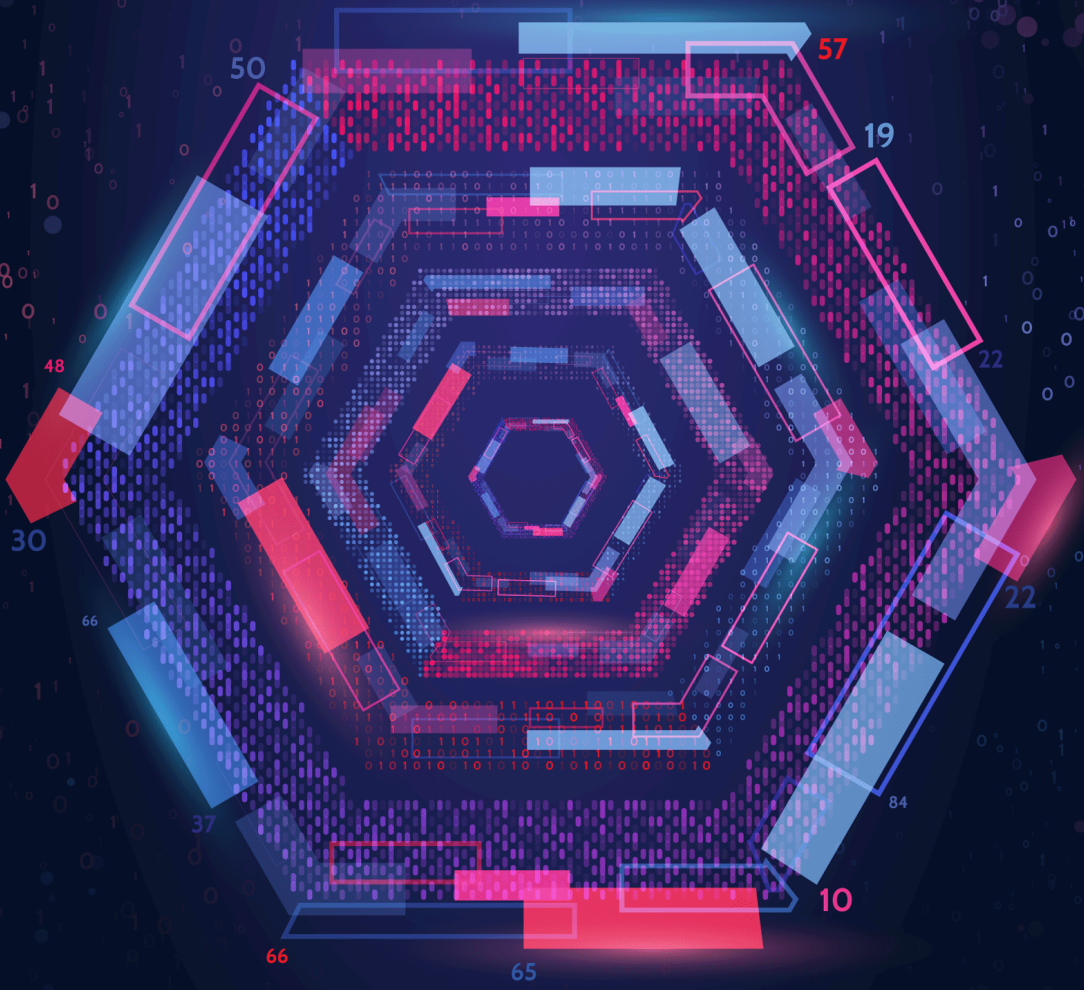
Sumérgete en los PATRONES DE DISEÑO



Alexander Shvets

Sumérgete en los PATRONES DE DISEÑO

v2021-1.7

VERSIÓN DE PRUEBA

Unas palabras sobre derechos de autor

¡Hola! Mi nombre es Alexander Shvets. Soy el autor del libro **Sumérgete en los patrones de diseño** y del curso online **Dive Into Refactoring**.

Este libro es para tu uso personal. Por favor, no lo compartas con terceras personas, a excepción de los miembros de tu familia. Si deseas compartir el libro con un amigo o colega, compra y envíale una nueva copia. También puede comprar una licencia para todo su equipo o para toda la empresa.

Todas las ganancias de las ventas de mis libros y cursos se dedican al desarrollo de **Refactoring.Guru**. Cada copia vendida ayuda inmensamente al proyecto y acerca un poco más el momento del lanzamiento de un nuevo libro.

support@refactoring.guru

Ilustraciones: Dmitry Zhart
Traducción: Álvaro Montero
Edición: Jorge F. Ramírez Ariza

Dedico este libro a mi esposa, Maria. Si no hubiese sido por ella, lo habría terminado unos 30 años más tarde.

Índice de contenido

Cómo leer este libro

Este libro contiene las descripciones de 22 patrones de diseño clásicos formulados por la “banda de los cuatro” (o simplemente GoF, por sus siglas en inglés) en 1994.

Cada capítulo explora un patrón particular. Por lo tanto, puedes leerlo de principio a fin, o ir eligiendo los patrones que te interesan.

Muchos patrones están relacionados, por lo que puedes saltar fácilmente de un tema a otro utilizando varios puntos de enlace. Al final de cada capítulo hay una lista de enlaces entre el patrón actual y otros. Si ves el nombre de un patrón que aún no habías visto, sigue leyendo, este patrón aparecerá en alguno de los capítulos siguientes.

Los patrones de diseño son universales. Por ello, todos los ejemplos de código de este libro están escritos en un pseudocódigo que no restringe el material a un lenguaje de programación particular.

Antes de estudiar los patrones, puedes refrescar tu memoria repasando los **términos clave de la programación orientada a objetos**. En ese capítulo también se explican los fundamentos de los diagramas UML (lenguaje unificado de modelado), lo que resulta de utilidad porque hay un montón de ellos en el libro. Por supuesto, si ya sabes todo esto, puedes proceder directamente a **aprender sobre patrones**.

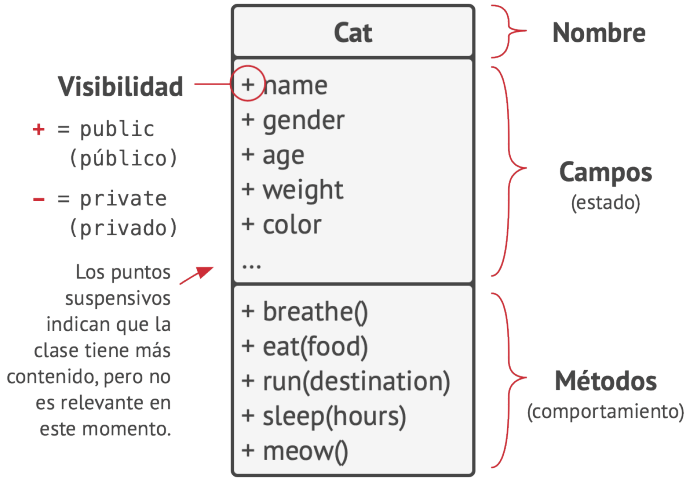
FUNDAMENTOS DE LA POO

Conceptos básicos de POO

La ***P**rogramación **o**rientada a **o**bjetos* (POO) es un paradigma basado en el concepto de envolver bloques de información y su comportamiento relacionado, en lotes especiales llamados **objetos**, que se construyen a partir de un grupo de "planos" definidos por un programador, que se denominan **clases**.

Objetos, clases

¿Te gustan los gatos? Espero que sí, porque voy a intentar explicar los conceptos de POO utilizando varios ejemplos con gatos.



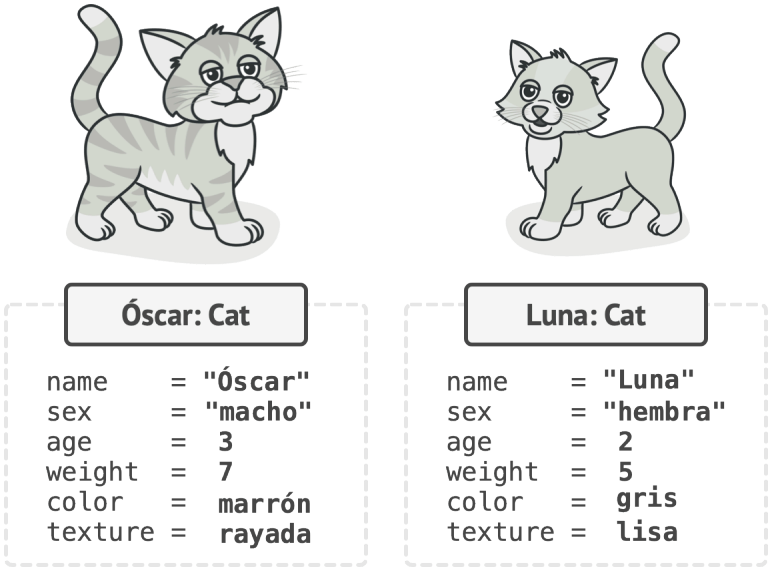
Esto es un diagrama de clases en UML. Encontrarás muchos diagramas como éste en el libro. Los nombres de las cosas en los diagramas están en inglés, como lo estarían en un código real. Sin embargo, los comentarios y las notas pueden estar en español.

Digamos que tienes un gato llamado Óscar. Óscar es un objeto, una instancia de la clase `Gato`. Cada gato tiene varios atributos estándar: nombre, sexo, edad, peso, color, comida favorita, etc. Estos son los *campos* de la clase.

> En este libro puedo referirme a los nombres de las clases en español, aunque aparezcan en diagramas o en código en inglés (como hice con la clase `Gato`). Quiero que leas el libro como si tuviéramos una conversación hablada entre amigos. No quiero que te topes con palabras extrañas cada vez que tenga que hacer referencia a alguna clase.

Además, todos los gatos se comportan de forma similar: respiran, comen, corren, duermen y maúllan. Estos son los *métodos* de la clase. Colectivamente, podemos referirnos a los campos y los métodos como los *miembros* de su clase.

> La información almacenada dentro de los campos del objeto suele denominarse *estado*, y todos los métodos del objeto definen su *comportamiento*.



Los objetos son instancias de clases.

Luna, la gata de tu amigo, también es una instancia de la clase `Gato`. Tiene el mismo grupo de atributos que Óscar. La diferencia está en los valores de estos atributos: su sexo es hembra, tiene un color diferente y pesa menos.

Por lo tanto, una *clase* es como un plano que define la estructura de los *objetos*, que son instancias concretas de esa clase.

Jerarquías de clase

Todo va muy bien mientras hablamos de una sola clase. Naturalmente, un programa real contiene más de una clase. Algunas de esas clases pueden estar organizadas en **jerarquías de clase**. Veamos lo que esto significa.

Digamos que tu vecino tiene un perro llamado Fido. Resulta que perros y gatos tienen mucho en común: nombre, sexo, edad y color, son atributos tanto de perros como de gatos. Los perros pueden respirar, dormir y correr igual que los gatos, por lo que podemos definir la clase base `Animal` que enumerará los atributos y comportamientos comunes.

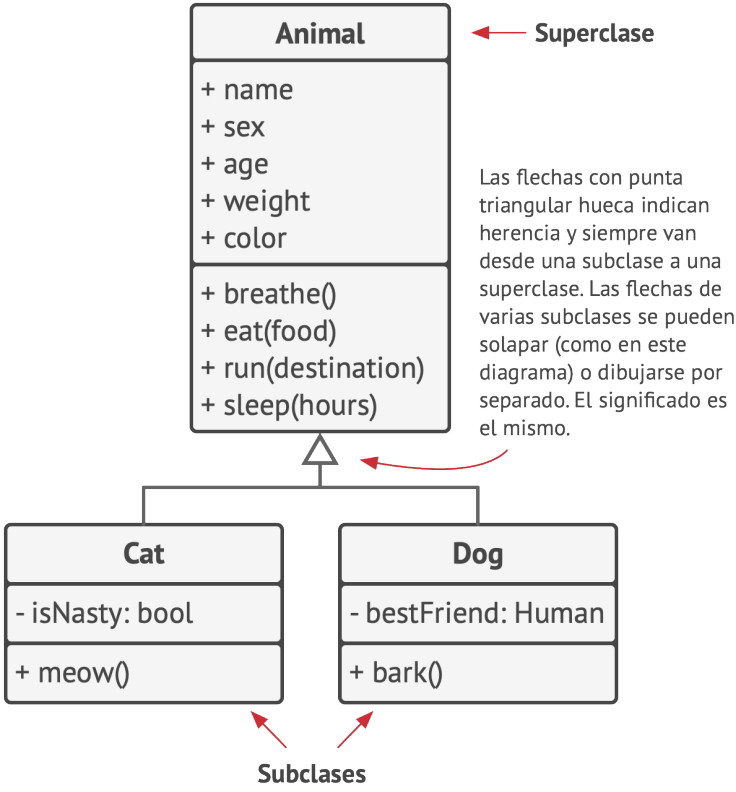
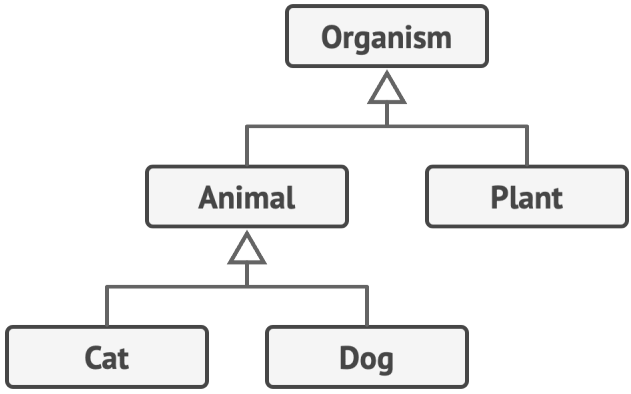


Diagrama UML de una jerarquía de clases. Todas las clases de este diagrama son parte de la jerarquía de clases `Animal`.

Una clase padre, como la que acabamos de definir, se denomina **superclase**. Sus hijas son las **subclases**. Las subclases he-

redan el estado y el comportamiento de su padre y se limitan a definir atributos o comportamientos que son diferentes. Por lo tanto, la clase `Gato` contendrá el método `maullar` y, la clase `Perro`, el método `ladrar`.

Asumiendo que tenemos una tarea relacionada, podemos ir más lejos y extraer una clase más genérica para todos los `Organismo` vivos, que se convertirá en una superclase para `Animal` y `Planta`. Tal pirámide de clases es una **jerarquía**. En esta jerarquía, la clase `Gato` lo hereda todo de las clases `Animal` y `Organismo`.



En un diagrama UML las clases se pueden simplificar si es más importante mostrar sus relaciones que sus contenidos.

Las subclases pueden sobrescribir el comportamiento de los métodos que heredan de clases padre. Una subclase puede sustituir completamente el comportamiento por defecto o limitarse a mejorarlo con material adicional.

PRINCIPIOS DE DISEÑO DE SOFTWARE

Características del buen diseño

Antes de proceder con los patrones propiamente dichos, discutamos el proceso del diseño de arquitectura de software: cosas que buscar y cosas que evitar.

Reutilización de código

Costos y tiempo son dos de los parámetros más valiosos a la hora de desarrollar cualquier producto de software. Dedicar menos tiempo al desarrollo se traduce en entrar en el mercado antes que la competencia. Unos costos más bajos en el desarrollo significa que habrá más dinero disponible para marketing y un alcance más amplio a clientes potenciales.

La **reutilización de código** es una de las formas más habituales de reducir costos de desarrollo. El propósito es obvio: en lugar de desarrollar algo una y otra vez desde el principio, ¿por qué no reutilizar el código existente en nuevos proyectos?

La idea se ve bien sobre el papel, pero resulta que hacer que el código existente funcione en un nuevo contexto, a menudo requiere de un esfuerzo adicional. Acoplamientos fuertes entre componentes, dependencias de clases concretas en lugar de interfaces, operaciones incrustadas en el código... Todo esto reduce la flexibilidad del código y hace que sea más difícil de reutilizar.

Utilizar patrones de diseño es una de las maneras de aumentar la flexibilidad de los componentes de software y hacerlos más fáciles de reutilizar. Sin embargo, en ocasiones esto tiene el precio de complicar los componentes.

Aquí tienes un apunte de sabiduría de Erich Gamma [1], uno de los padres fundadores de los patrones de diseño, sobre el papel que estos juegan en la reutilización de código:

> Yo veo tres niveles de reutilización.
>
> En el nivel más bajo, reutilizas clases: bibliotecas de clase, contenedores, quizá algunos "equipos" de clases, como contenedor/iterador.
>
> Los *frameworks* se encuentran en el nivel superior. Intentan destilar tus decisiones de diseño. Identifican las abstracciones clave para resolver un problema, las representan con clases y definen relaciones entre ellas. JUnit es un pequeño *framework*, por ejemplo. Es el "¡Hola, mundo!" de los *frameworks*. Tiene `Test`, `TestCase`, `TestSuite` y las relaciones definidas.
>
> Normalmente, un *framework* es más tosco que una única clase. Además, te enganchas a los *frameworks* creando subclases en alguna parte. Utilizan el llamado principio de Hollywood de "no nos llames, nosotros te llamamos a ti". El *framework* te permite definir tu comportamiento personalizado y ya te llamará cuando sea tu turno de hacer algo. Lo mismo sucede con JUnit,

1. Erich Gamma sobre flexibilidad y reutilización: **https://refactoring.guru/gamma-interview**

> ¿verdad? Te llama cuando quiere ejecutar una prueba para ti, pero el resto sucede en el *framework*.
>
> También hay un nivel intermedio. Aquí es donde veo patrones. Los patrones de diseño son más pequeños y más abstractos que los *frameworks*. En realidad, son una descripción sobre cómo pueden relacionarse un par de clases e interactuar entre sí. El nivel de reutilización aumenta cuando pasas de clases a patrones y por último a *frameworks*.
>
> Lo bueno de esta capa intermedia es que, a menudo, los patrones ofrecen la reutilización de un modo menos arriesgado que los *frameworks*. Crear un *framework* comprende un alto riesgo y una inversión considerable. Los patrones te permiten reutilizar ideas y conceptos de diseño con independencia del código concreto.
>
> ”

Extensibilidad

El **cambio** es lo único constante en la vida de un programador.

- Lanzaste un videojuego para Windows, pero ahora la gente demanda una versión macOS.
- Creaste un *framework* GUI con botones cuadrados, pero meses después los botones redondos son tendencia.
- Diseñaste una espectacular arquitectura para un sitio web de comercio electrónico, pero poco después los clientes piden una función que les permita aceptar pedidos por teléfono.

Cada desarrollador de software puede contar decenas de historias similares. Hay muchas razones para esto.

En primer lugar, comprendemos mejor el problema una vez que comenzamos a resolverlo. A menudo, para cuando finalizamos la primera versión de una aplicación, estamos listos para reescribirla desde el principio porque entonces comprendemos mucho mejor diversos aspectos del problema. También creciste profesionalmente y tu propio código te parece basura.

Algo fuera de tu control ha cambiado. Éste es el motivo por el que tantos equipos de desarrollo saltan de sus ideas originales hacia algo nuevo. Todos aquellos que se basaron en Flash para una aplicación online han rehecho o migrado su código desde que, navegador tras navegador, todos dejan de soportar Flash.

La tercera razón es que los postes de la portería se mueven. Tu cliente quedó encantado con la versión actual de la aplicación, pero ahora quiere once "pequeños" cambios para que haga otras cosas que nunca te mencionó en las primeras sesiones de planificación. No se trata de cambios frívolos: tu excelente primera versión le ha demostrado que todavía es posible mucho más. Hay un lado positivo: si alguien te pide cambiar algo de tu aplicación, eso significa que todavía le importa a alguien.

Por este motivo, todos los desarrolladores experimentados se preparan para posibles cambios futuros cuando diseñan la arquitectura de una aplicación.

Principios del diseño

¿Qué es un buen diseño de software? ¿Cómo medimos su calidad? ¿Qué prácticas debemos llevar a cabo para lograrlo? ¿Cómo podemos hacer nuestra arquitectura flexible, estable y fácil de comprender?

Éstas son las grandes preguntas, pero, lamentablemente, las respuestas varían en función del tipo de aplicación que estés creando. No obstante, existen varios principios universales del diseño de software que pueden ayudarte a responder estas preguntas para tu propio proyecto. La mayoría de los patrones de diseño tratados en este libro se basa en estos principios.

Encapsula lo que varía

> Identifica los aspectos de tu aplicación que varían y sepáralos de los que se mantienen inalterables.

El objetivo principal de este principio es minimizar el efecto provocado por los cambios.

Imagina que tu programa es un barco y los cambios son horribles minas escondidas bajo el agua. Si el barco golpea una mina, se hunde.

Sabiendo esto, puedes dividir el casco del barco en compartimentos individuales que puedan sellarse para limitar los daños a un único compartimento. De esto modo, si el barco golpea una mina, puede permanecer a flote.

Del mismo modo, puedes aislar las partes del programa que varían, en módulos independientes, protegiendo el resto del código frente a efectos adversos. Al hacerlo, dedicarás menos tiempo a lograr que el programa vuelva a funcionar, implementando y probando los cambios. Cuanto menos tiempo dediques a realizar cambios, más tiempo tendrás para implementar funciones.

Encapsulación a nivel del método

Digamos que estás creando un sitio web de comercio electrónico. En alguna parte de tu código, hay un método `obtenerTotaldelPedido` que calcula un total del pedido, impuestos incluidos.

Podemos anticipar que el código relacionado con los impuestos tendrá que cambiar en el futuro. La tasa impositiva dependerá de cada país, estado, o incluso ciudad en la que resida el cliente, y la fórmula puede variar a lo largo del tiempo con base a nuevas leyes o regulaciones. Esto hará que tengas que cambiar el método `obtenerTotaldelPedido` bastante a menudo. Pero incluso el nombre del método sugiere que no le importa *cómo* se calcula el impuesto.

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  if (order.country == "US")
    total += total * 0.07 // Impuesto sobre la venta de EUA
  else if (order.country == "EU"):
    total += total * 0.20 // IVA europeo

  return total
```

ANTES: el código de cálculo del impuesto está mezclado con el resto del código del método.

Puedes extraer la lógica de cálculo del impuesto a un método separado, escondiéndolo del método original.

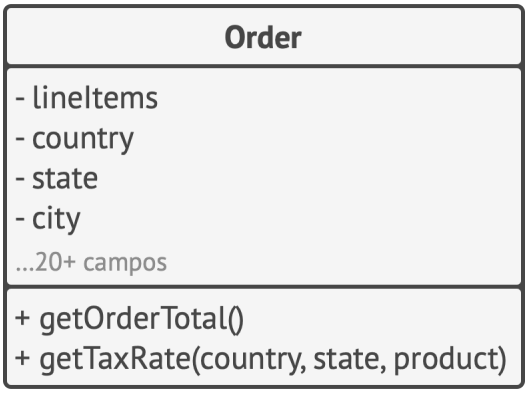
```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  total += total * getTaxRate(order.country)

  return total

method getTaxRate(country) is
  if (country == "US")
    return 0.07 // Impuesto sobre la venta de EUA
  else if (country == "EU")
    return 0.20 // IVA europeo
  else
    return 0
```

DESPUÉS: puedes obtener la tasa impositiva invocando un método designado.

Los cambios relacionados con el impuesto quedan aislados dentro de un único método. Además, si la lógica de cálculo del impuesto se complica demasiado, ahora es más sencillo moverla a una clase separada.

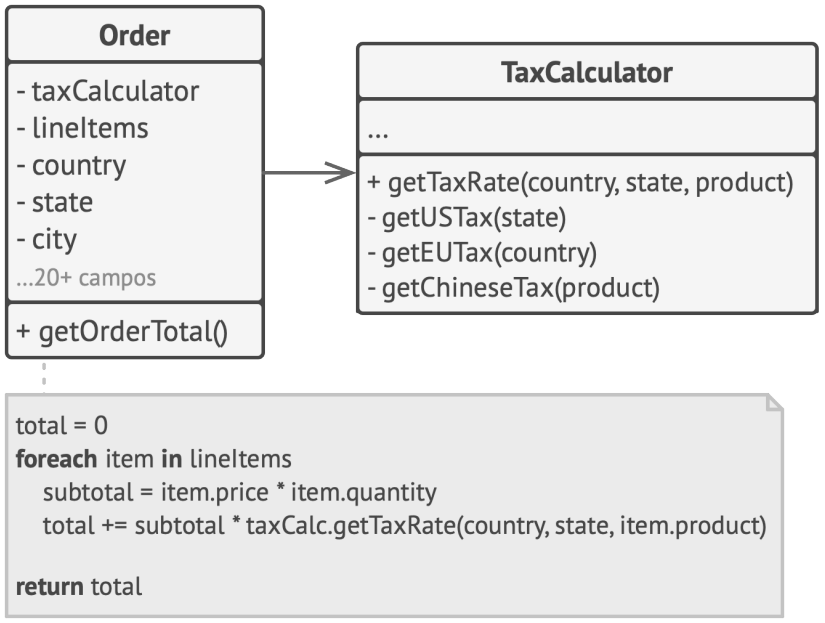
Encapsulación a nivel de la clase

Con el tiempo puedes añadir más y más responsabilidades a un método que solía hacer algo sencillo. Estos comportamientos añadidos suelen venir con sus propios campos y métodos de ayuda que acaban nublando la responsabilidad principal de la clase contenedora. Si se extrae todo a una nueva clase se puede conseguir mayor claridad y sencillez.



ANTES: cálculo del impuesto en la clase `Pedido`.

Los objetos de la clase `Pedido` delegan todo el trabajo relacionado con el impuesto a un objeto especial dedicado justo a eso.

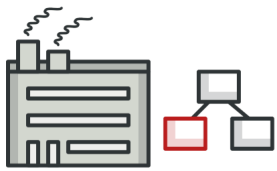


DESPUÉS: el cálculo del impuesto se esconde de la clase `Pedido`.

EL CATÁLOGO DE PATRONES DE DISEÑO

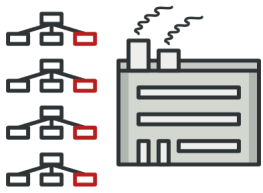
Patrones creacionales

Los patrones creacionales proporcionan varios mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente.



Factory Method

Proporciona una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

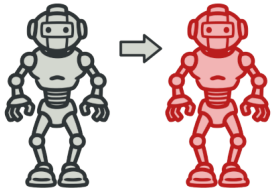


Abstract Factory

Permite producir familias de objetos relacionados sin especificar sus clases concretas.

Builder

Permite construir objetos complejos paso a paso. Este patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

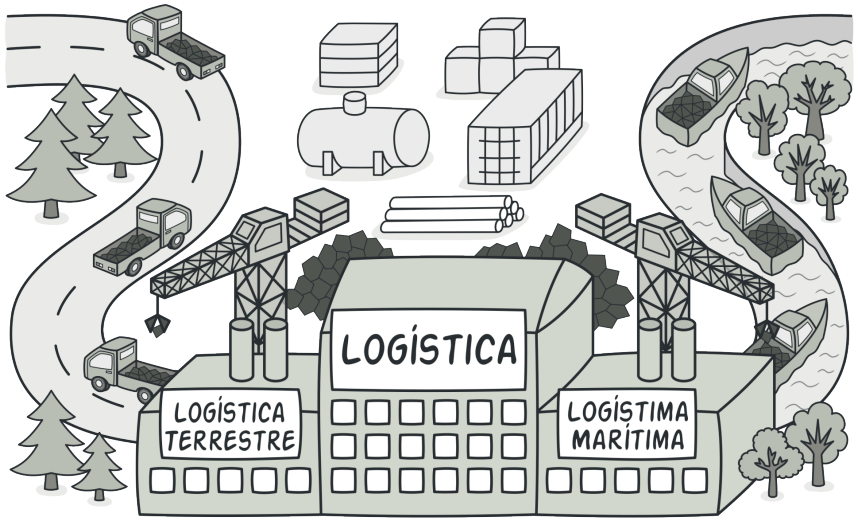


Prototype

Permite copiar objetos existentes sin que el código dependa de sus clases.

Singleton

Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.



FACTORY METHOD

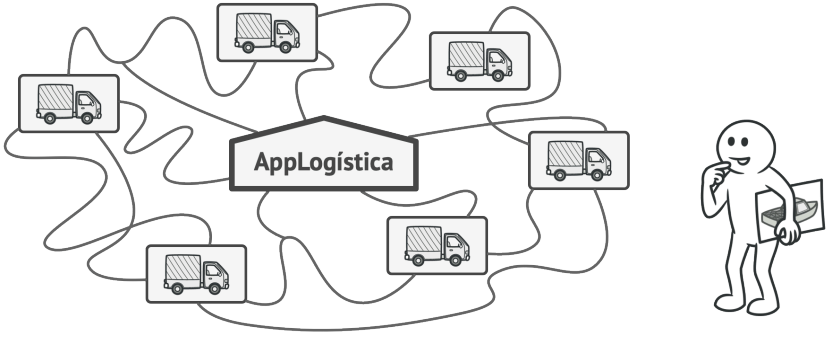
También llamado: Método fábrica, Constructor virtual

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.



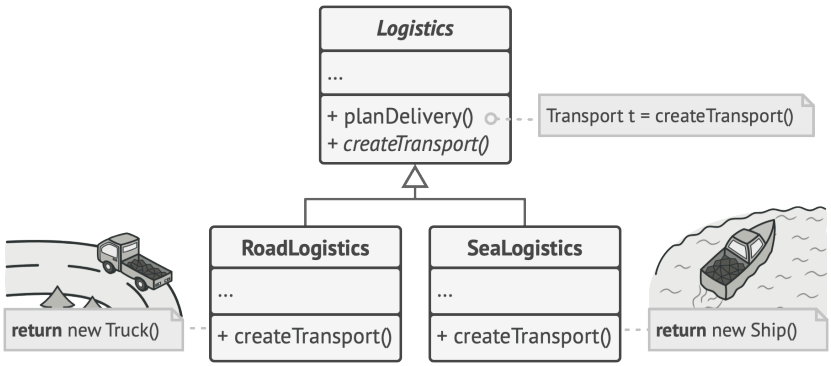
Añadir una nueva clase al programa no es tan sencillo si el resto del código ya está acoplado a clases existentes.

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

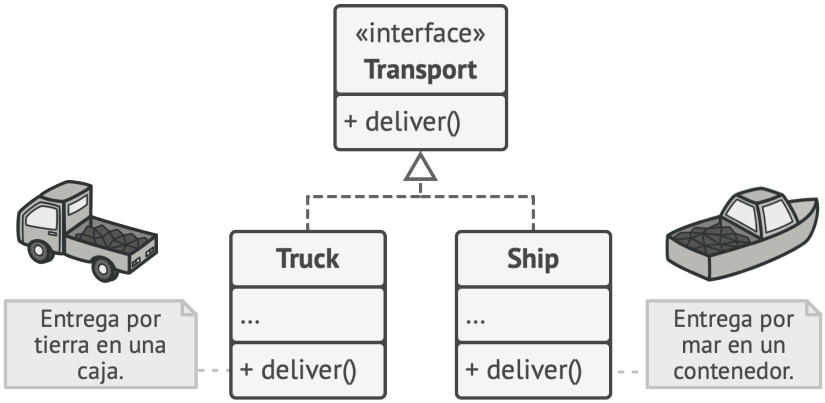
El patrón Factory Method sugiere que, en lugar de llamar al operador `new` para construir objetos directamente, se invoque a un método *fábrica* especial. No te preocupes: los objetos se siguen creando a través del operador `new`, pero se invocan desde el método fábrica. Los objetos devueltos por el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



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

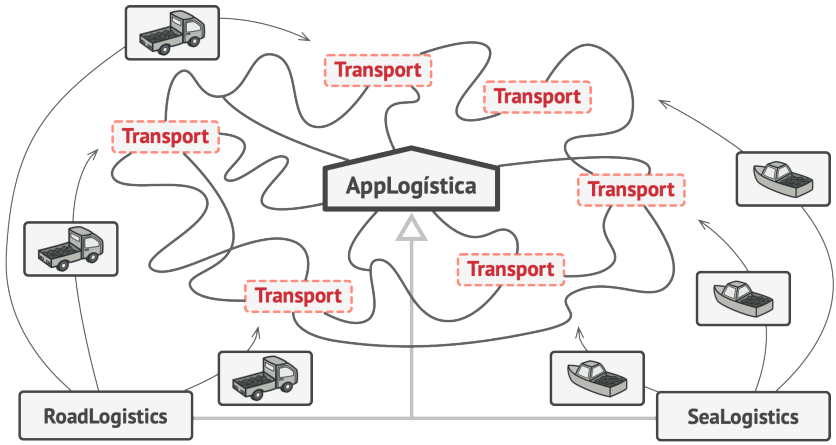
A simple vista, puede parecer que este cambio no tiene sentido, ya que tan solo hemos cambiado el lugar desde donde invocamos al constructor. Sin embargo, piensa en esto: ahora puedes sobrescribir el método fábrica en una subclase y cambiar la clase de los productos creados por el método.

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.

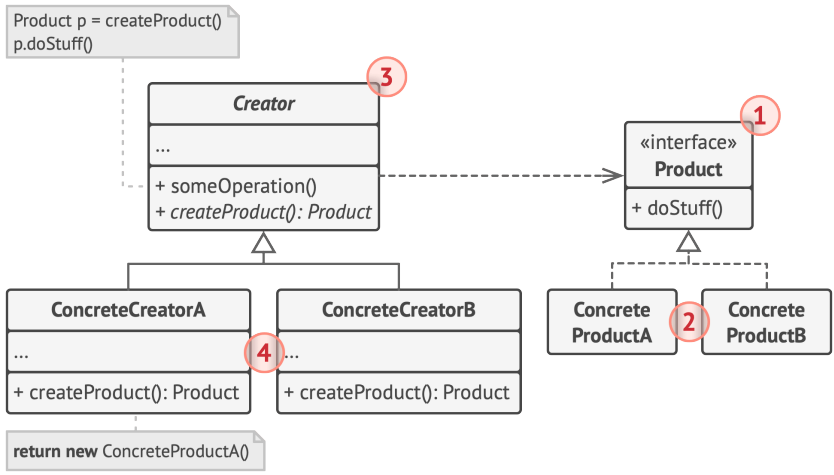
Por ejemplo, tanto la clase `Camión` como la clase `Barco` deben implementar la interfaz `Transporte`, que declara un método llamado `entrega`. Cada clase implementa este método de forma diferente: los camiones entregan su carga por tierra, mientras que los barcos lo hacen por mar. El método fábrica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



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.

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.

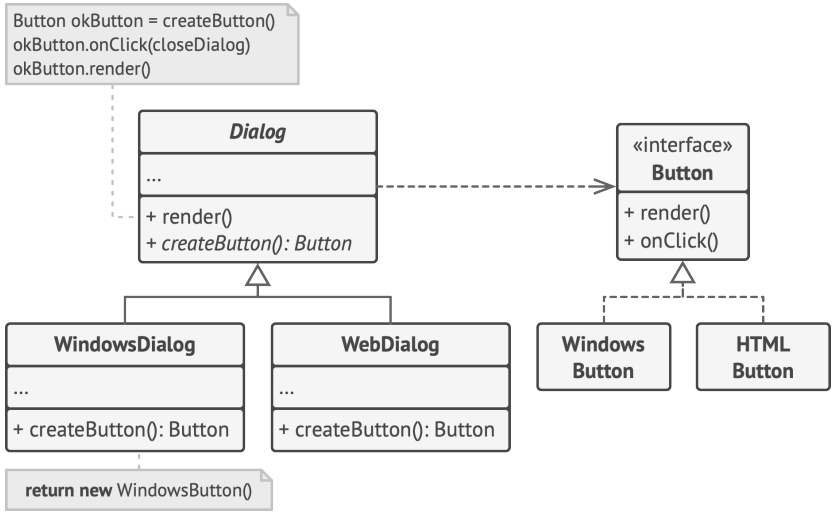
Observa que, a pesar de su nombre, la creación de producto **no** es la principal responsabilidad de la clase creadora. Normalmente, esta clase cuenta con alguna lógica de negocios central relacionada con los productos. El patrón Factory Method ayuda a desacoplar esta lógica de las clases concretas de 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.

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.

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.

Cómo implementarlo

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*.

342 páginas

del libro completo se omiten en la versión de prueba