crear objetos complejos con más facilidad.

Singleton

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.





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

¿En qué situaciones se aplicaría?

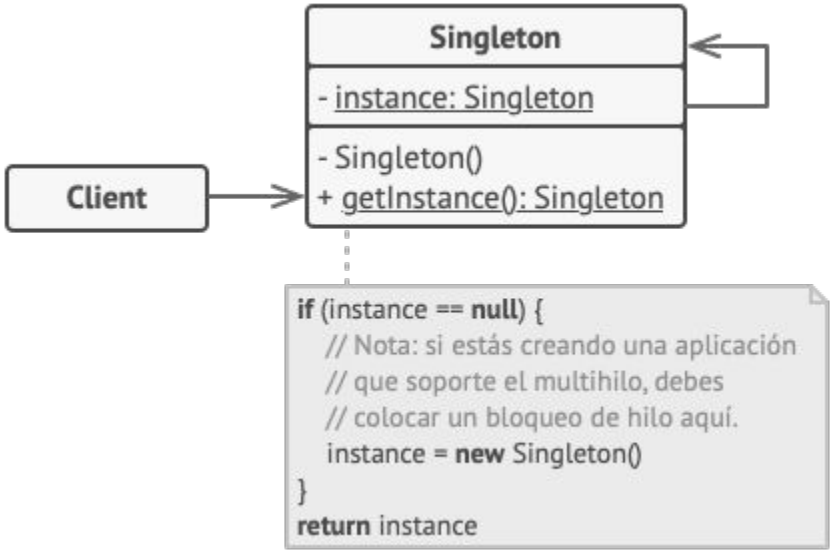
- Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.
    - El patrón Singleton deshabilita el resto de las maneras de crear objetos de una clase, excepto el método especial de creación. Este método crea un nuevo objeto, o bien devuelve uno existente si ya ha sido creado.

- Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.
    - Al contrario que las variables globales, el patrón Singleton garantiza que haya una única instancia de una clase. A excepción de la propia clase Singleton, nada puede sustituir la instancia en caché.
    - Ten en cuenta que siempre podrás ajustar esta limitación y permitir la creación de cierto número de instancias Singleton. La única parte del código que requiere cambios es el cuerpo del método **`getInstance.`**

1

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

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



Ventajas

- 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.

Desventajas

- 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.

Relaciones con otros patrones

- Una clase **Facade** a menudo puede transformarse en una **Singleton**, ya que un único objeto fachada es suficiente en la mayoría de los casos.

- **Flyweight** podría asemejarse a **Singleton** si de algún modo pudieras reducir todos los estados compartidos de los objetos a un único objeto flyweight. Pero existen dos diferencias fundamentales entre estos patrones:

  1. Solo debe haber una instancia Singleton, mientras que una clase Flyweight puede tener varias instancias con distintos estados intrínsecos.
  2. El objeto Singleton puede ser mutable. Los objetos flyweight son inmutables.

- Los patrones Abstract Factory, Builder y Prototype pueden todos ellos implementarse como Singletons.

Abstract Factory

Es un patrón creacional que en comparación con el Factory , en el que se devuelve objetos de diferentes tipos, en este caso , se devolverá diferentes clases Factory según el parámetro pasado, en pocas palabras nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

Situaciones en las que se aplica Abstract Factory

Imagina que estás creando un simulador de tienda de muebles. Tu código está compuesto por clases que representan lo siguiente:

1. Una familia de productos relacionados, digamos: `Silla` + `Sofá` + `Mesilla`.
2. Algunas variantes de esta familia. Por ejemplo, los productos `Silla` + `Sofá` + `Mesilla` están disponibles en estas variantes: `Moderna`, `Victoriana`, `ArtDecó`.

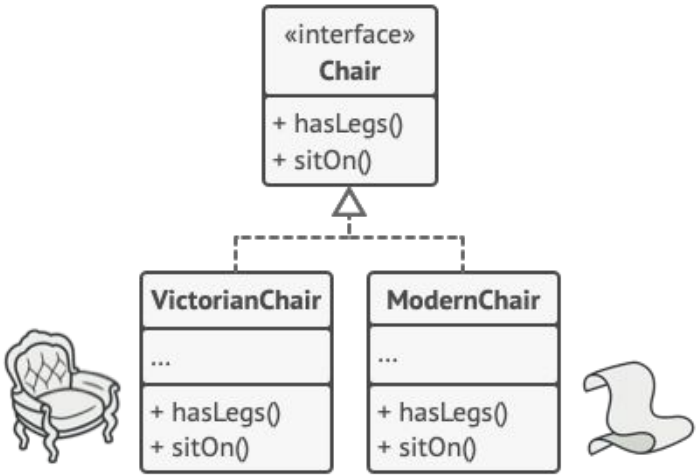
Familias de productos y sus variantes.

Necesitamos una forma de crear objetos individuales de mobiliario para que combinen con otros objetos de la misma familia. Los clientes se enfadan bastante cuando reciben muebles que no combinan.

Además, no queremos cambiar el código existente al añadir al programa nuevos productos o familias de productos. Los comerciantes de muebles actualizan sus catálogos muy a menudo, y debemos evitar tener que cambiar el código principal cada vez que esto ocurra

Paso 1

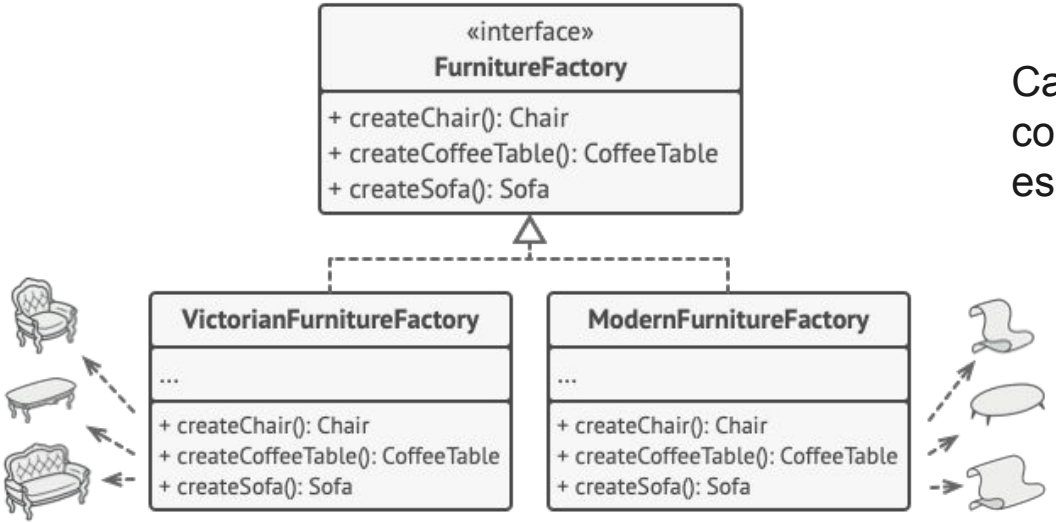
Lo primero que sugiere el patrón Abstract Factory es que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos (por ejemplo, silla, sofá o mesilla). Después podemos hacer que todas las variantes de los productos sigan esas interfaces. Por ejemplo, todas las variantes de silla pueden implementar la interfaz `Silla`, así como todas las variantes de mesilla pueden implementar la interfaz `Mesilla`, y así sucesivamente.



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

Paso 2

El siguiente paso consiste en declarar la *Fábrica abstracta*: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos (por ejemplo, `crearSilla`, `crearSofá` y `crearMesilla`). Estos métodos deben devolver productos abstractos representados por las interfaces que extrajimos previamente: `Silla`, `Sofá`, `Mesilla`, etc.



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

Ahora bien, ¿qué hay de las variantes de los productos? Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz `FábricaAbstracta`. Una fábrica es una clase que devuelve productos de un tipo particular. Por ejemplo, la `FábricadeMueblesModernos` sólo puede crear objetos de `SillaModerna`, `SofáModerno` y `MesillaModerna`.

El código cliente tiene que funcionar con fábricas y productos a través de sus respectivas interfaces abstractas. Esto nos permite cambiar el tipo de fábrica que pasamos al código cliente, así como la variante del producto que recibe el código cliente, sin descomponer el propio código cliente.

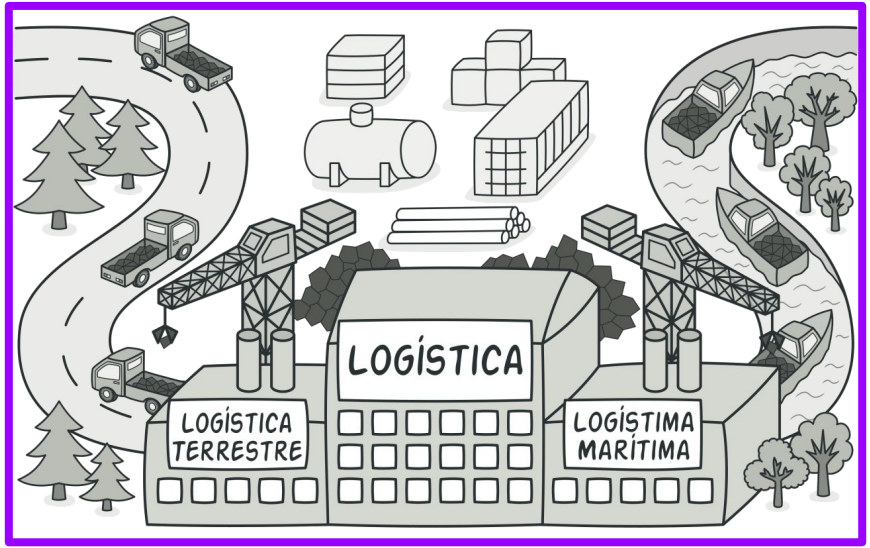
Al cliente no le debe importar la clase concreta de la fábrica con la que funciona.

Digamos que el cliente quiere una fábrica para producir una silla. El cliente no tiene que conocer la clase de la fábrica y tampoco importa el tipo de silla que obtiene. Ya sea un modelo moderno o una silla de estilo victoriano, el cliente debe tratar a todas las sillas del mismo modo, utilizando la interfaz abstracta `Silla`. Con este sistema, lo único que sabe el cliente sobre la silla es que implementa de algún modo el método `sentarse`. Además, sea cual sea la variante de silla devuelta, siempre combinará con el tipo de sofá o mesilla producida por el mismo objeto de fábrica.

Queda otro punto por aclarar: si el cliente sólo está expuesto a las interfaces abstractas, ¿cómo se crean los objetos de fábrica? Normalmente, la aplicación crea un objeto de fábrica concreto en la etapa de inicialización. Justo antes, la aplicación debe seleccionar el tipo de fábrica, dependiendo de la configuración o de los ajustes del entorno.

Factory Method

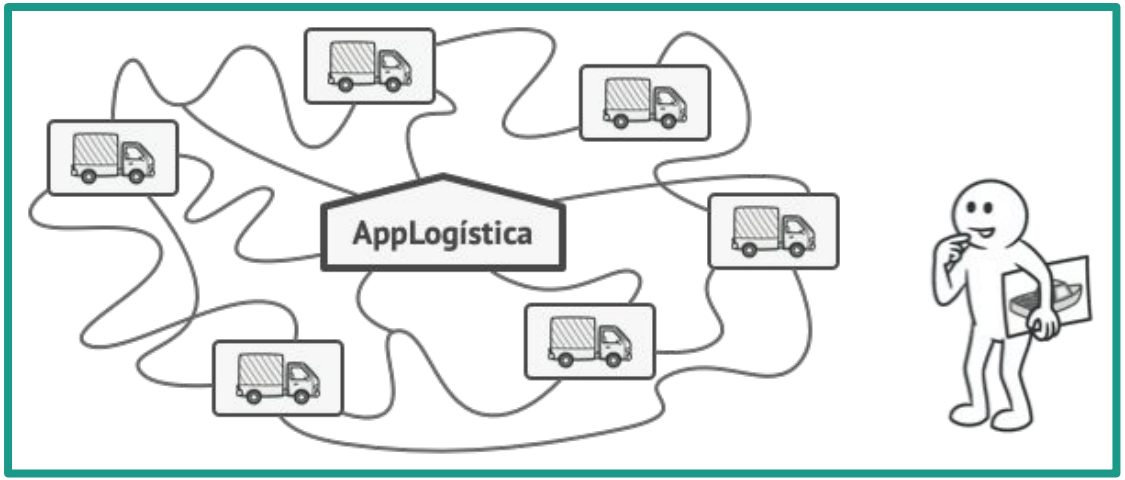
Factory method es un patrón diseño creacional que utiliza métodos de fábrica para tratar el problema de crear objetos sin tener que especificar la clase exacta del objeto que se creará. Esto se hace creando objetos llamando a un método de fábrica, ya sea especificado en una interfaz e implementado por clases secundarias, o implementado en una clase base y opcionalmente anulado por clases derivadas, en lugar de llamar a un constructor .



Situaciones en las que se aplica

- Se utiliza cuando no se conozca las dependencias y los tipos exactos de los objetos con lo que deba funcionar el código.
- Cuando se quiera ofrecer a los usuarios del framework o biblioteca, una forma de expandir el mismo
- Se aplica al querer ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez.

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.

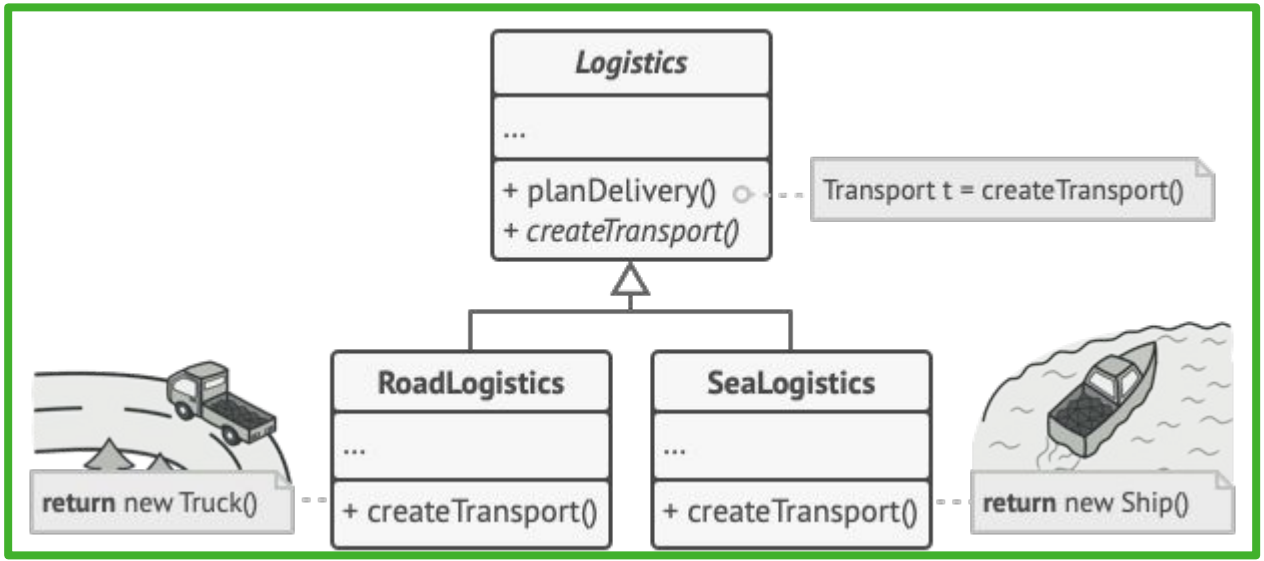


Añadir una nueva clase al programa no es tan sencillo si el resto del código ya está acoplado a clases existentes.

Pero ¿qué pasa con el código? En este momento, la mayor parte de tu 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 decides añadir otro tipo de transporte a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.
Al final acabarás 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 llamar al operador new para construir objetos directamente, se invoque a un método *fábrica* especial. 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*.

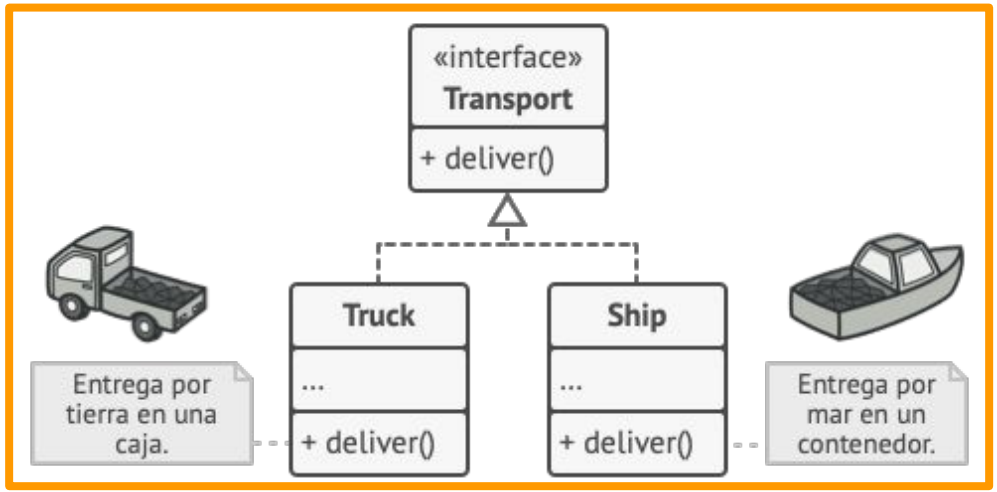


Las subclases pueden alterar la clase de los objetos devueltos por el método fábrica.

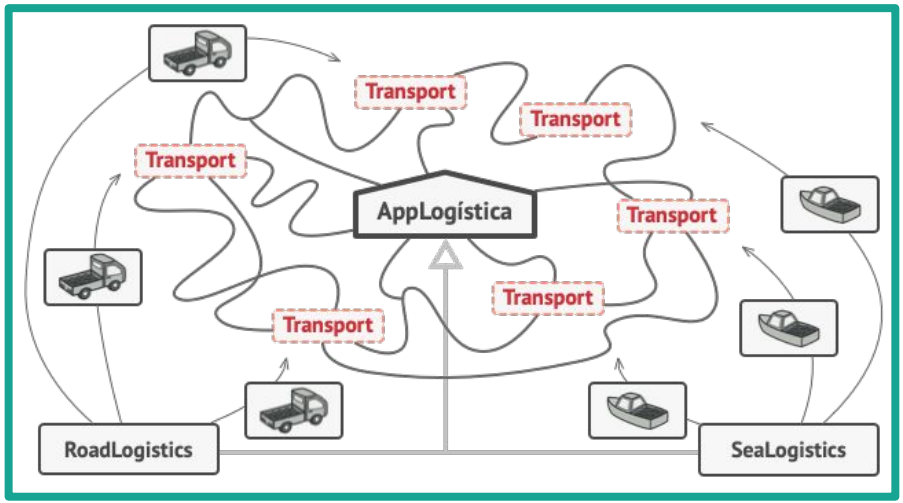
A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, piensa en esto: ahora puedes sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

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.

Todos los productos deben seguir la misma interfaz.



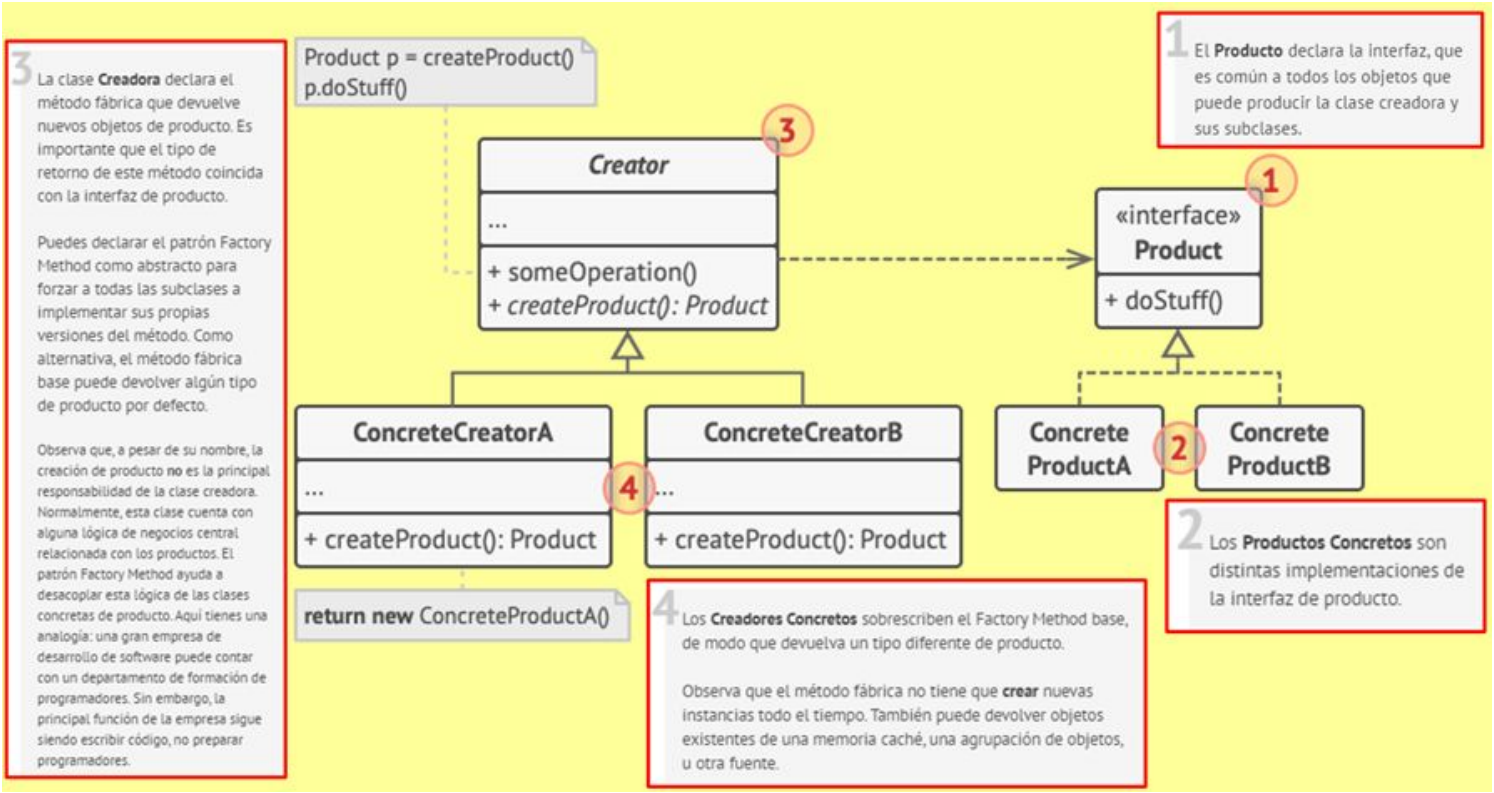
Por ejemplo, tanto la clase Camión como la clase Barco deben implementar la interfaz Transporte, que declara un método llamado entrega. Cada clase implementa este método de forma diferente: los camiones entregan su carga por tierra, mientras que los barcos lo hacen por mar. El método fábrica dentro de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



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

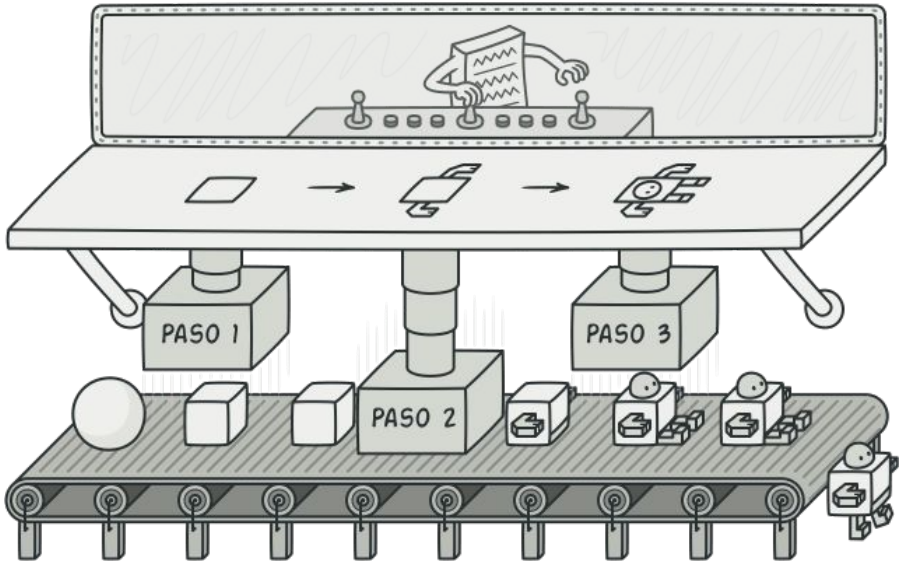
El código que utiliza el método fábrica (a menudo denominado código *cliente*) no encuentra diferencias entre los productos devueltos por varias subclases, y trata a todos los productos como la clase abstracta Transporte. El cliente sabe que todos los objetos de transporte deben tener el método entrega,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

Estructura


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.
Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
«interface»
Product
+ doStuff()
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
Concrete ProductA
Concrete ProductB
return new ConcreteProductA()
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.

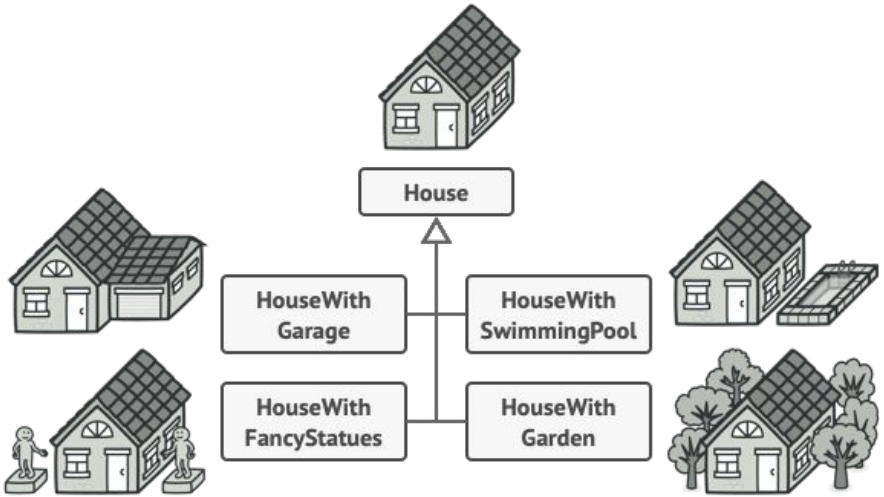
Builder

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.



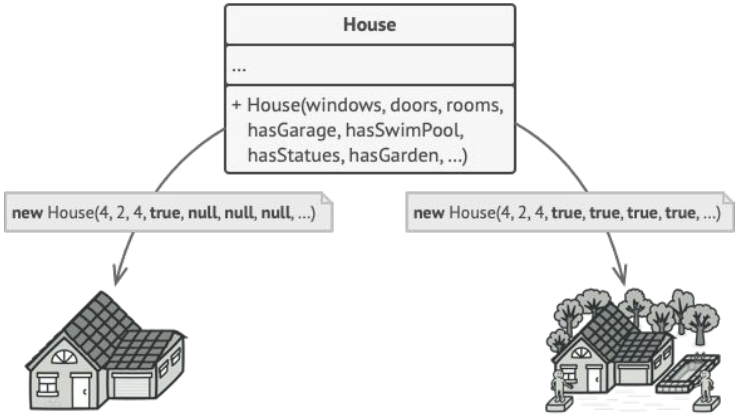
Situaciones en las que se aplica Builder

Imagina un objeto complejo que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados. Normalmente, este código de inicialización está sepultado dentro de un monstruoso constructor con una gran cantidad de parámetros. O, peor aún: disperso por todo el código cliente.



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

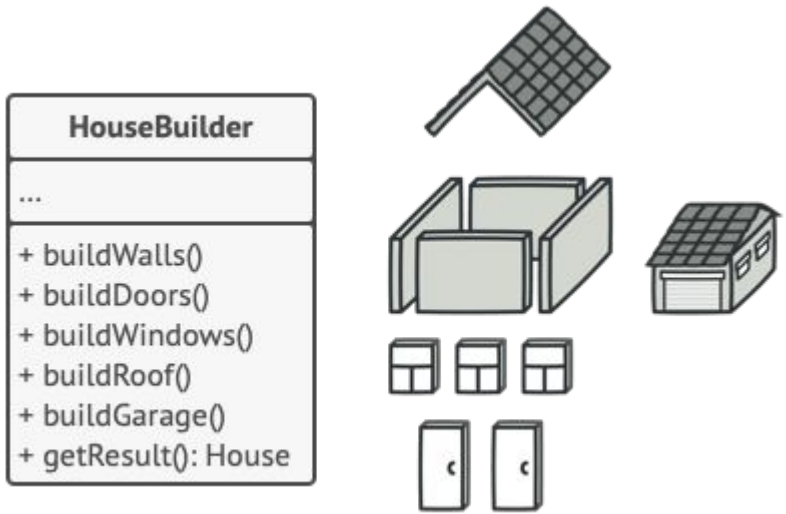
Por ejemplo, pensemos en cómo crear un objeto `Casa`. Para construir una casa sencilla, debemos construir cuatro paredes y un piso, así como instalar una puerta, colocar un par de ventanas y ponerle un tejado. Pero ¿qué pasa si quieres una casa más grande y luminosa, con un jardín y otros extras (como sistema de calefacción, instalación de fontanería y cableado eléctrico)? .La solución más sencilla es extender la clase base `Casa` y crear un grupo de subclases que cubran todas las combinaciones posibles de los parámetros. Pero, en cualquier caso, acabarás con una cantidad considerable de subclases. Cualquier parámetro nuevo, como el estilo del porche, exigirá que incrementes esta jerarquía aún más.Existe otra posibilidad que no implica generar subclases. Puedes crear un enorme constructor dentro de la clase base `Casa` con todos los parámetros posibles para controlar el objeto casa. Aunque es cierto que esta solución elimina la necesidad de las subclases, genera otro problema.En la mayoría de los casos, gran parte de los parámetros no se utilizará, lo que provocará que las llamadas al constructor sean bastante feas. Por ejemplo, solo una pequeña parte de las casas tiene piscina, por lo que los parámetros relacionados con piscinas serán inútiles en nueve de cada diez casos.



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

Solución

El patrón Builder sugiere que saques el código de construcción del objeto de su propia clase y lo coloques dentro de objetos independientes llamados constructores.



El patrón organiza la construcción de objetos en una serie de pasos (`construirParedes`, `construirPuerta`, etc.). Para crear un objeto, se ejecuta una serie de estos pasos en un objeto constructor. Lo importante es que no necesitas invocar todos los pasos. Puedes invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto.

Puede ser que algunos pasos de la construcción necesiten una implementación diferente cuando tengamos que construir distintas representaciones del producto. Por ejemplo, las paredes de una cabaña pueden ser de madera, pero las paredes de un castillo tienen que ser de piedra.

En este caso, podemos crear varias clases constructoras distintas que implementen la misma serie de pasos de construcción, pero de forma diferente. Entonces podemos utilizar estos constructores en el proceso de construcción (por ejemplo, una serie ordenada de llamadas a los pasos de construcción) para producir distintos tipos de objetos.

Por ejemplo, imagina un constructor que construye todo de madera y vidrio, otro que construye todo con piedra y hierro y un tercero que utiliza oro y diamantes. Al invocar la misma serie de pasos, obtenemos una casa normal del primer constructor, un pequeño castillo del segundo y un palacio del tercero. Sin embargo, esto sólo funcionaría si el código cliente que invoca los pasos de construcción es capaz de interactuar con los constructores mediante una interfaz común.

Clase directora

Puedes ir más lejos y extraer una serie de llamadas a los pasos del constructor que utilizas para construir un producto y ponerlas en una clase independiente llamada *directora*. La clase directora define el orden en el que se deben ejecutar los pasos de construcción, mientras que el constructor proporciona la implementación de dichos pasos.

No es estrictamente necesario tener una clase directora en el programa, ya que se pueden invocar los pasos de construcción en un orden específico directamente desde el código cliente. No obstante, la clase directora puede ser un buen lugar donde colocar distintas rutinas de construcción para poder reutilizarlas a lo largo del programa.

Además, la clase directora esconde por completo los detalles de la construcción del producto al código cliente. El cliente sólo necesita asociar un objeto constructor con una clase directora, utilizarla para iniciar la construcción, y obtener el resultado del objeto constructor.