Ingeniería de Software II

Patrones de diseño Creacionales

2023

Facultad de Ciencia y Tecnología
Universidad Autónoma de Entre Ríos

Sumérgete en los patrones de diseño

Alexander Shvets, Refactoring.Guru, 2019

support@refactoring.guru

https://refactoring.guru/es/design-patterns

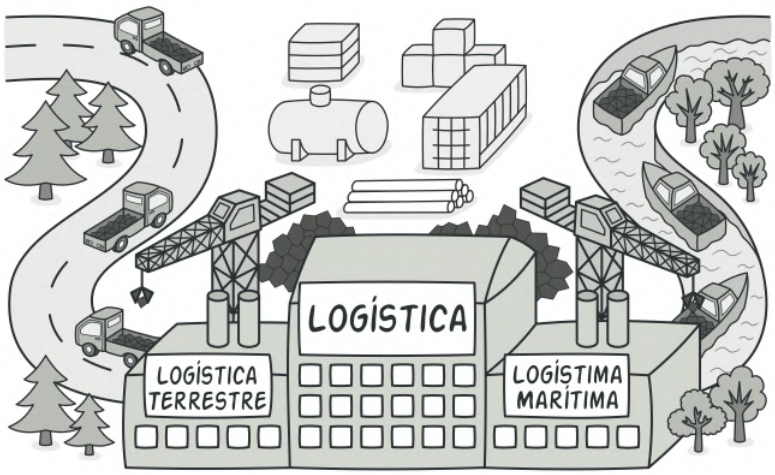
PATRONES CREACIONALES

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.

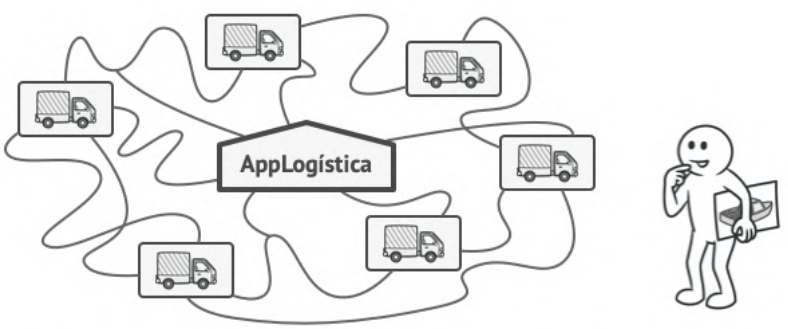
https://refactoring.guru/es/design-patterns/factory-method



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.



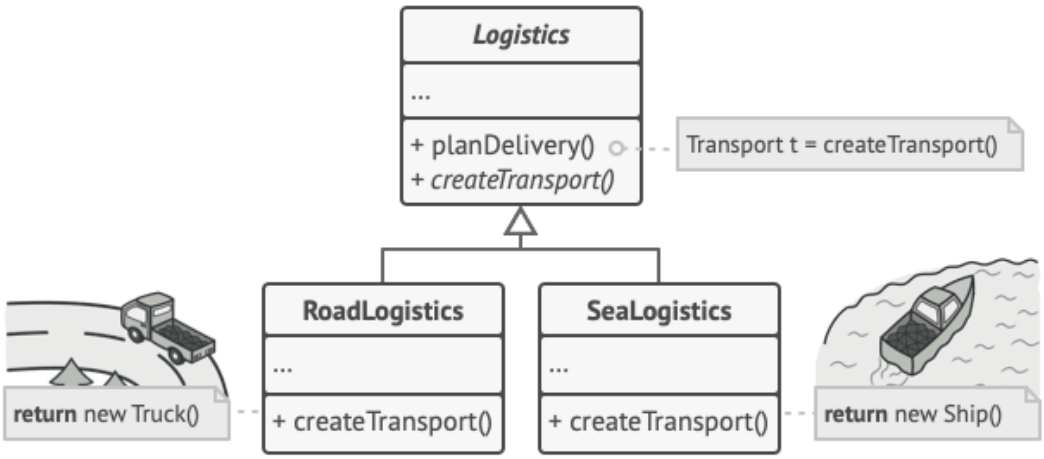
Añadir una nueva clase al programa no es tan sencillo si el resto del código ya está acoplado a clases existentes.

Estupendo, ¿verdad? 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

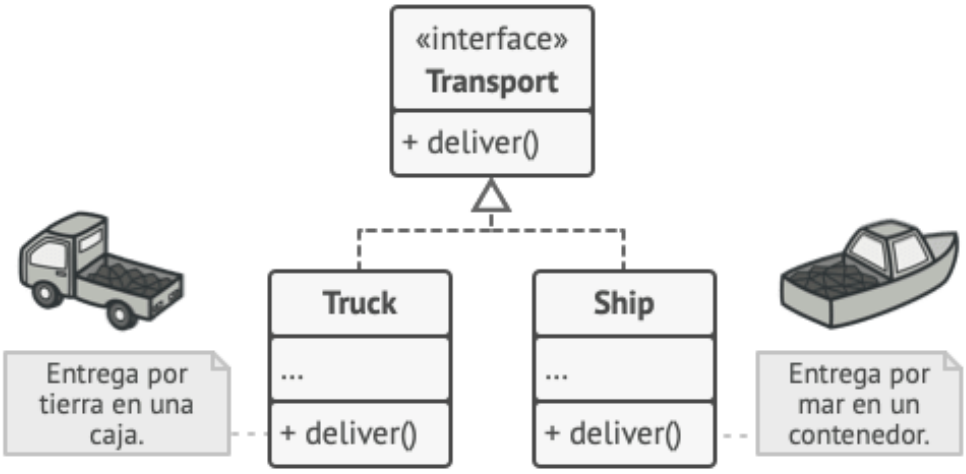
El patrón Factory Method sugiere que, en lugar de llamar al operador ***new*** para construir objetos directamente, se invoque a un método *fábrica* especial. No te preocupes: los objetos se siguen creando a través del operador new, pero se invocan desde el método fábrica. Los objetos devueltos por el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



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

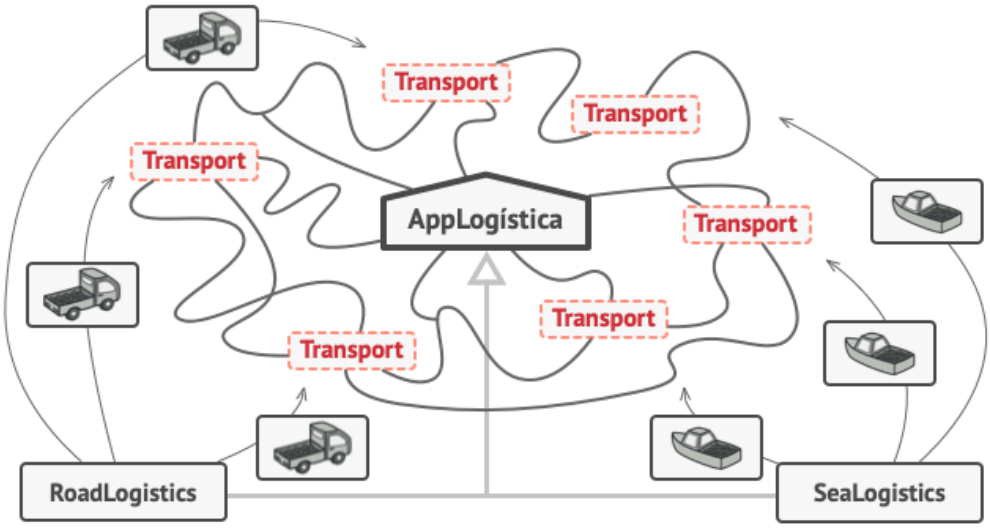
A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, piensa en esto: ahora puedes sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

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.

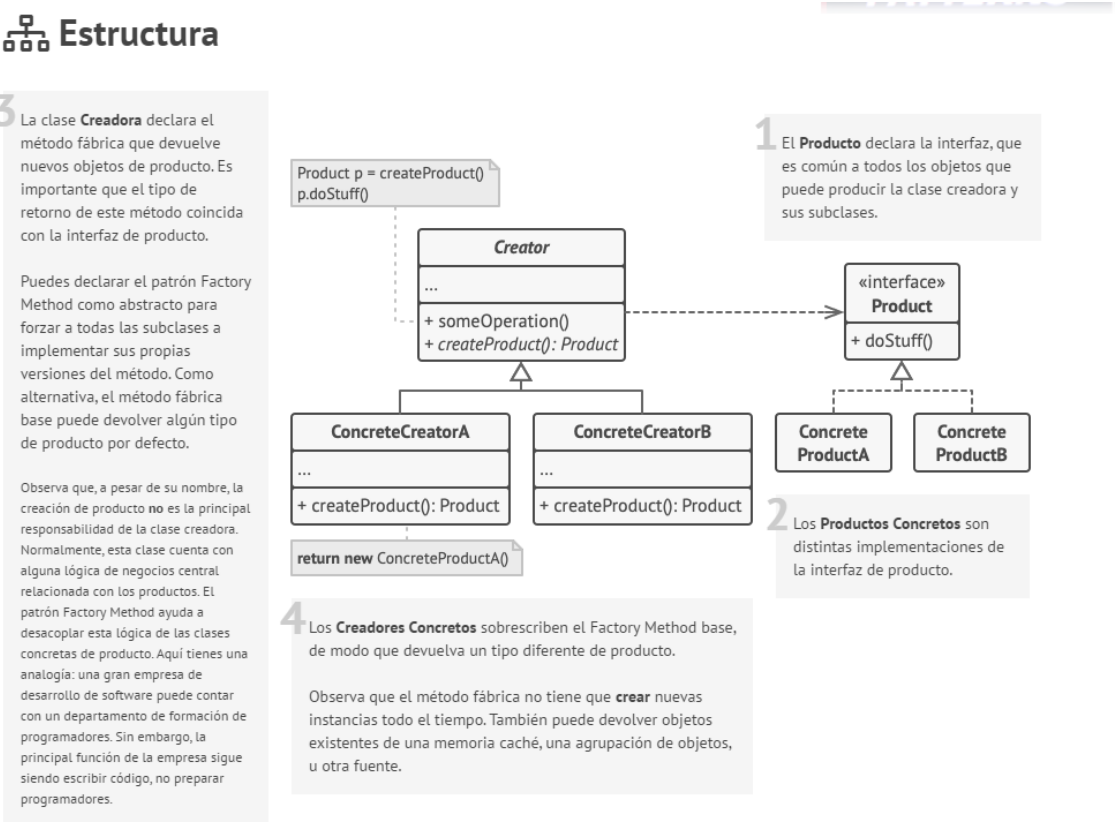
Por ejemplo, tanto la clase ***Camión*** como la clase ***Barco*** deben implementar la interfaz Transporte, que declara un método llamado entrega. Cada clase implementa este método de forma diferente: los camiones entregan su carga por tierra, mientras que los barcos lo hacen por mar. El método fábrica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

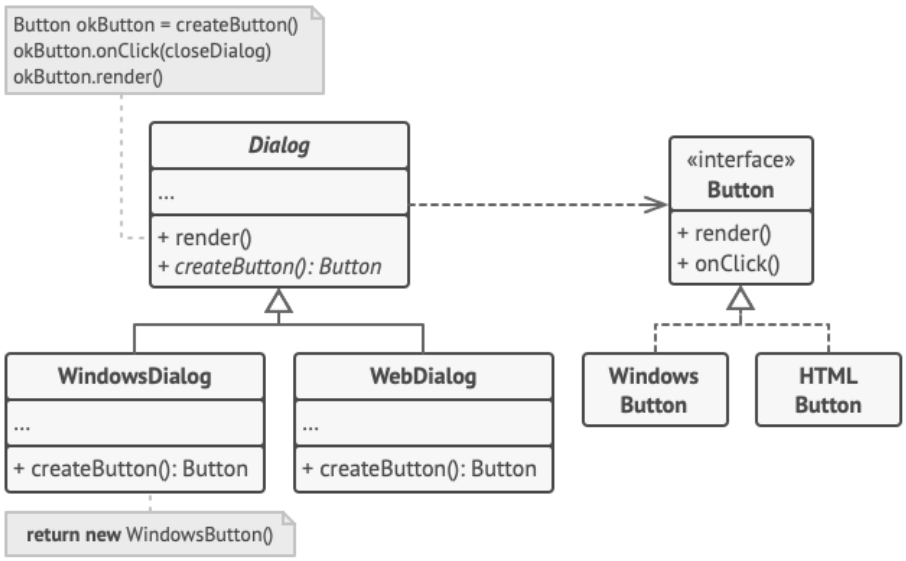
El código que utiliza el método fábrica (a menudo denominado código cliente) no encuentra diferencias entre los productos devueltos por varias subclases, y trata a todos los productos como la clase abstracta ***Transporte***. El cliente sabe que todos los objetos de transporte deben tener el método ***entrega***,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

Estructura



Pseudocódigo

Este ejemplo ilustra cómo puede utilizarse el patrón **Factory Method** para crear elementos de interfaz de usuario (UI) multiplataforma sin acoplar el código cliente a clases UI concretas.



Ejemplo del diálogo multiplataforma.

La clase base de diálogo utiliza distintos elementos UI para representar su ventana. En varios sistemas operativos, estos elementos pueden tener aspectos diferentes, pero su comportamiento debe ser consistente. Un botón en Windows sigue siendo un botón en Linux.

Cuando entra en juego el patrón Factory Method no hace falta reescribir la lógica del diálogo para cada sistema operativo. Si declaramos un patrón Factory Method que produce botones dentro de la clase base de diálogo, más tarde podremos crear una subclase de diálogo que devuelva botones al estilo de Windows desde el Factory Method. Entonces la subclase hereda la mayor parte del código del diálogo de la clase base, pero, gracias al Factory Method, puede representar botones al estilo de Windows en pantalla.

Para que este patrón funcione, la clase base de diálogo debe funcionar con botones abstractos, es decir, una clase base o una interfaz que sigan todos los botones concretos. De este modo, el código sigue siendo funcional, independientemente del tipo de botones con el que trabaje.

Por supuesto, también se puede aplicar este sistema a otros elementos UI. Sin embargo, con cada nuevo método de fábrica que añadas al diálogo, más te acercarás al patrón **Abstract Factory**. No temas, más adelante hablaremos sobre este patrón.

```
// La clase creadora declara el método fábrica que debe devolver
// un objeto de una clase de producto. Normalmente, las
// subclases de la creadora proporcionan la implementación de
// este método.
class Dialog is
    // La creadora también puede proporcionar cierta
    // implementación por defecto del método fábrica.
    abstract method createButton():Button

    // Observa que, a pesar de su nombre, la principal
    // responsabilidad de la creadora no es crear productos.
    // Normalmente contiene cierta lógica de negocio que depende
    // de los objetos de producto devueltos por el método
    // fábrica. Las subclases pueden cambiar indirectamente esa
    // lógica de negocio sobrescribiendo el método fábrica y
    // devolviendo desde él un tipo diferente de producto.
    method render() is
        // Invoca el método fábrica para crear un objeto de
        // producto.
        Button okButton = createButton()
        // Ahora utiliza el producto.
        okButton.onClick(closeDialog)
        okButton.render()


// Los creadores concretos sobrescriben el método fábrica para
// cambiar el tipo de producto resultante.
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()


// La interfaz de producto declara las operaciones que todos los
// productos concretos deben implementar.
interface Button is
    method render()
    method onClick(f)

// Los productos concretos proporcionan varias implementaciones
// de la interfaz de producto.

class WindowsButton implements Button is
    method render(a, b) is
        // Representa un botón en estilo Windows.
    method onClick(f) is
        // Vincula un evento clic de OS nativo.

class HTMLButton implements Button is
    method render(a, b) is
        // Devuelve una representación HTML de un botón.
    method onClick(f) is
        // Vincula un evento clic de navegador web.

class Application is
    field dialog: Dialog

    // La aplicación elige un tipo de creador dependiendo de la
    // configuración actual o los ajustes del entorno.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    // El código cliente funciona con una instancia de un
    // creador concreto, aunque a través de su interfaz base.
    // Siempre y cuando el cliente siga funcionando con el
    // creador a través de la interfaz base, puedes pasarle
    // cualquier subclase del creador.
    method main() is
        this.initialize()
        dialog.render()
```

Aplicabilidad

"Utiliza el Método Fábrica cuando no conozcas de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código."

- ✓ El patrón Factory Method separa el código de construcción de producto del código que hace uso del producto. Por ello, es más fácil extender el código de construcción de producto de forma independiente al resto del código.
  Por ejemplo, para añadir un nuevo tipo de producto a la aplicación, sólo tendrás que crear una nueva subclase creadora y sobrescribir el Factory Method que contiene.

"Utiliza el Factory Method cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos."

- ✓ La herencia es probablemente la forma más sencilla de extender el comportamiento por defecto de una biblioteca o un framework. Pero, ¿cómo reconoce el framework si debe utilizar tu subclase en lugar de un componente estándar?
  La solución es reducir el código que construye componentes en todo el framework a un único patrón Factory Method y permitir que cualquiera sobrescriba este método además de extender el propio componente.

  Veamos cómo funcionaría. Imagina que escribes una aplicación utilizando un framework de UI de código abierto. Tu aplicación debe tener botones redondos, pero el framework sólo proporciona botones cuadrados. Extiendes la clase estándar Botón con una maravillosa subclase BotónRedondo, pero ahora tienes que decirle a la clase principal FrameworkUI que utilice la nueva subclase de botón en lugar de la clase por defecto.

  Para conseguirlo, creamos una subclase UIConBotonesRedondos a partir de una clase base del framework y sobrescribimos su método crearBotón. Si bien este método devuelve objetos Botón en la clase base, haces que tu subclase devuelva objetos BotónRedondo. Ahora, utiliza la clase UIConBotonesRedondos en lugar de FrameworkUI. ¡Eso es todo!

"Utiliza el Factory Method cuando quieras ahorrar recursos del sistema mediante la reutilización de objetos existentes en lugar de reconstruirlos cada vez."

- ✓ A menudo experimentas esta necesidad cuando trabajas con objetos grandes y que consumen muchos recursos, como conexiones de bases de datos, sistemas de archivos y recursos de red.

  Pensemos en lo que hay que hacer para reutilizar un objeto existente:

1. Primero, debemos crear un almacenamiento para llevar un registro de todos los objetos creados.
2. Cuando alguien necesite un objeto, el programa deberá buscar un objeto disponible dentro de ese agrupamiento.
3. … y devolverlo al código cliente.
4. Si no hay objetos disponibles, el programa deberá crear uno nuevo (y añadirlo al agrupamiento).

   ¡Eso es mucho código! Y hay que ponerlo todo en un mismo sitio para no contaminar el programa con código duplicado.

Es probable que el lugar más evidente y cómodo para colocar este código sea el constructor de la clase cuyos objetos intentamos reutilizar. Sin embargo, un constructor siempre debe devolver nuevos objetos por definición. No puede devolver instancias existentes.

Por lo tanto, necesitas un método regular capaz de crear nuevos objetos, además de reutilizar los existentes. Eso suena bastante a lo que hace un patrón Factory Method.

Implementación

1. Haz que todos los productos sigan la misma interfaz. Esta interfaz deberá declarar métodos que tengan sentido en todos los productos.
2. Añade un patrón Factory Method vacío dentro de la clase creadora. El tipo de retorno del método deberá coincidir con la interfaz común de los productos.
3. Encuentra todas las referencias a constructores de producto en el código de la clase creadora. Una a una, sustitúyelas por invocaciones al Factory Method, mientras extraes el código de creación de productos para colocarlo dentro del Factory Method.
   Puede ser que tengas que añadir un parámetro temporal al Factory Method para controlar el tipo de producto devuelto.
   A estas alturas, es posible que el aspecto del código del Factory Method luzca bastante desagradable. Puede ser que tenga un operador switch largo que elige qué clase de producto instanciar. Pero, no te preocupes, lo arreglaremos enseguida.
4. Ahora, crea un grupo de subclases creadoras para cada tipo de producto enumerado en el Factory Method. Sobrescribe el Factory Method en las subclases y extrae las partes adecuadas de código constructor del método base.
5. Si hay demasiados tipos de producto y no tiene sentido crear subclases para todos ellos, puedes reutilizar el parámetro de control de la clase base en las subclases.

   Por ejemplo, imagina que tienes la siguiente jerarquía de clases: la clase base Correo con las subclases ***CorreoAéreo*** y ***CorreoTerrestre*** y la clase Transporte con ***Avión***, ***Camión*** y ***Tren***. La clase ***CorreoAéreo*** sólo utiliza objetos Avión, pero ***CorreoTerrestre*** puede funcionar tanto con objetos ***Camión***, como con objetos Tren. Puedes crear una nueva subclase (digamos, ***CorreoFerroviario***) que gestione ambos casos, pero hay otra opción. El código cliente puede pasar un argumento al Factory Method de la clase ***CorreoTerrestre*** para controlar el producto que quiere recibir.

6. Si, tras todas las extracciones, el Factory Method base queda vacío, puedes hacerlo abstracto. Si queda algo dentro, puedes convertirlo en un comportamiento por defecto del método.

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.

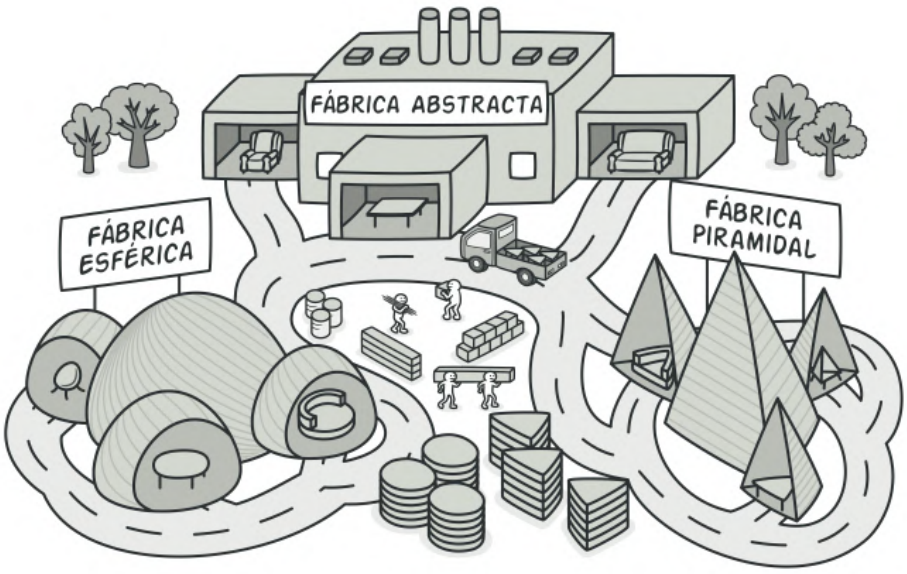
Relaciones con otros patrones

- 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.
- Puedes utilizar el patrón Factory Method junto con el Iterator para permitir que las subclases de la colección devuelvan distintos tipos de iteradores que sean compatibles con las colecciones.
- Prototype no se basa en la herencia, por lo que no presenta sus inconvenientes. No obstante, Prototype requiere de una inicialización complicada del objeto clonado. Factory Method se basa en la herencia, pero no requiere de un paso de inicialización.
- Factory Method es una especialización del Template Method. Al mismo tiempo, un Factory Method puede servir como paso de un gran Template Method.

Abstract Factory

Propósito

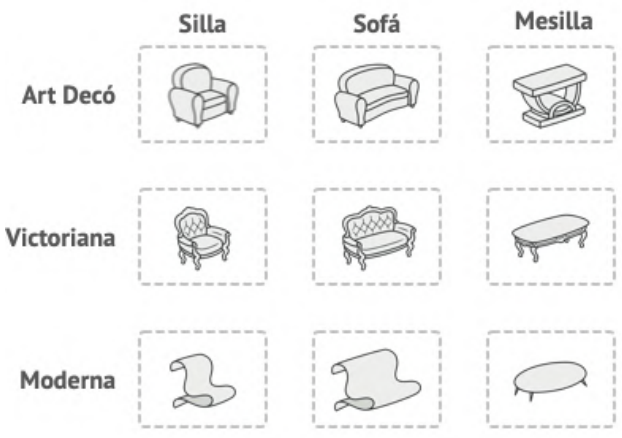
Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



Problema

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.

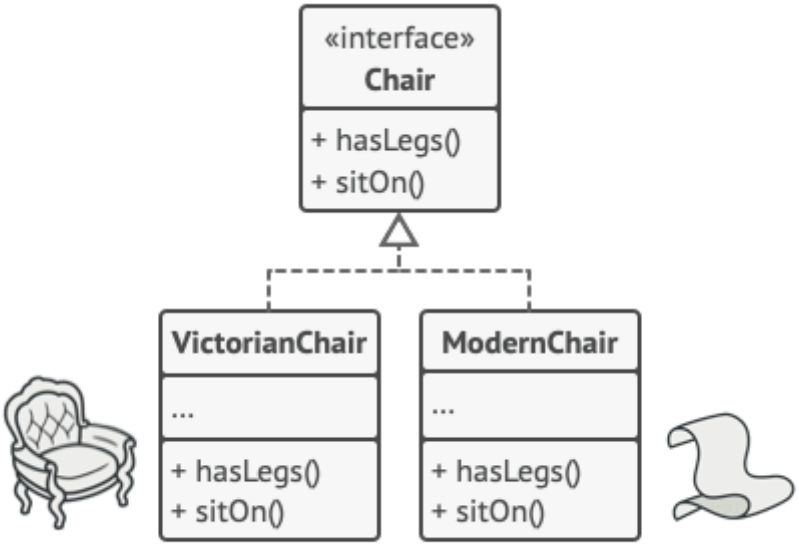
Necesitamos una forma de crear objetos individuales de mobiliario para que combinen con otros objetos de la misma familia. Los clientes se enfadan bastante cuando reciben muebles que no combinan.

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

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.

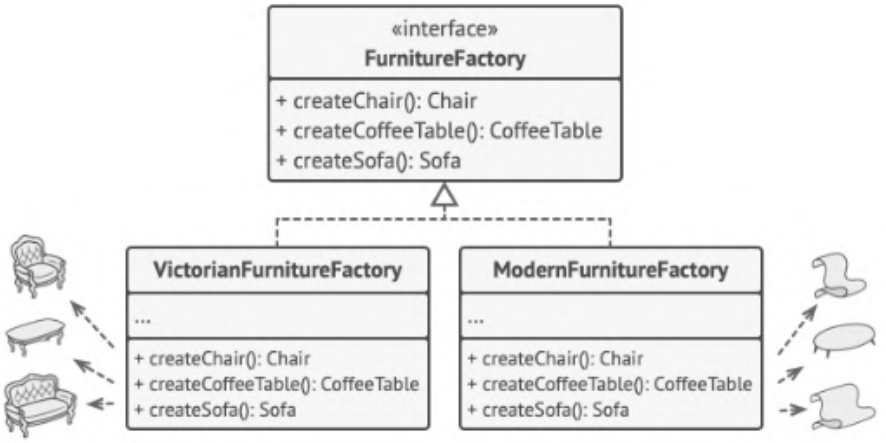
Solución

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.

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`.

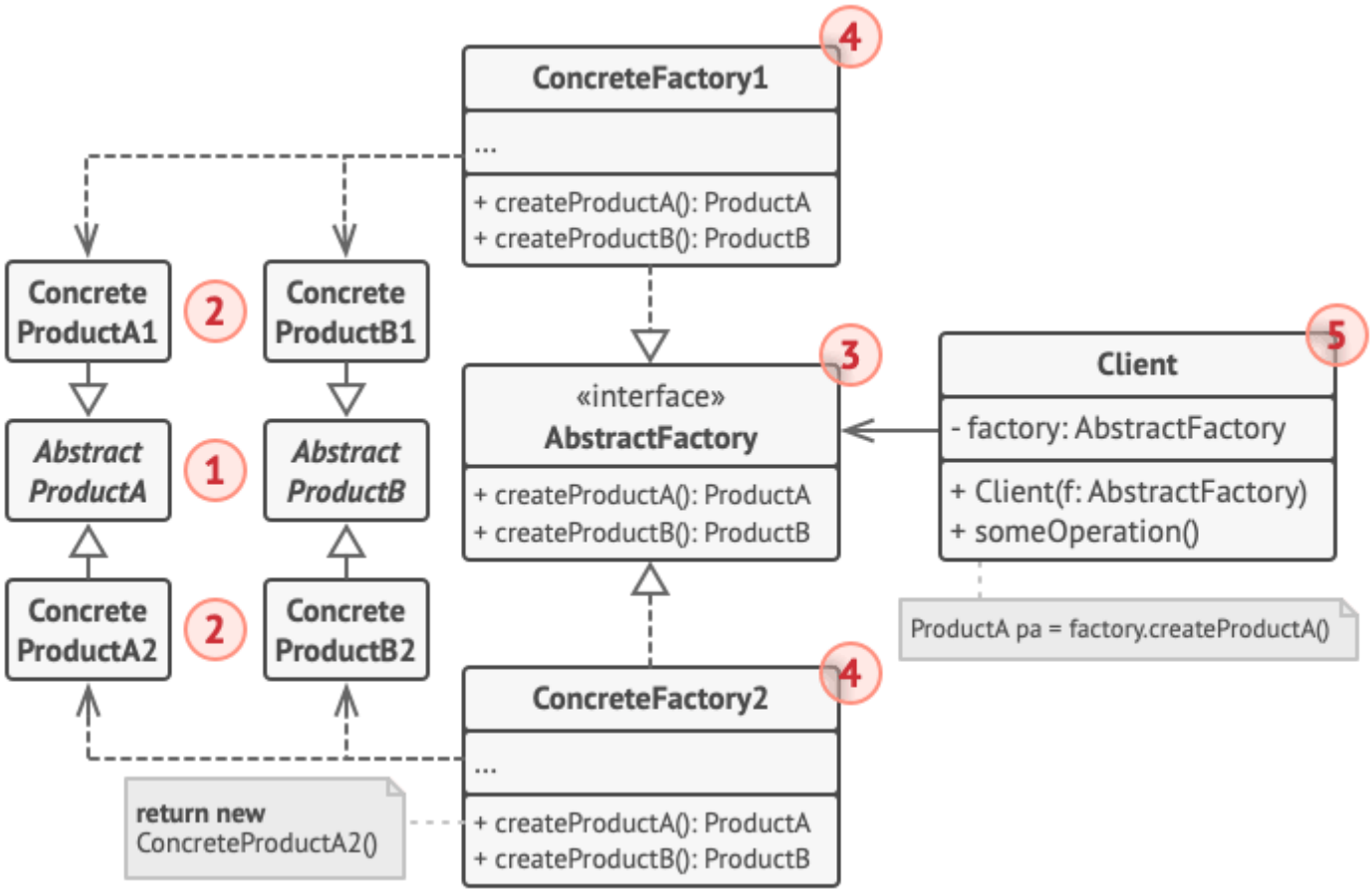
El código cliente tiene que funcionar con fábricas y productos a través de sus respectivas interfaces abstractas. Esto nos permite cambiar el tipo de fábrica que pasamos al código cliente, así como la variante del producto que recibe el código cliente, sin descomponer el propio código cliente.

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.

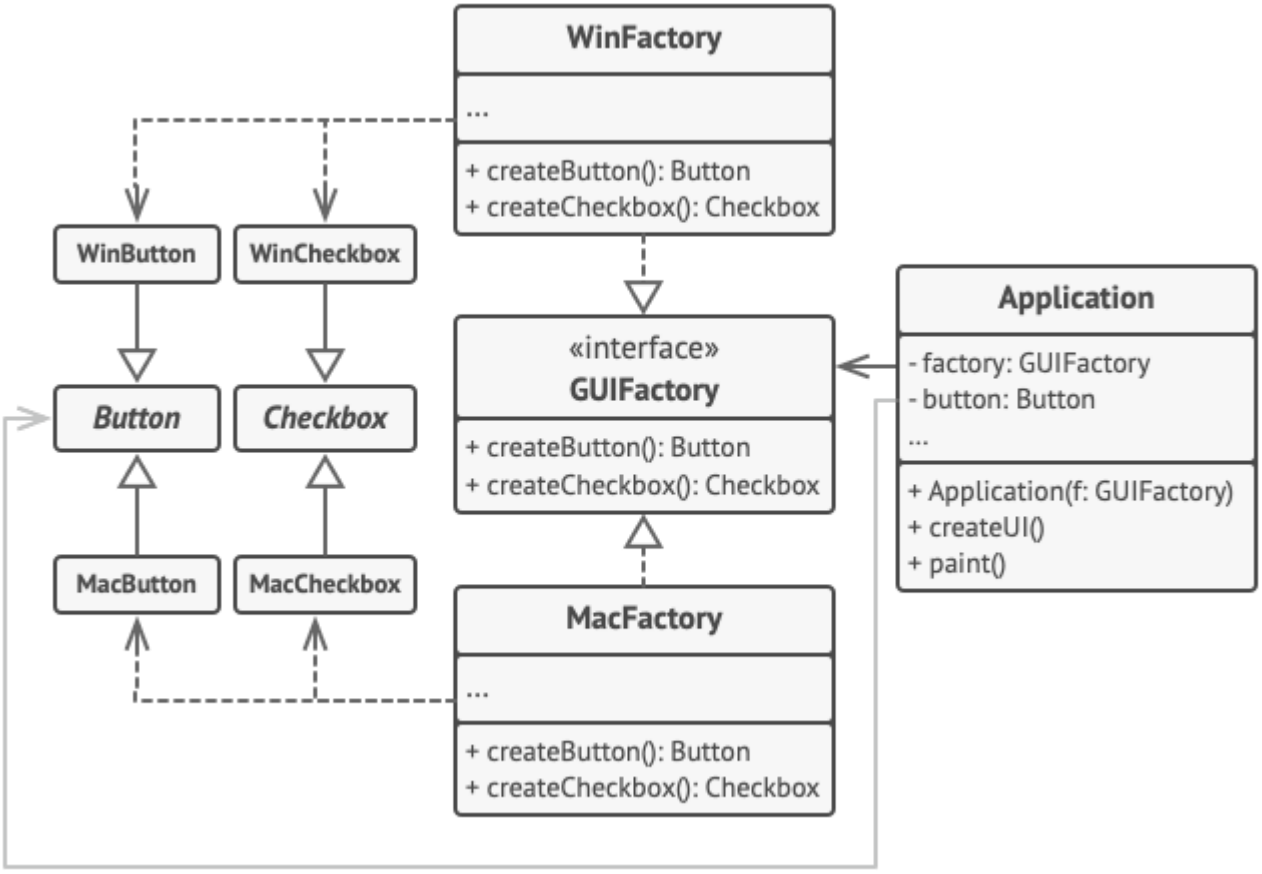
Estructura



1. Los **Productos Abstractos** declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.
2. Los **Productos Concretos** son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (silla/sofá) debe implementarse en todas las variantes dadas (victoriano/moderno).
3. La interfaz **Fábrica Abstracta** declara un grupo de métodos para crear cada uno de los productos abstractos.
4. Las **Fábricas Concretas** implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea tan solo dichas variantes de los productos.
5. Aunque las fábricas concretas instancian productos concretos, las firmas de sus métodos de creación deben devolver los productos *abstractos* correspondientes. De este modo, el código cliente que utiliza una fábrica no se acopla a la variante específica del producto que obtiene de una fábrica. El **Cliente** puede funcionar con cualquier variante fábrica/producto concreta, siempre y cuando se comunique con sus objetos a través de interfaces abstractas.

Pseudocódigo

Este ejemplo ilustra cómo puede utilizarse el patrón **Abstract Factory** para crear elementos de interfaz de usuario (UI) multiplataforma sin acoplar el código cliente a clases UI concretas, mientras se mantiene la consistencia de todos los elementos creados respecto al sistema operativo seleccionado.



Ejemplo de clases UI multiplataforma.

Es de esperar que los mismos elementos UI de una aplicación multiplataforma se comporten de forma parecida, aunque tengan un aspecto un poco diferente en distintos sistemas operativos. Además, es nuestro trabajo que los elementos UI coincidan con el estilo del sistema operativo en cuestión. No queremos que nuestro programa represente controles de macOS al ejecutarse en Windows.

La interfaz fábrica abstracta declara un grupo de métodos de creación que el código cliente puede utilizar para producir distintos tipos de elementos UI. Las fábricas concretas coinciden con sistemas operativos específicos y crean los elementos UI correspondientes.

Funciona así: cuando se lanza, la aplicación comprueba el tipo de sistema operativo actual. La aplicación utiliza esta información para crear un objeto de fábrica a partir de una clase que coincida con el sistema operativo. El resto del código utiliza esta fábrica para crear elementos UI. Esto evita que se creen elementos equivocados.

Con este sistema, el código cliente no depende de clases concretas de fábricas y elementos UI, siempre y cuando trabaje con estos objetos a través de sus interfaces abstractas. Esto también permite que el código cliente soporte otras fábricas o elementos UI que pudiéramos añadir más adelante.

Como consecuencia, no necesitas modificar el código cliente cada vez que añades una nueva variedad de elementos UI a tu aplicación. Tan solo debes crear una nueva clase de fábrica que produzca estos elementos y modifique ligeramente el código de inicialización de la aplicación, de modo que seleccione esa clase cuando resulte apropiado.

```
// La interfaz fábrica abstracta declara un grupo de métodos que
// devuelven distintos productos abstractos. Estos productos se
// denominan familia y están relacionados por un tema o concepto
// de alto nivel. Normalmente, los productos de una familia
// pueden colaborar entre sí. Una familia de productos puede
// tener muchas variantes, pero los productos de una variante
// son incompatibles con los productos de otra.
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox


// Las fábricas concretas producen una familia de productos que
// pertenecen a una única variante. La fábrica garantiza que los
// productos resultantes sean compatibles. Las firmas de los
// métodos de las fábricas concretas devuelven un producto
// abstracto mientras que dentro del método se instancia un
// producto concreto.
class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

// Cada fábrica concreta tiene una variante de producto
// correspondiente.
class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()


// Cada producto individual de una familia de productos debe
// tener una interfaz base. Todas las variantes del producto
// deben implementar esta interfaz.
interface Button is
    method paint()

// Los productos concretos son creados por las fábricas
// concretas correspondientes.
class WinButton implements Button is
    method paint() is
        // Representa un botón en estilo Windows.

class MacButton implements Button is
    method paint() is
        // Representa un botón en estilo macOS.

// Aquí está la interfaz base de otro producto. Todos los
// productos pueden interactuar entre sí, pero sólo entre
// productos de la misma variante concreta es posible una
// interacción adecuada.
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Representa una casilla en estilo Windows.

class MacCheckbox implements Checkbox is
    method paint() is
        // Representa una casilla en estilo macOS.


// El código cliente funciona con fábricas y productos
// únicamente a través de tipos abstractos: GUIFactory, Button y
// Checkbox. Esto te permite pasar cualquier subclase fábrica o
// producto al código cliente sin descomponerlo.
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()


// La aplicación elige el tipo de fábrica dependiendo de la
// configuración actual o de los ajustes del entorno y la crea
// durante el tiempo de ejecución (normalmente en la etapa de
// inicialización).
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")

        Application app = new Application(factory)
```

Aplicabilidad

- ✓ Utiliza el patrón Abstract Factory cuando tu código deba funcionar con varias familias de productos relacionados, pero no desees que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad.
- ✓ El patrón Abstract Factory nos ofrece una interfaz para crear objetos a partir de cada clase de la familia de productos. Mientras tu código cree objetos a través de esta interfaz, no tendrás que preocuparte por crear la variante errónea de un producto que no combine con los productos que ya ha creado tu aplicación.
- ✓ Considera la implementación del patrón Abstract Factory cuando tengas una clase con un grupo de **métodos de fábrica** que nublen su responsabilidad principal.
- ✓ En un programa bien diseñado *cada clase es responsable tan solo de una cosa*. Cuando una clase lidia con varios tipos de productos, puede ser que valga la pena extraer sus métodos de fábrica para ponerlos en una clase única de fábrica o una implementación completa del patrón Abstract Factory.

Implementación

1. Mapea una matriz de distintos tipos de productos frente a variantes de dichos productos.
2. Declara interfaces abstractas de producto para todos los tipos de productos. Después haz que todas las clases concretas de productos implementen esas interfaces.
3. Declara la interfaz de la fábrica abstracta con un grupo de métodos de creación para todos los productos abstractos.
4. Implementa un grupo de clases concretas de fábrica, una por cada variante de producto.
5. Crea un código de inicialización de la fábrica en algún punto de la aplicación. Deberá instanciar una de las clases concretas de la fábrica, dependiendo de la configuración de la aplicación o del entorno actual. Pasa este objeto de fábrica a todas las clases que construyen productos.
6. Explora el código y encuentra todas las llamadas directas a constructores de producto. Sustitúyelas por llamadas al método de creación adecuado dentro del objeto de fábrica.

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.

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).
- **Builder** se enfoca en construir objetos complejos, paso a paso. **Abstract Factory** se especializa en crear familias de objetos relacionados. *Abstract Factory* devuelve el producto inmediatamente, mientras que *Builder* te permite ejecutar algunos pasos adicionales de construcción antes de extraer el producto.
- 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.
- **Abstract Factory** puede servir como alternativa a **Facade** cuando tan solo deseas esconder la forma en que se crean los objetos del subsistema a partir del código cliente.

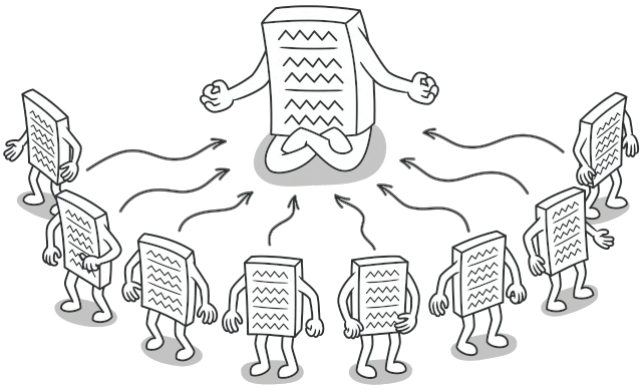
- Puedes utilizar **Abstract Factory** junto a **Bridge**. Este emparejamiento resulta útil cuando algunas abstracciones definidas por *Bridge* sólo pueden funcionar con implementaciones específicas. En este caso, *Abstract Factory* puede encapsular estas relaciones y esconder la complejidad al código cliente.
- Los patrones **Abstract Factory**, **Builder** y **Prototype** pueden todos ellos implementarse como **Singletons**.

Singleton

También llamado: Instancia única

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.



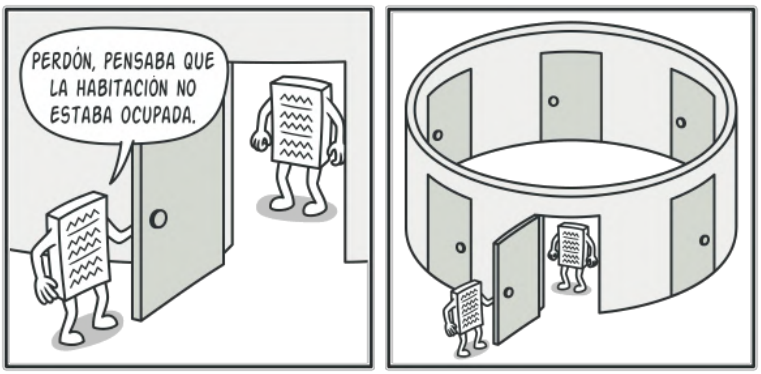
Problema

El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el *Principio de responsabilidad única*:

1. **Garantizar que una clase tenga una única instancia**. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? 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.



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

2. **Proporcionar un punto de acceso global a dicha instancia**. ¿Recuerdas esas variables globales que utilizaste (bueno, sí, fui yo) para almacenar objetos esenciales? Aunque son muy útiles, también son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.

   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 sobreescriba esa instancia.

   Este problema tiene otra cara: no queremos que el código que resuelve el primer problema se encuentre

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.

Hoy en día el patrón Singleton se ha popularizado tanto que la gente suele llamar *singleton* a cualquier patrón, incluso si solo resuelve uno de los problemas antes mencionados.

Solución

Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

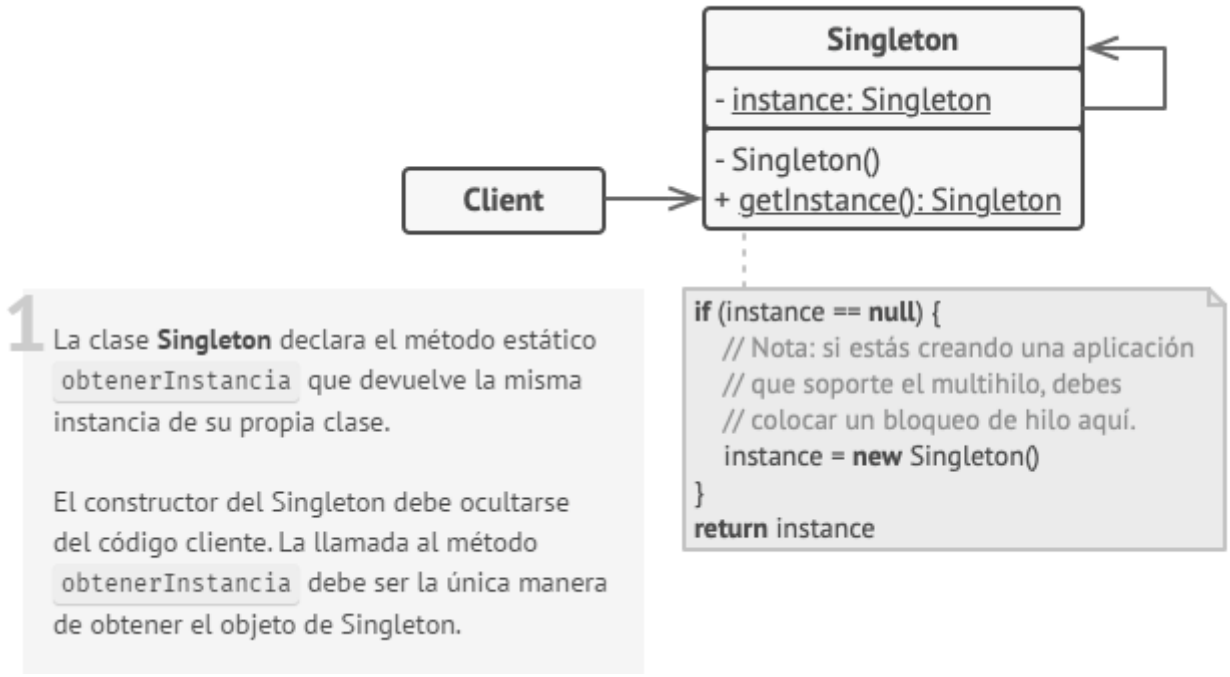
- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador `new` con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

Analogía en el mundo real

El gobierno es un ejemplo excelente del patrón Singleton. Un país sólo puede tener un gobierno oficial. Independientemente de las identidades personales de los individuos que forman el gobierno, el título “Gobierno de X” es un punto de acceso global que identifica al grupo de personas a cargo.

Estructura



Pseudocódigo

En este ejemplo, la clase de conexión de la base de datos actúa como **Singleton**. Esta clase no tiene un constructor público, por lo que la única manera de obtener su objeto es invocando el método `obtenerInstancia`. Este método almacena en caché el primer objeto creado y lo devuelve en todas las llamadas siguientes.

```
// La clase Base de datos define el método `obtenerInstancia`
// que permite a los clientes acceder a la misma instancia de
// una conexión de la base de datos a través del programa.
class Database is
    // El campo para almacenar la instancia singleton debe
    // declararse estático.
    private static field instance: Database

    // El constructor del singleton siempre debe ser privado
    // para evitar llamadas de construcción directas con el
    // operador `new`.
    private constructor Database() is
        // Algún código de inicialización, como la propia
        // conexión al servidor de una base de datos.
        // ...


// El método estático que controla el acceso a la instancia
    // singleton.
    public static method getInstance() is
        if (Database.instance == null) then
            acquireThreadLock() and then
                // Garantiza que la instancia aún no se ha

// inicializado por otro hilo mientras ésta ha
                // estado esperando el desbloqueo.
                if (Database.instance == null) then
                    Database.instance = new Database()
        return Database.instance

    // Por último, cualquier singleton debe definir cierta

// lógica de negocio que pueda ejecutarse en su instancia.
    public method query(sql) is

// Por ejemplo, todas las consultas a la base de datos
        // de una aplicación pasan por este método. Por lo

// tanto, aquí puedes colocar lógica de regularización
        // (throttling) o de envío a la memoria caché.
        // ...

class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // La variable `bar` contendrá el mismo objeto que la
        // variable `foo`.
```

Aplicabilidad

- ✓ 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`.

Implementación

1. Añade un campo estático privado a la clase para almacenar la instancia Singleton.
2. Declara un método de creación estático público para obtener la instancia Singleton.
3. Implementa una inicialización diferida dentro del método estático. Debe crear un nuevo objeto en su primera llamada y colocarlo dentro del campo estático. El método deberá devolver siempre esa instancia en todas las llamadas siguientes.
4. Declara el constructor de clase como privado. El método estático de la clase seguirá siendo capaz de invocar al constructor, pero no a los otros objetos.
5. Repasa el código cliente y sustituye todas las llamadas directas al constructor de la instancia Singleton por llamadas a su método de creación estático.

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.

Relaciones con otros patrones

- Una clase **fachada** 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:
- Solo debe haber una instancia Singleton, mientras que una clase *Flyweight* puede tener varias instancias con distintos estados intrínsecos.
- El objeto *Singleton* puede ser mutable. Los objetos flyweight son inmutables.
- Los patrones **Abstract Factory**, **Builder** y **Prototype** pueden todos ellos implementarse como **Singletons**.