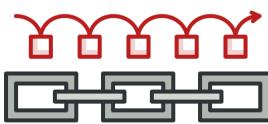


Patrones de comportamiento

Los patrones de comportamiento tratan con algoritmos y la asignación de responsabilidades entre objetos.



Chain of Responsibility

Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.



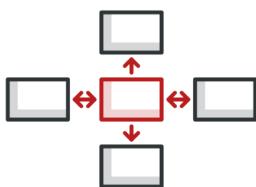
Command

Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.



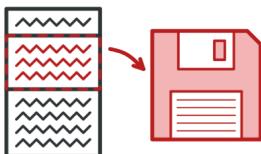
Iterator

Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).



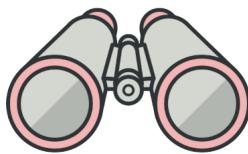
Mediator

Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.



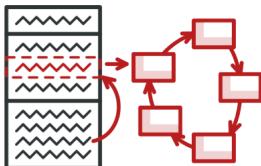
Memento

Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.



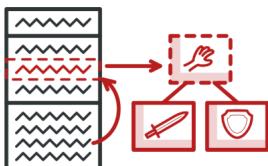
Observer

Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



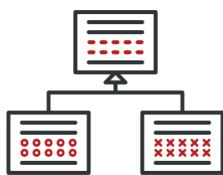
State

Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.



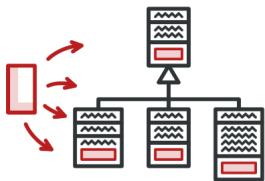
Strategy

Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.



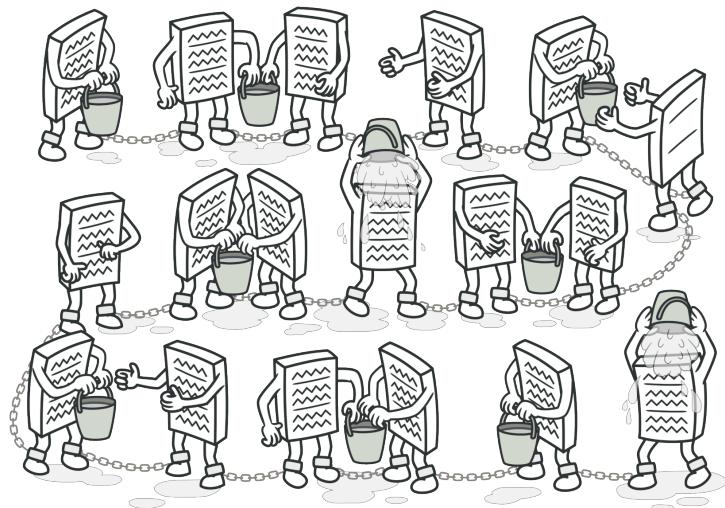
Template Method

Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.



Visitor

Permite separar algoritmos de los objetos sobre los que operan.



CHAIN OF RESPONSIBILITY

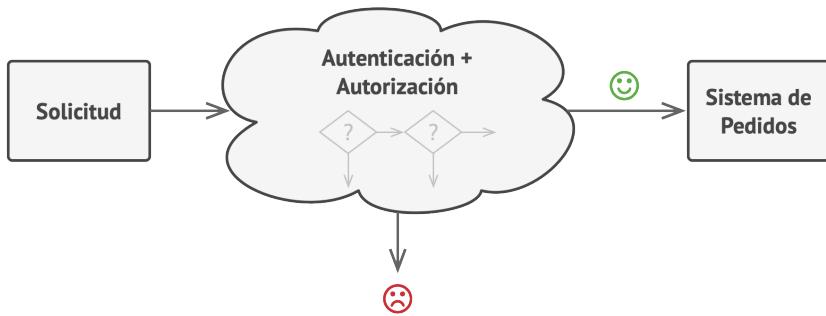
También llamado: *Cadena de responsabilidad, CoR, Chain of Command*

Chain of Responsibility es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

Problema

Imagina que estás trabajando en un sistema de pedidos online. Quieres restringir el acceso al sistema de forma que únicamente los usuarios autenticados puedan generar pedidos. Además, los usuarios que tengan permisos administrativos deben tener pleno acceso a todos los pedidos.

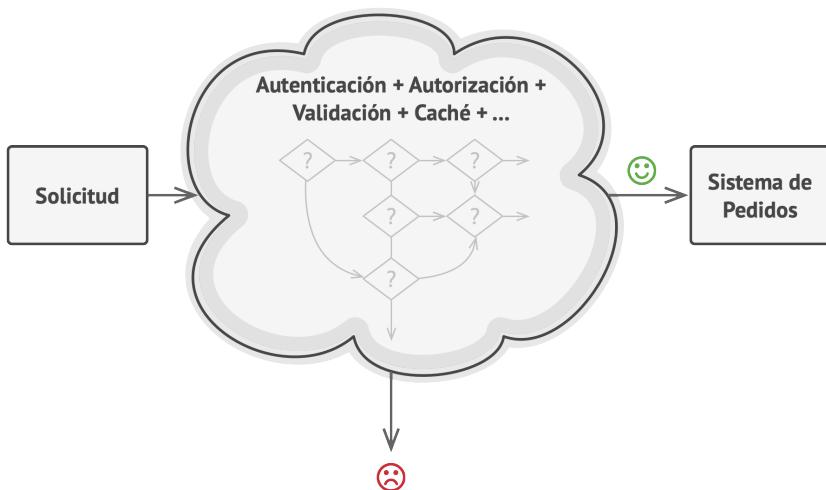
Tras planificar un poco, te das cuenta de que estas comprobaciones deben realizarse secuencialmente. La aplicación puede intentar autenticar a un usuario en el sistema cuando reciba una solicitud que contenga las credenciales del usuario. Sin embargo, si esas credenciales no son correctas y la autenticación falla, no hay razón para proceder con otras comprobaciones.



La solicitud debe pasar una serie de comprobaciones antes de que el propio sistema de pedidos pueda gestionarla.

Durante los meses siguientes, implementas varias de esas comprobaciones secuenciales.

- Uno de tus colegas sugiere que no es seguro pasar datos sin procesar directamente al sistema de pedidos. De modo que añades un paso adicional de validación para sanear los datos de una solicitud.
- Más tarde, alguien se da cuenta de que el sistema es vulnerable al desciframiento de contraseñas por la fuerza. Para evitarlo, añades rápidamente una comprobación que filtra las solicitudes fallidas repetidas que vengan de la misma dirección IP.
- Otra persona sugiere que podrías acelerar el sistema devolviendo los resultados en caché en solicitudes repetidas que contengan los mismos datos, de modo que añades otra comprobación que permite a la solicitud pasar por el sistema únicamente cuando no hay una respuesta adecuada en caché.



Cuanto más crece el código, más se complica.

El código de las comprobaciones, que ya se veía desordenado, se vuelve más y más abotargado cada vez que añades una nueva función. En ocasiones, un cambio en una comprobación afecta a las demás. Y lo peor de todo es que, cuando intentas reutilizar las comprobaciones para proteger otros componentes del sistema, tienes que duplicar parte del código, ya que esos componentes necesitan parte de las comprobaciones, pero no todas ellas.

El sistema se vuelve muy difícil de comprender y costoso de mantener. Luchas con el código durante un tiempo hasta que un día decides refactorizarlo todo.

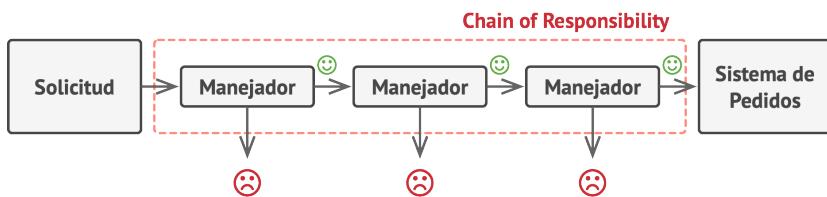
Solución

Al igual que muchos otros patrones de diseño de comportamiento, el **Chain of Responsibility** se basa en transformar comportamientos particulares en objetos autónomos llamados *manejadores*. En nuestro caso, cada comprobación debe ponerse dentro de su propia clase con un único método que realice la comprobación. La solicitud, junto con su información, se pasa a este método como argumento.

El patrón sugiere que vincules esos manejadores en una cadena. Cada manejador vinculado tiene un campo para almacenar una referencia al siguiente manejador de la cadena. Además de procesar una solicitud, los manejadores la pasan a lo largo de la cadena. La solicitud viaja por la cadena hasta que todos los manejadores han tenido la oportunidad de procesarla.

Y ésta es la mejor parte: un manejador puede decidir no pasar la solicitud más allá por la cadena y detener con ello el procesamiento.

En nuestro ejemplo de los sistemas de pedidos, un manejador realiza el procesamiento y después decide si pasa la solicitud al siguiente eslabón de la cadena. Asumiendo que la solicitud contiene la información correcta, todos los manejadores pueden ejecutar su comportamiento principal, ya sean comprobaciones de autenticación o almacenamiento en la memoria caché.

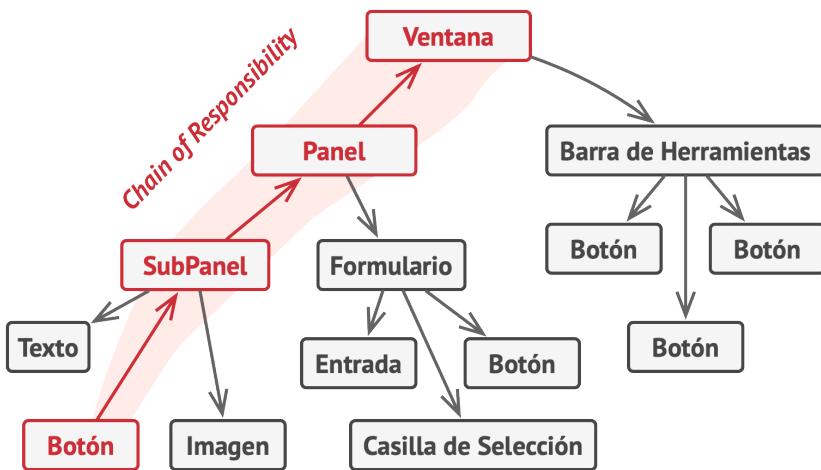


Los manejadores se alinean uno tras otro, formando una cadena.

No obstante, hay una solución ligeramente diferente (y un poco más estandarizada) en la que, al recibir una solicitud, un manejador decide si puede procesarla. Si puede, no pasa la solicitud más allá. De modo que un único manejador procesa la solicitud o no lo hace ninguno en absoluto. Esta solución es muy habitual cuando tratamos con eventos en pilas de elementos dentro de una interfaz gráfica de usuario (GUI).

Por ejemplo, cuando un usuario hace clic en un botón, el evento se propaga por la cadena de elementos GUI que comienza en el botón, recorre sus contenedores (como formularios o pa-

neles) y acaba en la ventana principal de la aplicación. El evento es procesado por el primer elemento de la cadena que es capaz de gestionarlo. Este ejemplo también es destacable porque muestra que siempre se puede extraer una cadena de un árbol de objetos.



Una cadena puede formarse a partir de una rama de un árbol de objetos.

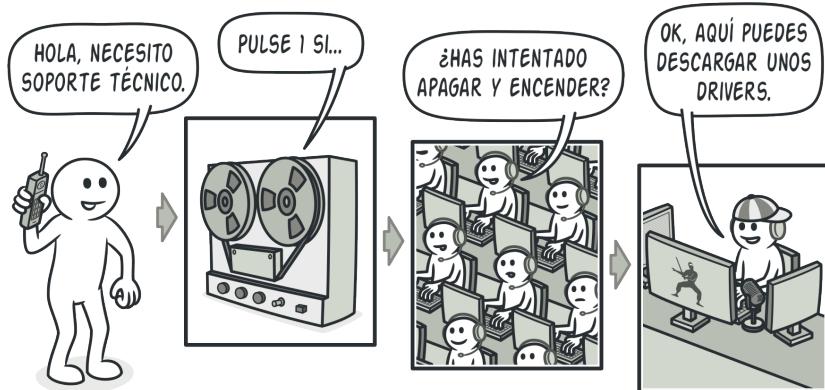
Es fundamental que todas las clases manejadoras implementen la misma interfaz. Cada manejadora concreta solo debe preocuparse por lo siguiente que cuente con el método `ejecutar`. De esta forma puedes componer cadenas durante el tiempo de ejecución, utilizando varios manejadores sin acoplar tu código a sus clases concretas.

🚗 Analogía en el mundo real

Acabas de comprar e instalar una nueva pieza de hardware en tu computadora. Como eres un fanático de la informática, la

computadora tiene varios sistemas operativos instalados. Intentas arrancarlos todos para ver si soportan el hardware. Windows detecta y habilita el hardware automáticamente. Sin embargo, tu querido Linux se niega a funcionar con el nuevo hardware. Ligeramente esperanzado, decides llamar al número de teléfono de soporte técnico escrito en la caja.

Lo primero que oyes es la voz robótica del contestador automático. Te sugiere nueve soluciones populares a varios problemas, pero ninguna de ellas es relevante a tu caso. Después de un rato, el robot te conecta con un operador humano.

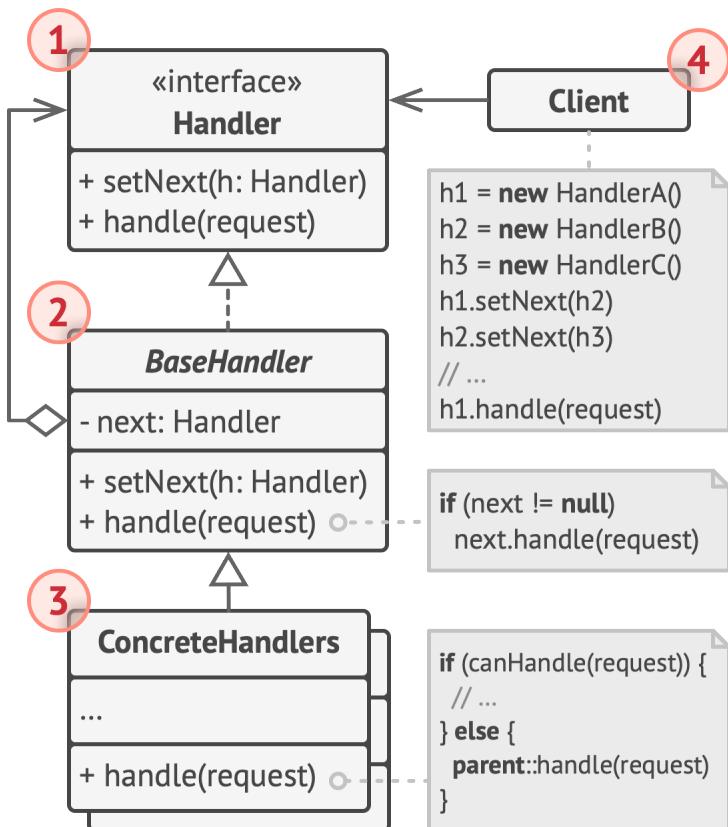


Una llamada al soporte técnico puede pasar por muchos operadores.

Por desgracia, el operador tampoco consigue sugerirte nada específico. Se dedica a recitar largos pasajes del manual, negándose a escuchar tus comentarios. Cuando escuchas por enésima vez la frase “¿has intentado apagar y encender la computadora?”, exiges que te pasen con un ingeniero de verdad.

Por fin, el operador pasa tu llamada a unos de los ingenieros, que probablemente ansiaba una conversación humana desde hacía tiempo, sentado en la solitaria sala del servidor del oscuro sótano de un edificio de oficinas. El ingeniero te indica dónde descargar los drivers adecuados para tu nuevo hardware y cómo instalarlos en Linux. Por fin, ¡la solución! Acabas la llamada dando saltos de alegría.

Estructura



1. La clase **Manejadora** declara la interfaz común a todos los manejadores concretos. Normalmente contiene un único método para manejar solicitudes, pero en ocasiones también puede contar con otro método para establecer el siguiente manejador de la cadena.
2. La clase **Manejadora Base** es opcional y es donde puedes colocar el código boilerplate (segmentos de código que suelen no alterarse) común para todas las clases manejadoras.

Normalmente, esta clase define un campo para almacenar una referencia al siguiente manejador. Los clientes pueden crear una cadena pasando un manejador al constructor o modificador (*setter*) del manejador previo. La clase también puede implementar el comportamiento de gestión por defecto: puede pasar la ejecución al siguiente manejador después de comprobar su existencia.

3. Los **Manejadores Concretos** contienen el código para procesar las solicitudes. Al recibir una solicitud, cada manejador debe decidir si procesarla y, además, si la pasa a lo largo de la cadena.

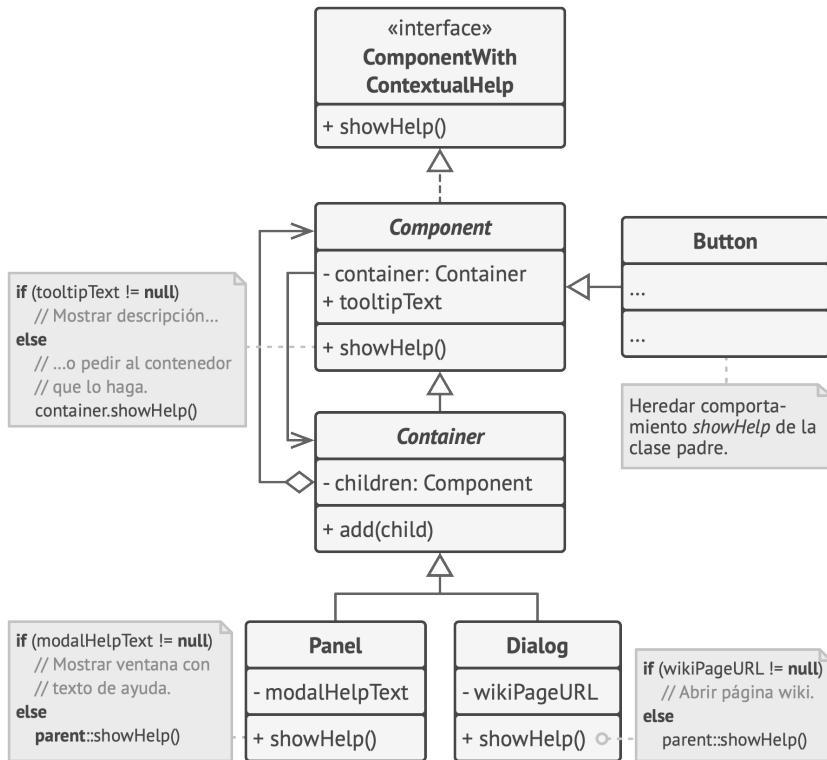
Habitualmente los manejadores son autónomos e inmutables, y aceptan toda la información necesaria únicamente a través del constructor.

4. El **Cliente** puede componer cadenas una sola vez o componerlas dinámicamente, dependiendo de la lógica de la aplicación.

Observa que se puede enviar una solicitud a cualquier manejador de la cadena; no tiene por qué ser al primero.

Pseudocódigo

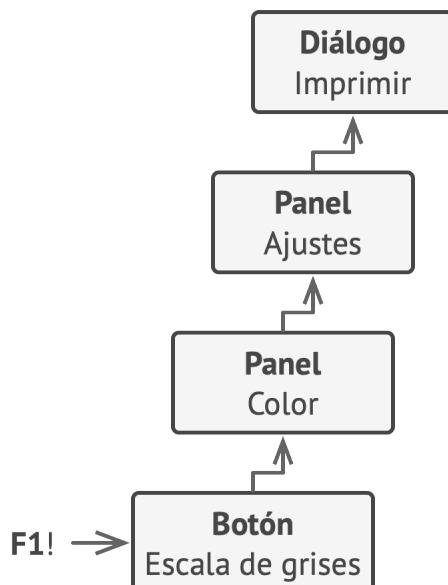
En este ejemplo, el patrón **Chain of Responsibility** es responsable de mostrar información de ayuda contextual para elementos GUI activos.



Las clases GUI se crean con el patrón Composite. Cada elemento se vincula a su elemento contenedor. En cualquier momento puedes crear una cadena de elementos que comience con el propio elemento y recorra todos los elementos contenedores.

La GUI de la aplicación se estructura normalmente como un árbol de objetos. Por ejemplo, la clase `Diálogo`, que representa la ventana principal de la aplicación, es la raíz del árbol de objetos. La clase diálogo contiene `Paneles`, que pueden contener otros paneles o simples elementos de bajo nivel, como `Botones` y `CamposdeTexto`.

Un simple componente puede mostrar breves pistas contextuales, siempre y cuando el componente tenga asignado cierto texto de ayuda. Pero los componentes más complejos definen su propia forma de mostrar ayuda contextual, por ejemplo, mostrando un extracto del manual o abriendo una página en un navegador.



Ésta es la forma en la que las solicitudes de ayuda recorren objetos GUI.

Cuando un usuario apunta el cursor del ratón a un elemento y pulsa la tecla F1, la aplicación detecta el componente bajo el puntero y le envía una solicitud de ayuda. La solicitud emerge por todos los contenedores del elemento hasta que llega al elemento capaz de mostrar la información de ayuda.

```
1 // La interfaz manejadora declara un método para construir una
2 // cadena de manejadores. También declara un método para
3 // ejecutar una solicitud.
4 interface ComponentWithContextualHelp is
5     method showHelp()
6
7
8 // La clase base para componentes simples.
9 abstract class Component implements ComponentWithContextualHelp is
10    field tooltipText: string
11
12 // El contenedor del componente actúa como el siguiente
13 // eslabón de la cadena de manejadores.
14 protected field container: Container
15
16 // El componente muestra una pista si tiene un texto de
17 // ayuda asignado. De lo contrario, reenvía la llamada al
18 // contenedor, si es que existe.
19 method showHelp() is
20    if (tooltipText != null)
21        // Muestra la pista.
22    else
23        container.showHelp()
24
25
```

```
26 // Los contenedores pueden contener componentes simples y otros
27 // contenedores como hijos. Las relaciones de la cadena se
28 // establecen aquí. La clase hereda el comportamiento showHelp
29 // (mostrarAyuda) de su padre.
30 abstract class Container extends Component is
31     protected field children: array of Component
32
33     method add(child) is
34         children.add(child)
35         child.container = this
36
37
38 // Los componentes primitivos pueden estar bien con la
39 // implementación de la ayuda por defecto...
40 class Button extends Component is
41     // ...
42
43 // Pero los componentes complejos pueden sobrescribir la
44 // implementación por defecto. Si no puede proporcionarse el
45 // texto de ayuda de una nueva forma, el componente siempre
46 // puede invocar la implementación base (véase la clase
47 // Componente).
48 class Panel extends Container is
49     field modalHelpText: string
50
51     method showHelp() is
52         if (modalHelpText != null)
53             // Muestra una ventana modal con el texto de ayuda.
54         else
55             super.showHelp()
56
57 // ...igual que arriba...
```

```
58 class Dialog extends Container is
59     field wikiPageURL: string
60
61     method showHelp() is
62         if (wikiPageURL != null)
63             // Abre la página de ayuda wiki.
64         else
65             super.showHelp()
66
67
68 // Código cliente.
69 class Application is
70     // Cada aplicación configura la cadena de forma diferente.
71     method createUI() is
72         dialog = new Dialog("Budget Reports")
73         dialog.wikiPageURL = "http://..."
74         panel = new Panel(0, 0, 400, 800)
75         panel.modalHelpText = "This panel does..."
76         ok = new Button(250, 760, 50, 20, "OK")
77         ok.tooltipText = "This is an OK button that..."
78         cancel = new Button(320, 760, 50, 20, "Cancel")
79         // ...
80         panel.add(ok)
81         panel.add(cancel)
82         dialog.add(panel)
83
84 // Imagina lo que pasa aquí.
85     method onF1KeyPress() is
86         component = this.getComponentAtMouseCoords()
87         component.showHelp()
```

Aplicabilidad

- ⚡ Utiliza el patrón **Chain of Responsibility** cuando tu programa deba procesar distintos tipos de solicitudes de varias maneras, pero los tipos exactos de solicitudes y sus secuencias no se conocan de antemano.
- ⚡ El patrón te permite encadenar varios manejadores y, al recibir una solicitud, “preguntar” a cada manejador si puede procesarla. De esta forma todos los manejadores tienen la oportunidad de procesar la solicitud.
- ⚡ Utiliza el patrón cuando sea fundamental ejecutar varios manejadores en un orden específico.
- ⚡ Ya que puedes vincular los manejadores de la cadena en cualquier orden, todas las solicitudes recorrerán la cadena exactamente como planees.
- ⚡ Utiliza el patrón **Chain of Responsibility** cuando el grupo de manejadores y su orden deban cambiar durante el tiempo de ejecución.
- ⚡ Si aportas modificadores (*setters*) para un campo de referencia dentro de las clases manejadoras, podrás insertar, eliminar o reordenar los manejadores dinámicamente.

Cómo implementarlo

1. Declara la interfaz manejadora y describe la firma de un método para manejar solicitudes.

Decide cómo pasará el cliente la información de la solicitud dentro del método. La forma más flexible consiste en convertir la solicitud en un objeto y pasarlo al método de gestión como argumento.

2. Para eliminar código boilerplate duplicado en manejadores concretos, puede merecer la pena crear una clase manejadora abstracta base, derivada de la interfaz manejadora.

Esta clase debe tener un campo para almacenar una referencia al siguiente manejador de la cadena. Considera hacer la clase inmutable. No obstante, si planeas modificar las cadenas durante el tiempo de ejecución, deberás definir un modificador (*setter*) para alterar el valor del campo de referencia.

También puedes implementar el comportamiento por defecto conveniente para el método de control, que consiste en reenviar la solicitud al siguiente objeto, a no ser que no quede ninguno. Los manejadores concretos podrán utilizar este comportamiento invocando al método padre.

3. Una a una, crea subclases manejadoras concretas e implementa los métodos de control. Cada manejador debe tomar dos decisiones cuando recibe una solicitud:

- Si procesa la solicitud.
 - Si pasa la solicitud al siguiente eslabón de la cadena.
4. El cliente puede ensamblar cadenas por su cuenta o recibir cadenas prefabricadas de otros objetos. En el último caso, debes implementar algunas clases fábrica para crear cadenas de acuerdo con los ajustes de configuración o de entorno.
5. El cliente puede activar cualquier manejador de la cadena, no solo el primero. La solicitud se pasará a lo largo de la cadena hasta que algún manejador se rehúse a pasarlo o hasta que llegue al final de la cadena.
6. Debido a la naturaleza dinámica de la cadena, el cliente debe estar listo para gestionar los siguientes escenarios:
- La cadena puede consistir en un único vínculo.
 - Algunas solicitudes pueden no llegar al final de la cadena.
 - Otras pueden llegar hasta el final de la cadena sin ser gestionadas.

Pros y contras

- ✓ Puedes controlar el orden de control de solicitudes.
- ✓ *Principio de responsabilidad única.* Puedes desacoplar las clases que invoquen operaciones de las que realicen operaciones.

- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevos manejadores en la aplicación sin descomponer el código cliente existente.
- ✗ Algunas solicitudes pueden acabar sin ser gestionadas.

↔ Relaciones con otros patrones

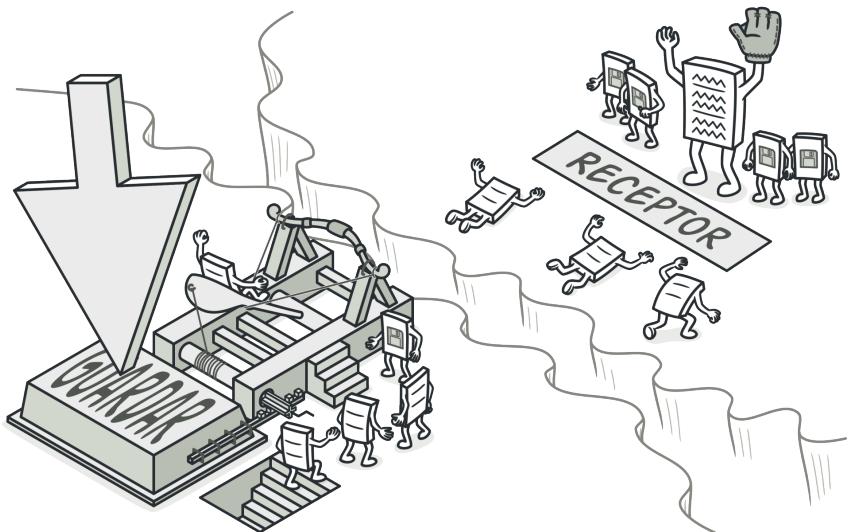
- **Chain of Responsibility**, **Command**, **Mediator** y **Observer** abordan distintas formas de conectar emisores y receptores de solicitudes:
 - *Chain of Responsibility* pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la gestiona.
 - *Command* establece conexiones unidireccionales entre emisores y receptores.
 - *Mediator* elimina las conexiones directas entre emisores y receptores, forzándolos a comunicarse indirectamente a través de un objeto mediador.
 - *Observer* permite a los receptores suscribirse o darse de baja dinámicamente a la recepción de solicitudes.
- **Chain of Responsibility** se utiliza a menudo junto a **Composite**. En este caso, cuando un componente hoja recibe una solicitud, puede pasarlala a lo largo de la cadena de todos los componentes padre hasta la raíz del árbol de objetos.

- Los manejadores del **Chain of Responsibility** se pueden implementar como **Comandos**. En este caso, puedes ejecutar muchas operaciones diferentes sobre el mismo objeto de contexto, representado por una solicitud.

Sin embargo, hay otra solución en la que la propia solicitud es un objeto *Comando*. En este caso, puedes ejecutar la misma operación en una serie de contextos diferentes vinculados en una cadena.

- **Chain of Responsibility** y **Decorator** tienen estructuras de clase muy similares. Ambos patrones se basan en la composición recursiva para pasar la ejecución a través de una serie de objetos. Sin embargo, existen varias diferencias fundamentales:

Los manejadores de *CoR* pueden ejecutar operaciones arbitrarias con independencia entre sí. También pueden dejar de pasar la solicitud en cualquier momento. Por otro lado, varios *decoradores* pueden extender el comportamiento del objeto manteniendo su consistencia con la interfaz base. Además, los decoradores no pueden romper el flujo de la solicitud.



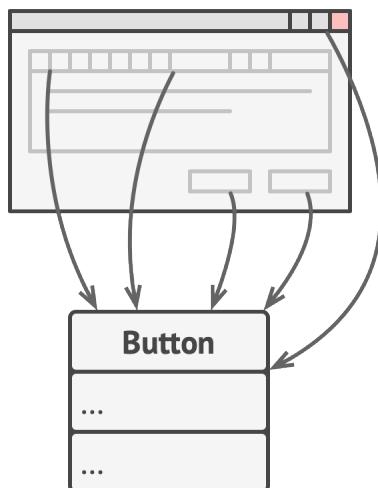
COMMAND

También llamado: Comando, Orden, Action, Transaction

Command es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y sopor tar operaciones que no se pueden realizar.

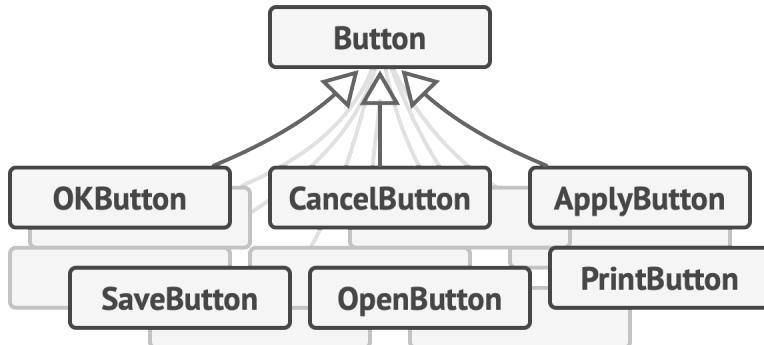
(:() Problema

Imagina que estás trabajando en una nueva aplicación de edición de texto. Tu tarea actual consiste en crear una barra de herramientas con unos cuantos botones para varias operaciones del editor. Crea una clase **Botón** muy limpia que puede utilizarse para los botones de la barra de herramientas y también para botones genéricos en diversos diálogos.



Todos los botones de la aplicación provienen de la misma clase.

Aunque todos estos botones se parecen, se supone que hacen cosas diferentes. ¿Dónde pondrías el código para los varios gestores de clics de estos botones? La solución más simple consiste en crear cientos de subclases para cada lugar donde se utilice el botón. Estas subclases contendrán el código que deberá ejecutarse con el clic en un botón.



Muchas subclases de botón. ¿Qué puede salir mal?

Pronto te das cuenta de que esta solución es muy deficiente. En primer lugar, tienes una enorme cantidad de subclases, lo cual no supondría un problema si no corrieras el riesgo de descomponer el código de esas subclases cada vez que modifiques la clase base `Botón`. Dicho de forma sencilla, tu código GUI depende torpemente del volátil código de la lógica de negocio.



Varias clases implementan la misma funcionalidad.

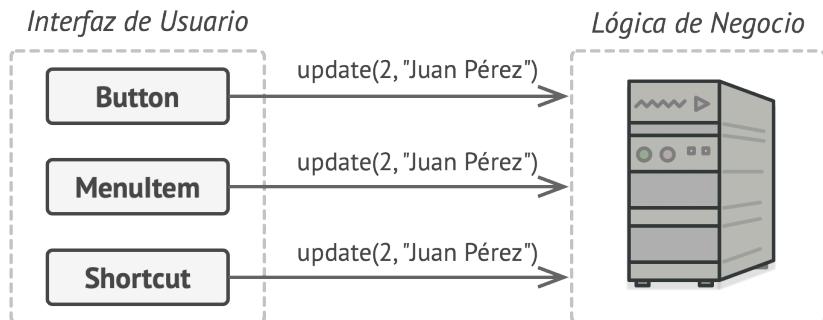
Y aquí está la parte más desagradable. Algunas operaciones, como copiar/pegar texto, deben ser invocadas desde varios lugares. Por ejemplo, un usuario podría hacer clic en un pequeño botón “Copiar” de la barra de herramientas, o copiar algo a través del menú contextual, o pulsar `Ctrl+C` en el teclado.

Inicialmente, cuando tu aplicación solo tenía la barra de herramientas, no había problema en colocar la implementación de varias operaciones dentro de las subclases de botón. En otras palabras, tener el código para copiar texto dentro de la subclase `BotónCopiar` estaba bien. Sin embargo, cuando implementas menús contextuales, atajos y otros elementos, debes duplicar el código de la operación en muchas clases, o bien hacer menús dependientes de los botones, lo cual es una opción aún peor.

Solución

El buen diseño de software a menudo se basa en el principio de separación de responsabilidades, lo que suele tener como resultado la división de la aplicación en capas. El ejemplo más habitual es tener una capa para la interfaz gráfica de usuario (GUI) y otra capa para la lógica de negocio. La capa GUI es responsable de representar una bonita imagen en pantalla, capturar entradas y mostrar resultados de lo que el usuario y la aplicación están haciendo. Sin embargo, cuando se trata de hacer algo importante, como calcular la trayectoria de la luna o componer un informe anual, la capa GUI delega el trabajo a la capa subyacente de la lógica de negocio.

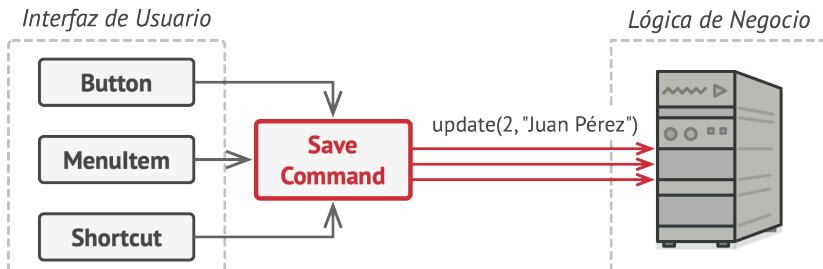
El código puede tener este aspecto: un objeto GUI invoca a un método de un objeto de la lógica de negocio, pasándole algunos argumentos. Este proceso se describe habitualmente como un objeto que envía a otro una *solicitud*.



Los objetos GUI pueden acceder directamente a los objetos de la lógica de negocio.

El patrón Command sugiere que los objetos GUI no envíen estas solicitudes directamente. En lugar de ello, debes extraer todos los detalles de la solicitud, como el objeto que está siendo invocado, el nombre del método y la lista de argumentos, y ponerlos dentro de una clase *comando* separada con un único método que activa esta solicitud.

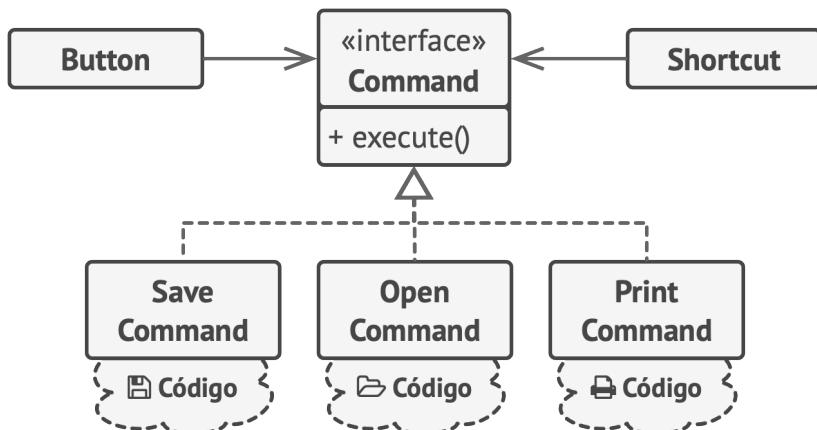
Los objetos de comando sirven como vínculo entre varios objetos GUI y de lógica de negocio. De ahora en adelante, el objeto GUI no tiene que conocer qué objeto de la lógica de negocio recibirá la solicitud y cómo la procesará. El objeto GUI activa el comando, que gestiona todos los detalles.



Acceso a la capa de lógica de negocio a través de un comando.

El siguiente paso es hacer que tus comandos implementen la misma interfaz. Normalmente tiene un único método de ejecución que no acepta parámetros. Esta interfaz te permite utilizar varios comandos con el mismo emisor de la solicitud, sin acoplarla a clases concretas de comandos. Adicionalmente, ahora puedes cambiar objetos de comando vinculados al emisor, cambiando efectivamente el comportamiento del emisor durante el tiempo de ejecución.

Puede que hayas observado que falta una pieza del rompecabezas, que son los parámetros de la solicitud. Un objeto GUI puede haber proporcionado al objeto de la capa de negocio algunos parámetros. Ya que el método de ejecución del comando no tiene parámetros, ¿cómo pasaremos los detalles de la solicitud al receptor? Resulta que el comando debe estar preconfigurado con esta información o ser capaz de conseguirla por su cuenta.



Los objetos GUI delegan el trabajo a los comandos.

Regresemos a nuestro editor de textos. Tras aplicar el patrón Command, ya no necesitamos todas esas subclases de botón para implementar varios comportamientos de clic. Basta con colocar un único campo dentro de la clase base Botón que almacene una referencia a un objeto de comando y haga que el botón ejecute ese comando en un clic.

Implementarás un puñado de clases de comando para toda operación posible y las vincularás con botones particulares, dependiendo del comportamiento pretendido de los botones.

Otros elementos GUI, como menús, atajos o diálogos enteros, se pueden implementar del mismo modo. Se vincularán a un comando que se ejecuta cuando un usuario interactúa con el elemento GUI. Como probablemente ya habrás adivinado, los elementos relacionados con las mismas operaciones se vincu-

larán a los mismos comandos, evitando cualquier duplicación de código.

Como resultado, los comandos se convierten en una conveniente capa intermedia que reduce el acoplamiento entre las capas de la GUI y la lógica de negocio. ¡Y esto es tan solo una fracción de las ventajas que ofrece el patrón Command!

🚗 Analogía en el mundo real



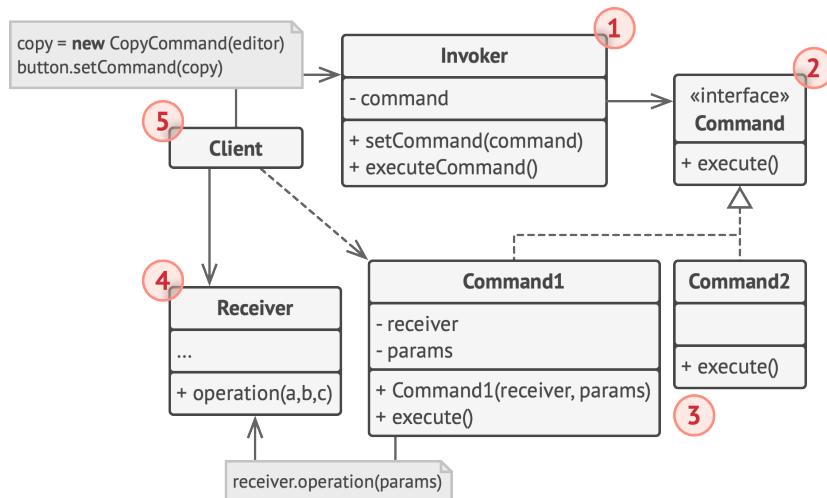
Realizando un pedido en un restaurante.

Tras un largo paseo por la ciudad, entras en un buen restaurante y te sientas a una mesa junto a la ventana. Un amable camarero se acerca y toma tu pedido rápidamente, apuntándolo en un papel. El camarero se va a la cocina y pega el pedido a la pared. Al cabo de un rato, el pedido llega al chef, que lo lee y prepara la comida. El cocinero coloca la comida en una bandeja junto al pedido. El camarero descubre la bandeja, co-

mprueba el pedido para asegurarse de que todo está como lo querías, y lo lleva todo a tu mesa.

El pedido en papel hace la función de un comando. Permanece en una cola hasta que el chef está listo para servirlo. Este pedido contiene toda la información relevante necesaria para preparar la comida. Permite al chef empezar a cocinar de inmediato, en lugar de tener que correr de un lado a otro aclarando los detalles del pedido directamente contigo.

Estructura



1. La clase **Emisora** (o *invocadora*) es responsable de inicializar las solicitudes. Esta clase debe tener un campo para almacenar una referencia a un objeto de comando. El emisor activa este comando en lugar de enviar la solicitud directamente al receptor. Ten en cuenta que el emisor no es responsable de crear el

objeto de comando. Normalmente, obtiene un comando pre-creado de parte del cliente a través del constructor.

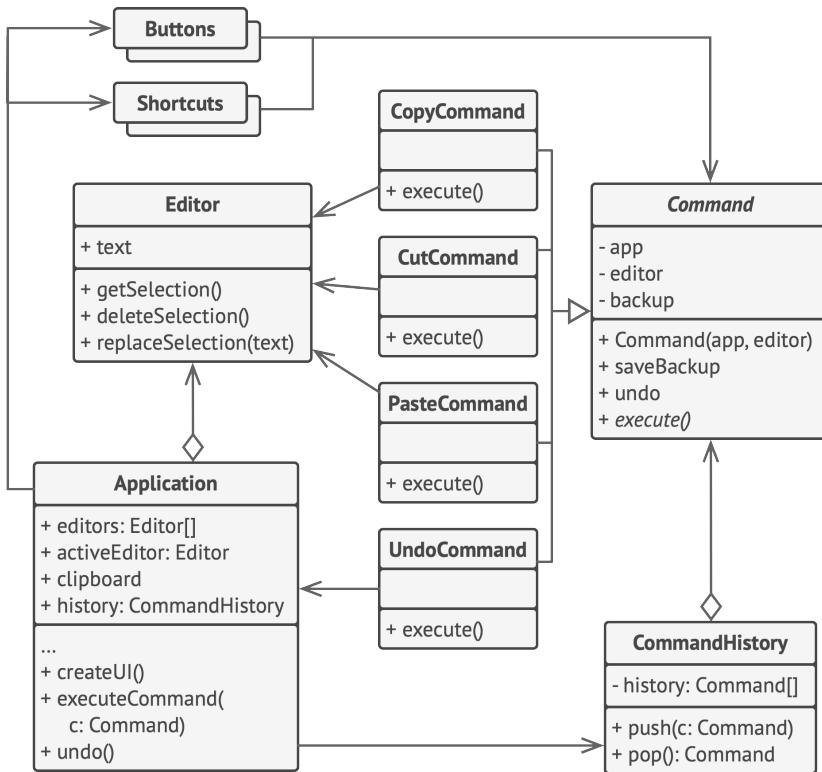
2. La interfaz **Comando** normalmente declara un único método para ejecutar el comando.
3. Los **Comandos Concretos** implementan varios tipos de solicitudes. Un comando concreto no se supone que tenga que realizar el trabajo por su cuenta, sino pasar la llamada a uno de los objetos de la lógica de negocio. Sin embargo, para lograr simplificar el código, estas clases se pueden fusionar.

Los parámetros necesarios para ejecutar un método en un objeto receptor pueden declararse como campos en el comando concreto. Puedes hacer inmutables los objetos de comando permitiendo la inicialización de estos campos únicamente a través del constructor.

4. La clase **Receptora** contiene cierta lógica de negocio. Casi cualquier objeto puede actuar como receptor. La mayoría de los comandos solo gestiona los detalles sobre cómo se pasa una solicitud al receptor, mientras que el propio receptor hace el trabajo real.
5. El **Cliente** crea y configura los objetos de comando concretos. El cliente debe pasar todos los parámetros de la solicitud, incluyendo una instancia del receptor, dentro del constructor del comando. Después de eso, el comando resultante puede asociarse con uno o varios emisores.

Pseudocódigo

En este ejemplo, el patrón **Command** ayuda a rastrear el historial de operaciones ejecutadas y hace posible revertir una operación si es necesario.



Operaciones que no se pueden realizar en un editor de texto.

Los comandos que resultan en cambiar el estado del editor (por ejemplo, cortar y pegar) realizan una copia de seguridad del estado del editor antes de ejecutar una operación asociada con el comando. Una vez que un comando es ejecutado,

se coloca en el historial del comando (una pila de objetos de comando) junto a la copia de seguridad del estado del editor en ese momento. Más tarde, si el usuario necesita revertir la operación, la aplicación puede tomar el comando más reciente del historial, leer la copia asociada del estado del editor, y restaurarla.

El código cliente (elementos GUI, historial de comando, etc.) no se acopla a clases concretas de comando porque trabaja con los comandos a través de la interfaz de comando. Esta solución te permite introducir nuevos comandos en la aplicación sin descomponer el código existente.

```
1 // La clase base comando define la interfaz común a todos los
2 // comandos concretos.
3 abstract class Command is
4     protected field app: Application
5     protected field editor: Editor
6     protected field backup: text
7
8     constructor Command(app: Application, editor: Editor) is
9         this.app = app
10        this.editor = editor
11
12     // Realiza una copia de seguridad del estado del editor.
13     method saveBackup() is
14         backup = editor.text
15
16     // Restaura el estado del editor.
17     method undo() is
```

```
18     editor.text = backup
19
20     // El método de ejecución se declara abstracto para forzar a
21     // todos los comandos concretos a proporcionar sus propias
22     // implementaciones. El método debe devolver verdadero o
23     // falso dependiendo de si el comando cambia el estado del
24     // editor.
25     abstract method execute()
26
27
28 // Los comandos concretos van aquí.
29 class CopyCommand extends Command is
30     // El comando copiar no se guarda en el historial ya que no
31     // cambia el estado del editor.
32     method execute() is
33         app.clipboard = editor.getSelection()
34         return false
35
36 class CutCommand extends Command is
37     // El comando cortar no cambia el estado del editor, por lo
38     // que debe guardarse en el historial. Y se guardará siempre
39     // y cuando el método devuelva verdadero.
40     method execute() is
41         saveBackup()
42         app.clipboard = editor.getSelection()
43         editor.deleteSelection()
44         return true
45
46 class PasteCommand extends Command is
47     method execute() is
48         saveBackup()
49         editor.replaceSelection(app.clipboard)
```

```
50     return true
51
52 // La operación deshacer también es un comando.
53 class UndoCommand extends Command is
54     method execute() is
55         app.undo()
56         return false
57
58
59 // El historial global de comandos tan solo es una pila.
60 class CommandHistory is
61     private field history: array of Command
62
63 // El último dentro...
64 method push(c: Command) is
65     // Empuja el comando al final de la matriz del
66     // historial.
67
68 // ...el primero fuera.
69 method pop():Command is
70     // Obtiene el comando más reciente del historial.
71
72
73 // La clase editora tiene operaciones reales de edición de
74 // texto. Juega el papel de un receptor: todos los comandos
75 // acaban delegando la ejecución a los métodos del editor.
76 class Editor is
77     field text: string
78
79 method getSelection() is
80     // Devuelve el texto seleccionado.
```

```
82 method deleteSelection() is
83     // Borra el texto seleccionado.
84
85 method replaceSelection(text) is
86     // Inserta los contenidos del portapapeles en la
87     // posición actual.
88
89
90 // La clase Aplicación establece relaciones entre objetos. Actúa
91 // como un emisor: cuando algo debe hacerse, crea un objeto de
92 // comando y lo ejecuta.
93 class Application is
94     field clipboard: string
95     field editors: array of Editors
96     field activeEditor: Editor
97     field history: CommandHistory
98
99 // El código que asigna comandos a objetos UI puede tener
100 // este aspecto.
101 method createUI() is
102     // ...
103     copy = function() { executeCommand(
104         new CopyCommand(this, activeEditor)) }
105     copyButton.setCommand(copy)
106     shortcuts.onKeyPress("Ctrl+C", copy)
107
108     cut = function() { executeCommand(
109         new CutCommand(this, activeEditor)) }
110     cutButton.setCommand(cut)
111     shortcuts.onKeyPress("Ctrl+X", cut)
112
113     paste = function() { executeCommand(
```

```

114     new PasteCommand(this, activeEditor)) }
115     pasteButton.setCommand(paste)
116     shortcuts.onKeyPress("Ctrl+V", paste)
117
118     undo = function() { executeCommand(
119         new UndoCommand(this, activeEditor)) }
120     undoButton.setCommand(undo)
121     shortcuts.onKeyPress("Ctrl+Z", undo)
122
123 // Ejecuta un comando y comprueba si debe añadirse al
124 // historial.
125 method executeCommand(command) is
126     if (command.execute)
127         history.push(command)
128
129 // Toma el comando más reciente del historial y ejecuta su
130 // método deshacer. Observa que no conocemos la clase de ese
131 // comando. Pero no tenemos por qué, ya que el comando sabe
132 // cómo deshacer su propia acción.
133 method undo() is
134     command = history.pop()
135     if (command != null)
136         command.undo()

```

Aplicabilidad

-  Utiliza el patrón Command cuando quieras parametrizar objetos con operaciones.
-  El patrón Command puede convertir una llamada a un método específico en un objeto autónomo. Este cambio abre la puer-

ta a muchos usos interesantes: puedes pasar comandos como argumentos de método, almacenarlos dentro de otros objetos, cambiar comandos vinculados durante el tiempo de ejecución, etc.

Aquí tienes un ejemplo: estás desarrollando un componente GUI, como un menú contextual, y quieres que los usuarios puedan configurar opciones del menú que activen operaciones cuando un usuario final haga clic sobre ellos.

- ⚡ **Utiliza el patrón Command cuando quieras poner operaciones en cola, programar su ejecución, o ejecutarlas de forma remota.**
- ⚡ Como pasa con cualquier otro objeto, un comando se pueden serializar, lo cual implica convertirlo en una cadena que pueda escribirse fácilmente a un archivo o una base de datos. Más tarde, la cadena puede restaurarse como el objeto de comando inicial. De este modo, puedes retardar y programar la ejecución del comando. ¡Pero aún hay más! Del mismo modo, puedes poner comandos en cola, así como registrarlos o enviarlos por la red.
- ⚡ **Utiliza el patrón Command cuando quieras implementar operaciones reversibles.**
- ⚡ Aunque hay muchas formas de implementar deshacer/rehacer, el patrón Command es quizá la más popular de todas.

Para poder revertir operaciones, debes implementar el historial de las operaciones realizadas. El historial de comando es una pila que contiene todos los objetos de comando ejecutados junto a copias de seguridad relacionadas del estado de la aplicación.

Este método tiene dos desventajas. Primero, no es tan fácil guardar el estado de una aplicación, porque parte de ella puede ser privada. Este problema puede mitigarse con el patrón **Memento**.

Segundo, las copias de seguridad de estado pueden consumir mucha memoria RAM. Por lo tanto, en ocasiones puedes recurrir a una implementación alternativa: en lugar de restaurar el estado pasado, el comando realiza la operación inversa, aunque ésta también tiene un precio, ya que puede resultar difícil o incluso imposible de implementar.

Cómo implementarlo

1. Declara la interfaz de comando con un único método de ejecución.
2. Empieza extrayendo solicitudes y poniéndolas dentro de clases concretas de comando que implementen la interfaz de comando. Cada clase debe contar con un grupo de campos para almacenar los argumentos de las solicitudes junto con referencias al objeto receptor. Todos estos valores deben inicializarse a través del constructor del comando.

3. Identifica clases que actúen como *emisoras*. Añade los campos para almacenar comandos dentro de estas clases. Las emisoras deberán comunicarse con sus comandos tan solo a través de la interfaz de comando. Normalmente las emisoras no crean objetos de comando por su cuenta, sino que los obtienen del código cliente.
4. Cambia las emisoras de forma que ejecuten el comando en lugar de enviar directamente una solicitud al receptor.
5. El cliente debe inicializar objetos en el siguiente orden:
 - Crear receptores.
 - Crear comandos y asociarlos con receptores si es necesario.
 - Crear emisores y asociarlos con comandos específicos.

ΔΔ Pros y contras

- ✓ *Principio de responsabilidad única*. Puedes desacoplar las clases que invocan operaciones de las que realizan esas operaciones.
- ✓ *Principio de abierto/cerrado*. Puedes introducir nuevos comandos en la aplicación sin descomponer el código cliente existente.
- ✓ Puedes implementar deshacer/rehacer.
- ✓ Puedes implementar la ejecución diferida de operaciones.
- ✓ Puedes ensamblar un grupo de comandos simples para crear uno complejo.

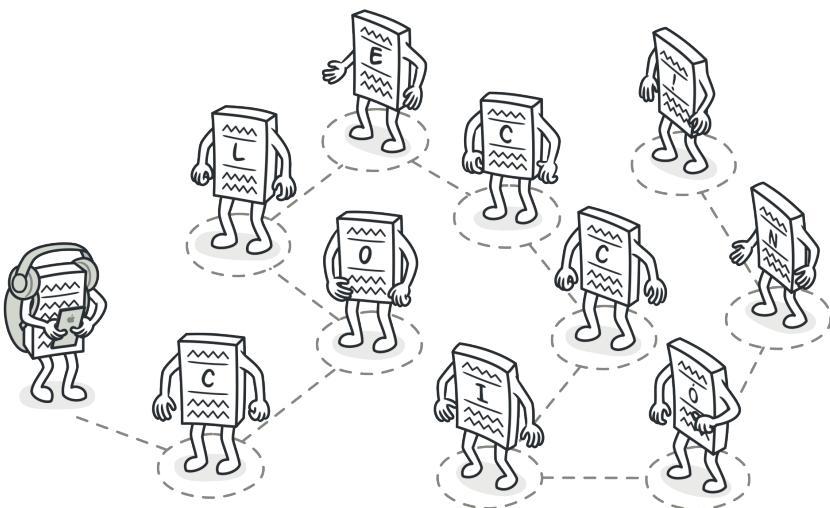
- ✗ El código puede complicarse, ya que estás introduciendo una nueva capa entre emisores y receptores.

↔ Relaciones con otros patrones

- Chain of Responsibility, Command, Mediator y Observer abordan distintas formas de conectar emisores y receptores de solicitudes:
 - *Chain of Responsibility* pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la gestiona.
 - *Command* establece conexiones unidireccionales entre emisores y receptores.
 - *Mediator* elimina las conexiones directas entre emisores y receptores, forzándolos a comunicarse indirectamente a través de un objeto mediador.
 - *Observer* permite a los receptores suscribirse o darse de baja dinámicamente a la recepción de solicitudes.
- Los manejadores del **Chain of Responsibility** se pueden implementar como **Comandos**. En este caso, puedes ejecutar muchas operaciones diferentes sobre el mismo objeto de contexto, representado por una solicitud.

Sin embargo, hay otra solución en la que la propia solicitud es un objeto *Comando*. En este caso, puedes ejecutar la misma operación en una serie de contextos diferentes vinculados en una cadena.

- Puedes utilizar **Command** y **Memento** juntos cuando implementes “deshacer”. En este caso, los comandos son responsables de realizar varias operaciones sobre un objeto destino, mientras que los mementos guardan el estado de ese objeto justo antes de que se ejecute el comando.
- **Command** y **Strategy** pueden resultar similares porque puedes usar ambos para parametrizar un objeto con cierta acción. No obstante, tienen propósitos muy diferentes.
 - Puedes utilizar *Command* para convertir cualquier operación en un objeto. Los parámetros de la operación se convierten en campos de ese objeto. La conversión te permite aplazar la ejecución de la operación, ponerla en cola, almacenar el historial de comandos, enviar comandos a servicios remotos, etc.
 - Por su parte, *Strategy* normalmente describe distintas formas de hacer lo mismo, permitiéndote intercambiar estos algoritmos dentro de una única clase contexto.
- **Prototype** puede ayudar a cuando necesitas guardar copias de **Comandos** en un historial.
- Puedes tratar a **Visitor** como una versión potente del patrón **Command**. Sus objetos pueden ejecutar operaciones sobre varios objetos de distintas clases.



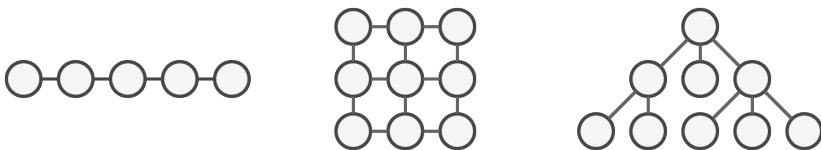
ITERATOR

También llamado: Iterador

Iterator es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

(:() Problema

Las colecciones son de los tipos de datos más utilizados en programación. Sin embargo, una colección tan solo es un contenedor para un grupo de objetos.



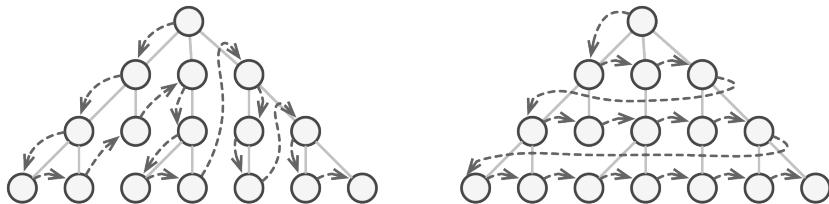
Varios tipos de colecciones.

La mayoría de las colecciones almacena sus elementos en simples listas, pero algunas de ellas se basan en pilas, árboles, grafos y otras estructuras complejas de datos.

Independientemente de cómo se estructure una colección, debe aportar una forma de acceder a sus elementos de modo que otro código pueda utilizar dichos elementos. Debe haber una forma de recorrer cada elemento de la colección sin acceder a los mismos elementos una y otra vez.

Esto puede parecer un trabajo sencillo si tienes una colección basada en una lista. En este caso sólo tienes que recorrer en bucle todos sus elementos. Pero, ¿cómo recorres secuencialmente elementos de una estructura compleja de datos, como un árbol? Por ejemplo, un día puede bastarte con un recorrido de profundidad de un árbol, pero, al día siguiente, quizás necesites un recorrido en anchura. Y, la semana siguiente, puedes

necesitar otra cosa, como un acceso aleatorio a los elementos del árbol.



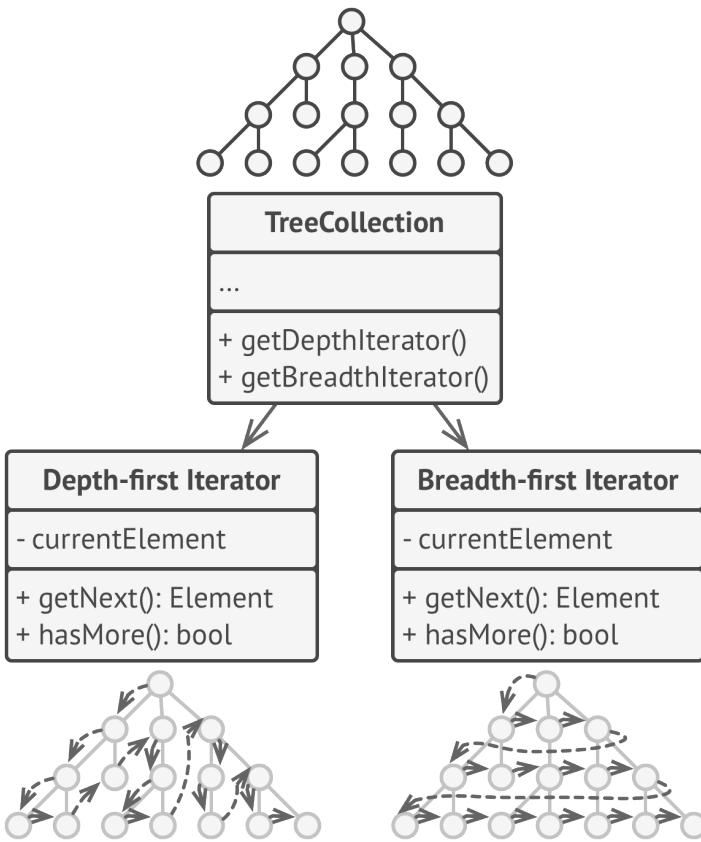
La misma colección puede recorrerse de varias formas diferentes.

Añadir más y más algoritmos de recorrido a la colección nubla gradualmente su responsabilidad principal, que es el almacenamiento eficiente de la información. Además, puede que algunos algoritmos estén personalizados para una aplicación específica, por lo que incluirlos en una clase genérica de colección puede resultar extraño.

Por otro lado, el código cliente que debe funcionar con varias colecciones puede no saber cómo éstas almacenan sus elementos. No obstante, ya que todas las colecciones proporcionan formas diferentes de acceder a sus elementos, no tienes otra opción más que acoplar tu código a las clases de la colección específica.

😊 Solución

La idea central del patrón Iterator es extraer el comportamiento de recorrido de una colección y colocarlo en un objeto independiente llamado *iterador*.



Los iteradores implementan varios algoritmos de recorrido. Varios objetos iteradores pueden recorrer la misma colección al mismo tiempo.

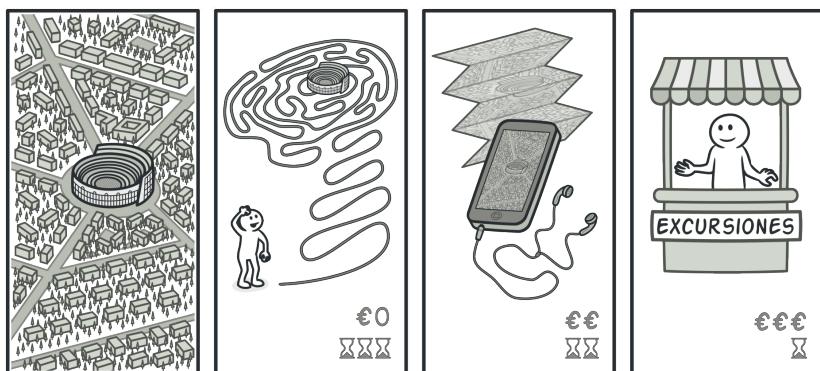
Además de implementar el propio algoritmo, un objeto iterador encapsula todos los detalles del recorrido, como la posición actual y cuántos elementos quedan hasta el final. Debido a esto, varios iteradores pueden recorrer la misma colección al mismo tiempo, independientemente los unos de los otros.

Normalmente, los iteradores aportan un método principal para extraer elementos de la colección. El cliente puede continuar

ejecutando este método hasta que no devuelva nada, lo que significa que el iterador ha recorrido todos los elementos.

Todos los iteradores deben implementar la misma interfaz. Esto hace que el código cliente sea compatible con cualquier tipo de colección o cualquier algoritmo de recorrido, siempre y cuando exista un iterador adecuado. Si necesitas una forma particular de recorrer una colección, creas una nueva clase iteradora sin tener que cambiar la colección o el cliente.

🚗 Analogía en el mundo real



Varias formas de pasear por Roma.

Planeas visitar Roma por unos días y ver todas sus atracciones y puntos de interés. Pero, una vez allí, podrías perder mucho tiempo dando vueltas, incapaz de encontrar siquiera el Coliseo.

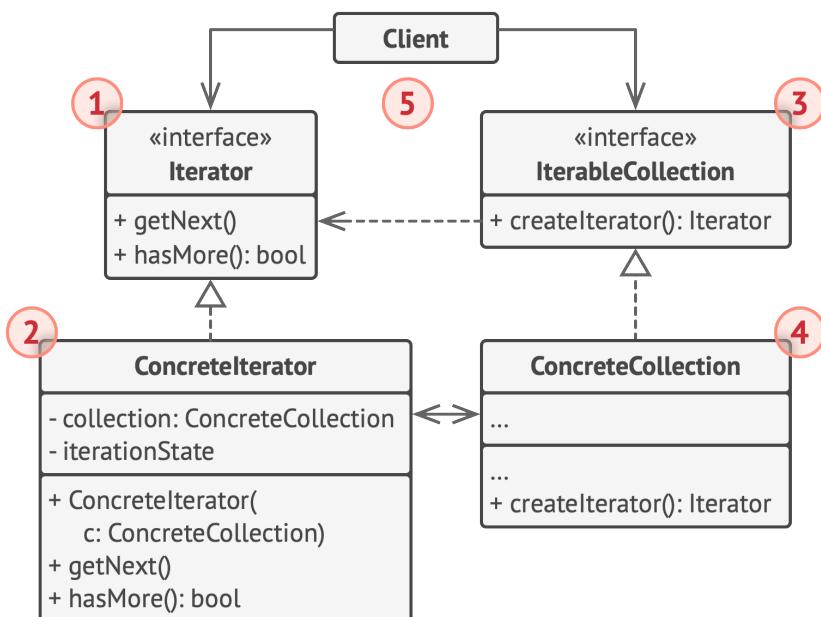
En lugar de eso, podrías comprar una aplicación de guía virtual para tu smartphone y utilizarla para moverte. Es buena y

barata y puedes quedarte en sitios interesantes todo el tiempo que quieras.

Una tercera alternativa sería dedicar parte del presupuesto del viaje a contratar un guía local que conozca la ciudad como la palma de su mano. El guía podría adaptar la visita a tus gustos, mostrarte las atracciones y contarte un montón de emocionantes historias. Eso sería más divertido pero, lamentablemente, también más caro.

Todas estas opciones –las direcciones aleatorias en tu cabeza, el navegador del smartphone o el guía humano–, actúan como iteradores sobre la amplia colección de visitas y atracciones de Roma.

Estructura



1. La interfaz **Iteradora** declara las operaciones necesarias para recorrer una colección: extraer el siguiente elemento, recuperar la posición actual, reiniciar la iteración, etc.
2. Los **Iteradores Concretos** implementan algoritmos específicos para recorrer una colección. El objeto iterador debe controlar el progreso del recorrido por su cuenta. Esto permite a varios iteradores recorrer la misma colección con independencia entre sí.
3. La interfaz **Colección** declara uno o varios métodos para obtener iteradores compatibles con la colección. Observa que el tipo de retorno de los métodos debe declararse como la inte-

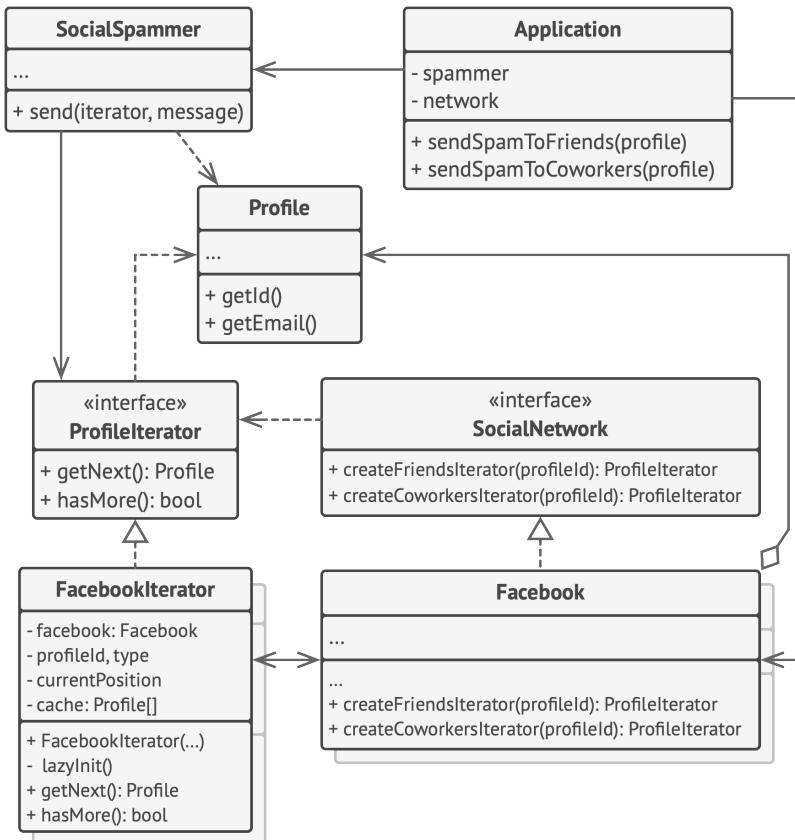
rfaz iteradora de forma que las colecciones concretas puedan devolver varios tipos de iteradores.

4. Las **Colecciones Concretas** devuelven nuevas instancias de una clase iteradora concreta particular cada vez que el cliente solicita una. Puede que te estés preguntando: ¿dónde está el resto del código de la colección? No te preocupes, debe estar en la misma clase. Lo que pasa es que estos detalles no son fundamentales para el patrón en sí, por eso los omitimos.
5. El **Cliente** debe funcionar con colecciones e iteradores a través de sus interfaces. De este modo, el cliente no se acopla a clases concretas, permitiéndote utilizar varias colecciones e iteradores con el mismo código cliente.

Normalmente, los clientes no crean iteradores por su cuenta, en lugar de eso, los obtienen de las colecciones. Sin embargo, en algunos casos, el cliente puede crear uno directamente, como cuando define su propio iterador especial.

Pseudocódigo

En este ejemplo, el patrón **Iterator** se utiliza para recorrer un tipo especial de colección que encapsula el acceso al grafo social de Facebook. La colección proporciona varios iteradores que recorren perfiles de distintas formas.



Ejemplo de iteración de perfiles sociales.

El iterador ‘amigos’ puede utilizarse para recorrer los amigos de un perfil dado. El iterador ‘colegas’ hace lo mismo, excepto que omite amigos que no trabajen en la misma empresa que la persona objetivo. Ambos iteradores implementan una interfaz común que permite a los clientes extraer perfiles sin profundizar en los detalles de la implementación, como la autenticación y el envío de solicitudes REST.

El código cliente no está acoplado a clases concretas porque sólo trabaja con colecciones e iteradores a través de interfaces. Si decides conectar tu aplicación a una nueva red social, sólo necesitas proporcionar nuevas clases de colección e iteradoras, sin cambiar el código existente.

```
1 // La interfaz de colección debe declarar un método fábrica para
2 // producir iteradores. Puedes declarar varios métodos si hay
3 // distintos tipos de iteración disponibles en tu programa.
4 interface SocialNetwork is
5     method createFriendsIterator(profileId):ProfileIterator
6     method createCoworkersIterator(profileId):ProfileIterator
7
8
9 // Cada colección concreta está acoplada a un grupo de clases
10 // iteradoras concretas que devuelve, pero el cliente no lo
11 // está, ya que la firma de estos métodos devuelve interfaces
12 // iteradoras.
13 class Facebook implements SocialNetwork is
14     // ... El grueso del código de la colección debe ir aquí ...
15     // Código de creación del iterador.
16     method createFriendsIterator(profileId) is
17         return new FacebookIterator(this, profileId, "friends")
18     method createCoworkersIterator(profileId) is
19         return new FacebookIterator(this, profileId, "coworkers")
20
21
22 // La interfaz común a todos los iteradores.
23 interface ProfileIterator is
24     method getNext():Profile
25     method hasMore():bool
```

```
26
27
28 // La clase iteradora concreta.
29 class FacebookIterator implements ProfileIterator is
30     // El iterador necesita una referencia a la colección que
31     // recorre.
32     private field facebook: Facebook
33     private field profileId, type: string
34
35     // Un objeto iterador recorre la colección
36     // independientemente de otro iterador, por eso debe
37     // almacenar el estado de iteración.
38     private field currentPosition
39     private field cache: array of Profile
40
41 constructor FacebookIterator(facebook, profileId, type) is
42     this.facebook = facebook
43     this.profileId = profileId
44     this.type = type
45
46     private method lazyInit() is
47         if (cache == null)
48             cache = facebook.socialGraphRequest(profileId, type)
49
50     // Cada clase iteradora concreta tiene su propia
51     // implementación de la interfaz iteradora común.
52     method getNext() is
53         if (hasMore())
54             currentPosition++
55             return cache[currentPosition]
56
57     method hasMore() is
```

```
58     lazyInit()
59     return currentPosition < cache.length
60
61
62 // Aquí tienes otro truco útil: puedes pasar un iterador a una
63 // clase cliente en lugar de darle acceso a una colección
64 // completa. De esta forma, no expones la colección al cliente.
65 //
66 // Y hay otra ventaja: puedes cambiar la forma en la que el
67 // cliente trabaja con la colección durante el tiempo de
68 // ejecución pasándole un iterador diferente. Esto es posible
69 // porque el código cliente no está acoplado a clases iteradoras
70 // concretas.
71 class SocialSpammer is
72     method send(iterator: ProfileIterator, message: string) is
73         while (iterator.hasMore())
74             profile = iterator.getNext()
75             System.sendEmail(profile.getEmail(), message)
76
77
78 // La clase Aplicación configura colecciones e iteradores y
79 // después los pasa al código cliente.
80 class Application is
81     field network: SocialNetwork
82     field spammer: SocialSpammer
83
84     method config() is
85         if working with Facebook
86             this.network = new Facebook()
87         if working with LinkedIn
88             this.network = new LinkedIn()
89         this.spammer = new SocialSpammer()
```

```
90  
91     method sendSpamToFriends(profile) is  
92         iterator = network.createFriendsIterator(profile.getId())  
93         spammer.send(iterator, "Very important message")  
94  
95     method sendSpamToCoworkers(profile) is  
96         iterator = network.createCoworkersIterator(profile.getId())  
97         spammer.send(iterator, "Very important message")
```

💡 Aplicabilidad

- ⚡ Utiliza el patrón Iterator cuando tu colección tenga una estructura de datos compleja a nivel interno, pero quieras ocultar su complejidad a los clientes (ya sea por conveniencia o por razones de seguridad).
- ⚡ El iterador encapsula los detalles del trabajo con una estructura de datos compleja, proporcionando al cliente varios métodos simples para acceder a los elementos de la colección. Esta solución, además de ser muy conveniente para el cliente, también protege la colección frente a acciones descuidadas o maliciosas que el cliente podría realizar si trabajara con la colección directamente.
- ⚡ Utiliza el patrón para reducir la duplicación en el código de recorrido a lo largo de tu aplicación.
- ⚡ El código de los algoritmos de iteración no triviales tiende a ser muy voluminoso. Cuando se coloca dentro de la lógica de

negocio de una aplicación, puede nublar la responsabilidad del código original y hacerlo más difícil de mantener. Mover el código de recorrido a iteradores designados puede ayudarte a hacer el código de la aplicación más breve y limpio.

-  **Utiliza el patrón Iterator cuando quieras que tu código pueda recorrer distintas estructuras de datos, o cuando los tipos de estas estructuras no se conozcan de antemano.**
-  El patrón proporciona un par de interfaces genéricas para colecciones e iteradores. Teniendo en cuenta que ahora tu código utiliza estas interfaces, seguirá funcionando si le pasas varios tipos de colecciones e iteradores que implementen esas interfaces.

Cómo implementarlo

1. Declara la interfaz iteradora. Como mínimo, debe tener un método para extraer el siguiente elemento de una colección. Por conveniencia, puedes añadir un par de métodos distintos, como para extraer el elemento previo, localizar la posición actual o comprobar el final de la iteración.
2. Declara la interfaz de colección y describe un método para buscar iteradores. El tipo de retorno debe ser igual al de la interfaz iteradora. Puedes declarar métodos similares si planeas tener varios grupos distintos de iteradores.

3. Implementa clases iteradoras concretas para las colecciones que quieras que sean recorridas por iteradores. Un objeto iterador debe estar vinculado a una única instancia de la colección. Normalmente, este vínculo se establece a través del constructor del iterador.
4. Implementa la interfaz de colección en tus clases de colección. La idea principal es proporcionar al cliente un atajo para crear iteradores personalizados para una clase de colección particular. El objeto de colección debe pasarse a sí mismo al constructor del iterador para establecer un vínculo entre ellos.
5. Repasa el código cliente para sustituir todo el código de recorrido de la colección por el uso de iteradores. El cliente busca un nuevo objeto iterador cada vez que necesita recorrer los elementos de la colección.

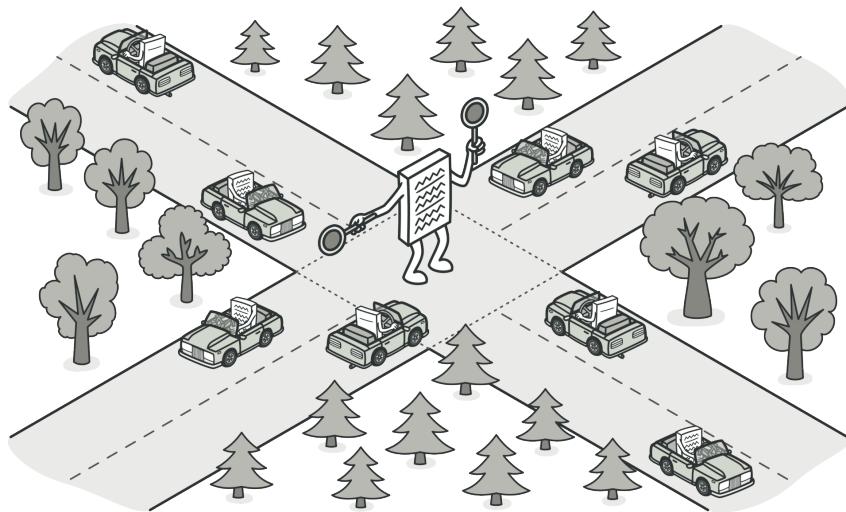
⚠️ Pros y contras

- ✓ *Principio de responsabilidad única.* Puedes limpiar el código cliente y las colecciones extrayendo algoritmos de recorrido voluminosos y colocándolos en clases independientes.
- ✓ *Principio de abierto/cerrado.* Puedes implementar nuevos tipos de colecciones e iteradores y pasarlo al código existente sin descomponer nada.
- ✓ Puedes recorrer la misma colección en paralelo porque cada objeto iterador contiene su propio estado de iteración.

- ✓ Por la misma razón, puedes retrasar una iteración y continuar cuando sea necesario.
- ✗ Aplicar el patrón puede resultar excesivo si tu aplicación funciona únicamente con colecciones sencillas.
- ✗ Utilizar un iterador puede ser menos eficiente que recorrer directamente los elementos de algunas colecciones especializadas.

↔ Relaciones con otros patrones

- Puedes utilizar **Iteradores** para recorrer árboles **Composite**.
- 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.
- Puedes usar **Memento** junto con **Iterator** para capturar el estado de la iteración actual y reanudarla si fuera necesario.
- Puedes utilizar **Visitor** junto con **Iterator** para recorrer una estructura de datos compleja y ejecutar alguna operación sobre sus elementos, incluso aunque todos tengan clases distintas.



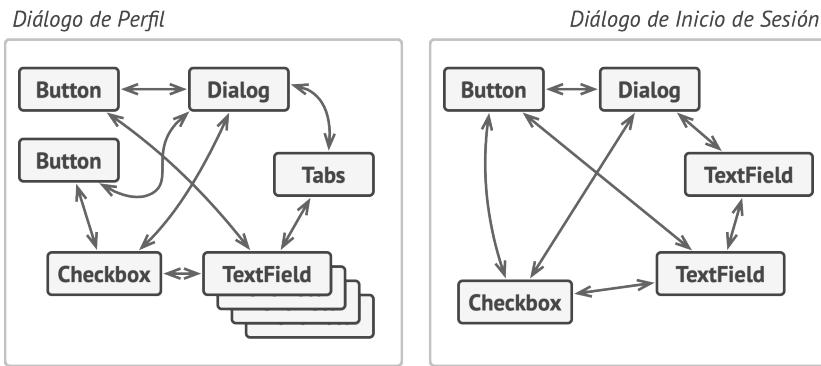
MEDIATOR

También llamado: Mediator, Intermediary, Controller

Mediator es un patrón de diseño de comportamiento que te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

(:() Problema

Digamos que tienes un diálogo para crear y editar perfiles de cliente. Consiste en varios controles de formulario, como campos de texto, casillas, botones, etc.



Las relaciones entre los elementos de la interfaz de usuario pueden volverse caóticas cuando la aplicación crece.

Algunos de los elementos del formulario pueden interactuar con otros. Por ejemplo, al seleccionar la casilla “tengo un perro” puede aparecer un campo de texto oculto para introducir el nombre del perro. Otro ejemplo es el botón de envío que tiene que validar los valores de todos los campos antes de guardar la información.



Los elementos pueden tener muchas relaciones con otros elementos. Por eso, los cambios en algunos elementos pueden afectar a los demás.

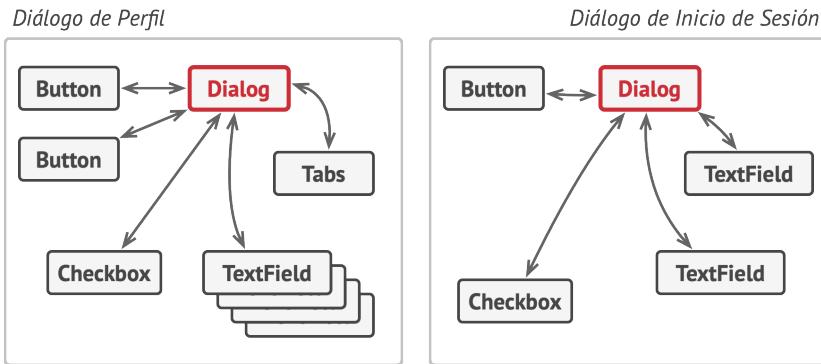
Al implementar esta lógica directamente dentro del código de los elementos del formulario, haces que las clases de estos elementos sean mucho más difíciles de reutilizar en otros formularios de la aplicación. Por ejemplo, no podrás utilizar la clase de la casilla dentro de otro formulario porque está acoplada al campo de texto del perro. O bien podrás utilizar todas las clases implicadas en representar el formulario de perfil, o no podrás usar ninguna en absoluto.

😊 Solución

El patrón Mediator sugiere que detengas toda comunicación directa entre los componentes que quieras hacer independientes entre sí. En lugar de ello, estos componentes deberán colaborar indirectamente, invocando un objeto mediador especial que redireccione las llamadas a los componentes adecuados. Como resultado, los componentes dependen únicamente de una sola clase mediadora, en lugar de estar acoplados a decenas de sus colegas.

En nuestro ejemplo del formulario de edición de perfiles, la propia clase de diálogo puede actuar como mediadora. Lo más

probable es que la clase de diálogo conozca ya todos sus subelementos, por lo que ni siquiera será necesario que introduzcas nuevas dependencias en esta clase.



Los elementos UI deben comunicarse indirectamente, a través del objeto mediador.

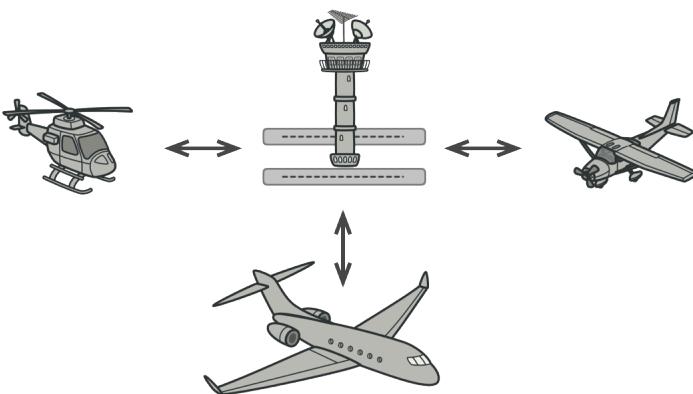
El cambio más significativo lo sufren los propios elementos del formulario. Pensemos en el botón de envío. Antes, cada vez que un usuario hacía clic en el botón, tenía que validar los valores de todos los elementos individuales del formulario. Ahora su único trabajo consiste en notificar al diálogo acerca del clic. Al recibir esta notificación, el propio diálogo realiza las validaciones o pasa la tarea a los elementos individuales. De este modo, en lugar de estar ligado a una docena de elementos del formulario, el botón solo es dependiente de la clase diálogo.

Puedes ir más lejos y reducir en mayor medida la dependencia extrayendo la interfaz común para todos los tipos de diálogo. La interfaz declarará el método de notificación que pueden uti-

lizar todos los elementos del formulario para notificar al diálogo sobre los eventos que le suceden a estos elementos. Por lo tanto, ahora nuestro botón de envío debería poder funcionar con cualquier diálogo que implemente esa interfaz.

De este modo, el patrón Mediator te permite encapsular una compleja red de relaciones entre varios objetos dentro de un único objeto mediador. Cuantas menos dependencias tenga una clase, más fácil es modificar, extender o reutilizar esa clase.

Analogía en el mundo real



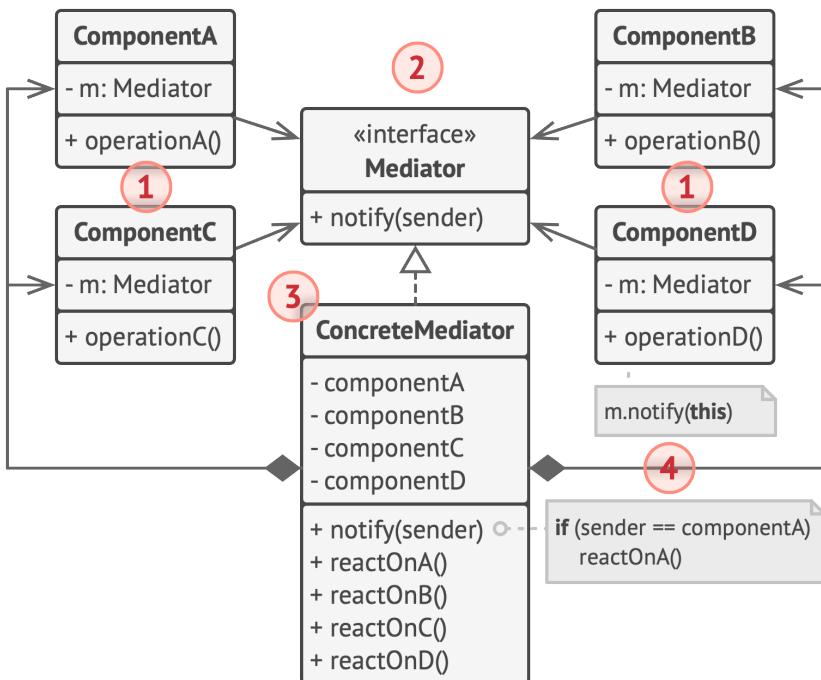
Los pilotos de aviones no hablan directamente entre sí para decidir quién es el siguiente en aterrizar su avión. Todas las comunicaciones pasan por la torre de control.

Los pilotos de los aviones que llegan o salen del área de control del aeropuerto no se comunican directamente entre sí. En lugar de eso, hablan con un controlador de tráfico aéreo, que está sentado en una torre alta cerca de la pista de aterrizaje.

Sin el controlador de tráfico aéreo, los pilotos tendrían que ser conscientes de todos los aviones en las proximidades del aeropuerto y discutir las prioridades de aterrizaje con un comité de decenas de otros pilotos. Probablemente, esto provocaría que las estadísticas de accidentes aéreos se dispararan.

La torre no necesita controlar el vuelo completo. Sólo existe para imponer límites en el área de la terminal porque el número de actores implicados puede resultar difícil de gestionar para un piloto.

Estructura

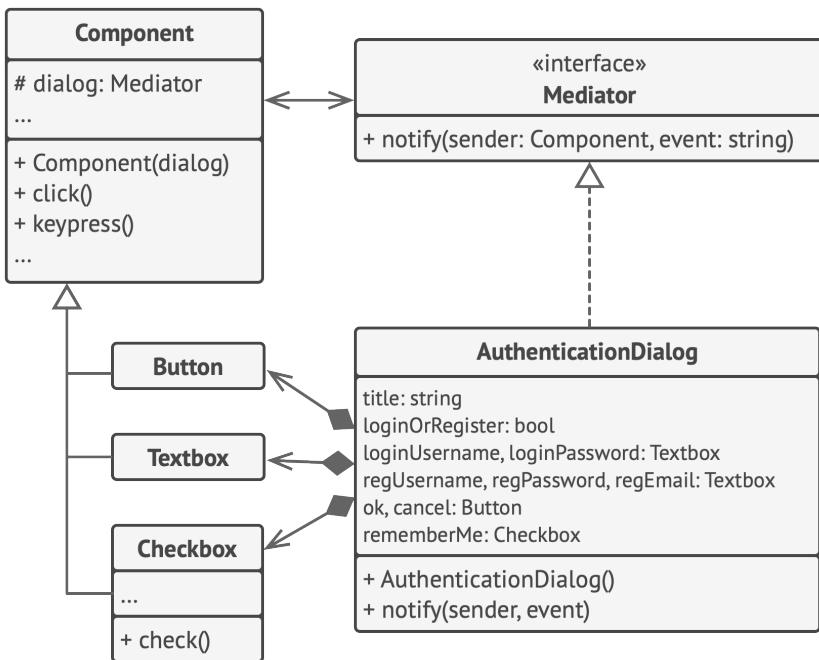


1. Los **Componentes** son varias clases que contienen cierta lógica de negocio. Cada componente tiene una referencia a una interfaz mediadora, declarada con su tipo. El componente no conoce la clase de la interfaz mediadora, por lo que puedes reutilizarlo en otros programas vinculándolo a una mediadora diferente.
2. La interfaz **Mediadora** declara métodos de comunicación con los componentes, que normalmente incluyen un único método de notificación. Los componentes pueden pasar cualquier contexto como argumentos de este método, incluyendo sus propios objetos, pero sólo de tal forma que no haya acoplamiento entre un componente receptor y la clase del emisor.
3. Los **Mediadores Concretos** encapsulan las relaciones entre varios componentes. Los mediadores concretos a menudo mantienen referencias a todos los componentes que gestionan y en ocasiones gestionan incluso su ciclo de vida.
4. Los componentes no deben conocer otros componentes. Si le sucede algo importante a un componente, o dentro de él, sólo debe notificar a la interfaz mediadora. Cuando la mediadora recibe la notificación, puede identificar fácilmente al emisor, lo cual puede ser suficiente para decidir qué componente debe activarse en respuesta.

Desde la perspectiva de un componente, todo parece una caja negra. El emisor no sabe quién acabará gestionando su solicitud, y el receptor no sabe quién envió la solicitud.

Pseudocódigo

En este ejemplo, el patrón **Mediator** te ayuda a eliminar dependencias mutuas entre varias clases UI: botones, casillas y etiquetas de texto.



Estructura de las clases de diálogo UI.

Un elemento activado por un usuario, no se comunica directamente con otros elementos, aunque parezca que debería. En lugar de eso, el elemento solo necesita dar a conocer el evento al mediador, pasando la información contextual junto a la notificación.

En este ejemplo, el diálogo de autenticación actúa como mediador. Sabe cómo deben colaborar los elementos concretos y facilita su comunicación indirecta. Al recibir una notificación sobre un evento, el diálogo decide qué elemento debe encargarse del evento y redirige la llamada en consecuencia.

```
1 // La interfaz mediadora declara un método utilizado por los
2 // componentes para notificar al mediador sobre varios eventos.
3 // El mediador puede reaccionar a estos eventos y pasar la
4 // ejecución a otros componentes.
5 interface Mediator is
6     method notify(sender: Component, event: string)
7
8
9 // La clase concreta mediadora. La red entrecruzada de
10 // conexiones entre componentes individuales se ha desenredado y
11 // se ha colocado dentro de la mediadora.
12 class AuthenticationDialog implements Mediator is
13     private field title: string
14     private field loginOrRegisterChkBx: Checkbox
15     private field loginUsername, loginPassword: Textbox
16     private field registrationUsername, registrationPassword,
17             registrationEmail: Textbox
18     private field okBtn, cancelBtn: Button
19
20 constructor AuthenticationDialog() is
21     // Crea todos los objetos del componente y pasa el
22     // mediador actual a sus constructores para establecer
23     // vínculos.
24
25 // Cuando sucede algo con un componente, notifica al
```

```
26 // mediador, que al recibir la notificación, puede hacer
27 // algo por su cuenta o pasar la solicitud a otro
28 // componente.
29 method notify(sender, event) is
30   if (sender == loginOrRegisterChkBx and event == "check")
31     if (loginOrRegisterChkBx.checked)
32       title = "Log in"
33       // 1. Muestra los componentes del formulario de
34       // inicio de sesión.
35       // 2. Esconde los componentes del formulario de
36       // registro.
37   else
38     title = "Register"
39     // 1. Muestra los componentes del formulario de
40     // registro.
41     // 2. Esconde los componentes del formulario de
42     // inicio de sesión.
43
44   if (sender == okBtn && event == "click")
45     if (loginOrRegister.checked)
46       // Intenta encontrar un usuario utilizando las
47       // credenciales de inicio de sesión.
48     if (!found)
49       // Muestra un mensaje de error sobre el
50       // campo de inicio de sesión.
51   else
52     // 1. Crea una cuenta de usuario utilizando
53     // información de los campos de registro.
54     // 2. Ingresá a ese usuario.
55     // ...
56
57
```

```
58 // Los componentes se comunican con un mediador utilizando la
59 // interfaz mediadora. Gracias a ello, puedes utilizar los
60 // mismos componentes en otros contextos vinculándolos con
61 // diferentes objetos mediadores.
62 class Component is
63     field dialog: Mediator
64
65     constructor Component(dialog) is
66         this.dialog = dialog
67
68     method click() is
69         dialog.notify(this, "click")
70
71     method keypress() is
72         dialog.notify(this, "keypress")
73
74 // Los componentes concretos no hablan entre sí. Sólo tienen un
75 // canal de comunicación, que es el envío de notificaciones al
76 // mediador.
77 class Button extends Component is
78     // ...
79
80 class Textbox extends Component is
81     // ...
82
83 class Checkbox extends Component is
84     method check() is
85         dialog.notify(this, "check")
86     // ...
```

Aplicabilidad

-  **Utiliza el patrón Mediator cuando resulte difícil cambiar algunas de las clases porque están fuertemente acopladas a un puñado de otras clases.**
-  El patrón te permite extraer todas las relaciones entre clases dentro de una clase separada, aislando cualquier cambio en un componente específico, del resto de los componentes.
-  **Utiliza el patrón cuando no puedas reutilizar un componente en un programa diferente porque sea demasiado dependiente de otros componentes.**
-  Una vez aplicado el patrón Mediator, los componentes individuales no conocen los otros componentes. Todavía pueden comunicarse entre sí, aunque indirectamente, a través del objeto mediador. Para reutilizar un componente en una aplicación diferente, debes darle una nueva clase mediadora.
-  **Utiliza el patrón Mediator cuando te encuentres creando cientos de subclases de componente sólo para reutilizar un comportamiento básico en varios contextos.**
-  Debido a que todas las relaciones entre componentes están contenidas dentro del mediador, resulta fácil definir formas totalmente nuevas de colaboración entre estos componentes introduciendo nuevas clases mediadoras, sin tener que cambiar los propios componentes.

Cómo implementarlo

1. Identifica un grupo de clases fuertemente acopladas que se beneficiarían de ser más independientes (p. ej., para un mantenimiento más sencillo o una reutilización más simple de esas clases).
2. Declara la interfaz mediadora y describe el protocolo de comunicación deseado entre mediadores y otros varios componentes. En la mayoría de los casos, un único método para recibir notificaciones de los componentes es suficiente.

Esta interfaz es fundamental cuando quieras reutilizar las clases del componente en distintos contextos. Siempre y cuando el componente trabaje con su mediador a través de la interfaz genérica, podrás vincular el componente con una implementación diferente del mediador.

3. Implementa la clase concreta mediadora. Esta clase se beneficiará de almacenar referencias a todos los componentes que gestiona.
4. Puedes ir más lejos y hacer la interfaz mediadora responsable de la creación y destrucción de objetos del componente. Tras esto, la mediadora puede parecerse a una **fábrica** o una **fachada**.
5. Los componentes deben almacenar una referencia al objeto mediador. La conexión se establece normalmente en el con-

structor del componente, donde un objeto mediador se pasa como argumento.

6. Cambia el código de los componentes de forma que invoquen el método de notificación del mediador en lugar de los métodos de otros componentes. Extrae el código que implique llamar a otros componentes dentro de la clase mediadora. Ejecuta este código cuando el mediador reciba notificaciones de ese componente.

Pros y contras

- ✓ *Principio de responsabilidad única.* Puedes extraer las comunicaciones entre varios componentes dentro de un único sitio, haciéndolo más fácil de comprender y mantener.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevos mediadores sin tener que cambiar los propios componentes.
- ✓ Puedes reducir el acoplamiento entre varios componentes de un programa.
- ✓ Puedes reutilizar componentes individuales con mayor facilidad.
- Con el tiempo, un mediador puede evolucionar a un **objeto todopoderoso.**



MEMENTO

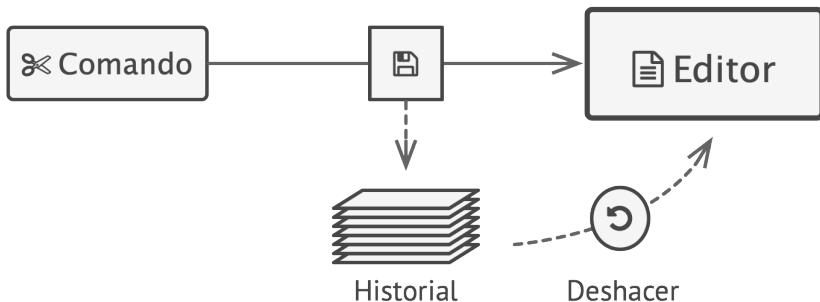
También llamado: Recuerdo, Instantánea, Snapshot

Memento es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

Problema

Imagina que estás creando una aplicación de edición de texto. Además de editar texto, tu programa puede formatearlo, así como insertar imágenes en línea, etc.

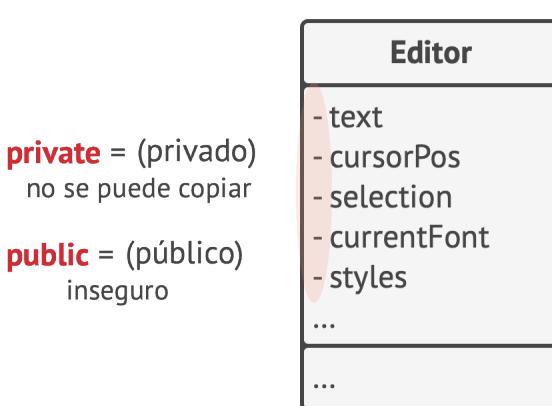
En cierto momento, decides permitir a los usuarios deshacer cualquier operación realizada en el texto. Esta función se ha vuelto tan habitual en los últimos años que hoy en día todo el mundo espera que todas las aplicaciones la tengan. Para la implementación eliges la solución directa. Antes de realizar cualquier operación, la aplicación registra el estado de todos los objetos y lo guarda en un almacenamiento. Más tarde, cuando un usuario decide revertir una acción, la aplicación extrae la última *instantánea* del historial y la utiliza para restaurar el estado de todos los objetos.



Antes de ejecutar una operación, la aplicación guarda una instantánea del estado de los objetos, que más tarde se puede utilizar para restaurar objetos a su estado previo.

Pensemos en estas instantáneas de estado. ¿Cómo producirías una, exactamente? Probablemente tengas que recorrer todos los campos de un objeto y copiar sus valores en el almacenamiento. Sin embargo, esto sólo funcionará si el objeto tiene unas restricciones bastante laxas al acceso a sus contenidos. Lamentablemente, la mayoría de objetos reales no permite a otros asomarse a su interior fácilmente, y esconden todos los datos significativos en campos privados.

Ignora ese problema por ahora y asumamos que nuestros objetos se comportan como hippies: prefieren relaciones abiertas y mantienen su estado público. Aunque esta solución resolvería el problema inmediato y te permitiría producir instantáneas de estados de objetos a voluntad, sigue teniendo algunos inconvenientes serios. En el futuro, puede que decidas refactorizar algunas de las clases editoras, o añadir o eliminar algunos de los campos. Parece fácil, pero esto también exige cambiar las clases responsables de copiar el estado de los objetos afectados.



¿Cómo hacer una copia del estado privado del objeto?

Pero aún hay más. Pensemos en las propias “instantáneas” del estado del editor. ¿Qué datos contienen? Como mínimo, deben contener el texto, las coordenadas del cursor, la posición actual de desplazamiento, etc. Para realizar una instantánea debes recopilar estos valores y meterlos en algún tipo de contenedor.

Probablemente almacenarás muchos de estos objetos de contenedor dentro de una lista que represente el historial. Por lo tanto, probablemente los contenedores acaben siendo objetos de una clase. La clase no tendrá apenas métodos, pero sí muchos campos que reflejen el estado del editor. Para permitir que otros objetos escriban y lean datos a y desde una instantánea, es probable que tengas que hacer sus campos públicos. Esto expondrá todos los estados del editor, privados o no. Otras clases se volverán dependientes de cada pequeño cambio en la clase de la instantánea, que de otra forma ocurri-

ría dentro de campos y métodos privados sin afectar a clases externas.

Parece que hemos llegado a un callejón sin salida: o bien expones todos los detalles internos de las clases, haciéndolas demasiado frágiles, o restringes el acceso a su estado, haciendo imposible producir instantáneas. ¿Hay alguna otra forma de implementar el "deshacer"?

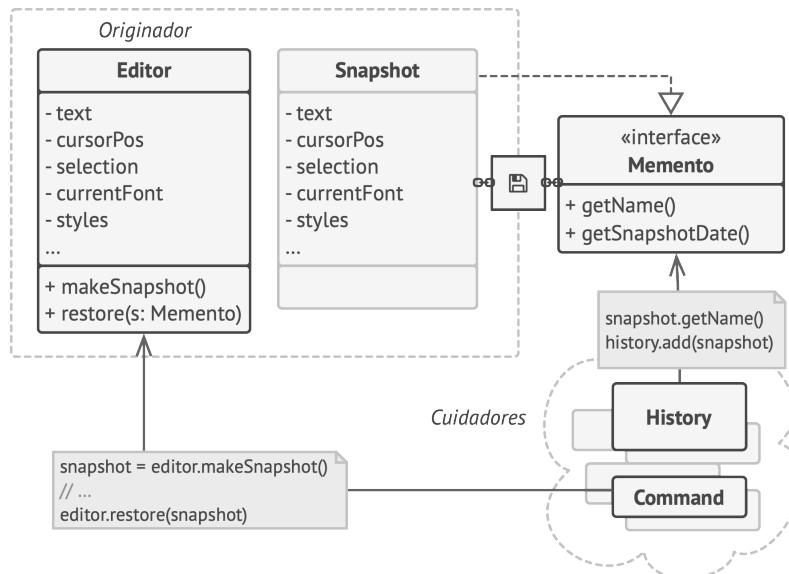
Solución

Todos los problemas que hemos experimentado han sido provocados por una encapsulación fragmentada. Algunos objetos intentan hacer más de lo que deben. Para recopilar los datos necesarios para realizar una acción, invaden el espacio privado de otros objetos en lugar de permitir a esos objetos realizar la propia acción.

El patrón Memento delega la creación de instantáneas de estado al propietario de ese estado, el objeto *originador*. Por lo tanto, en lugar de que haya otros objetos intentando copiar el estado del editor desde el “exterior”, la propia clase editorial puede hacer la instantánea, ya que tiene pleno acceso a su propio estado.

El patrón sugiere almacenar la copia del estado del objeto en un objeto especial llamado *memento*. Los contenidos del memento no son accesibles para ningún otro objeto excepto el que lo produjo. Otros objetos deben comunicarse con memen-

tos utilizando una interfaz limitada que pueda permitir extraer los metadatos de la instantánea (tiempo de creación, el nombre de la operación realizada, etc.), pero no el estado del objeto original contenido en la instantánea.



El originador tiene pleno acceso al memento, mientras que el cuidador sólo puede acceder a los metadatos.

Una política tan restrictiva te permite almacenar mementos dentro de otros objetos, normalmente llamados *cuidadores*. Debido a que el cuidador trabaja con el memento únicamente a través de la interfaz limitada, no puede manipular el estado almacenado dentro del memento. Al mismo tiempo, el originador tiene acceso a todos los campos dentro del memento, permitiéndole restaurar su estado previo a voluntad.

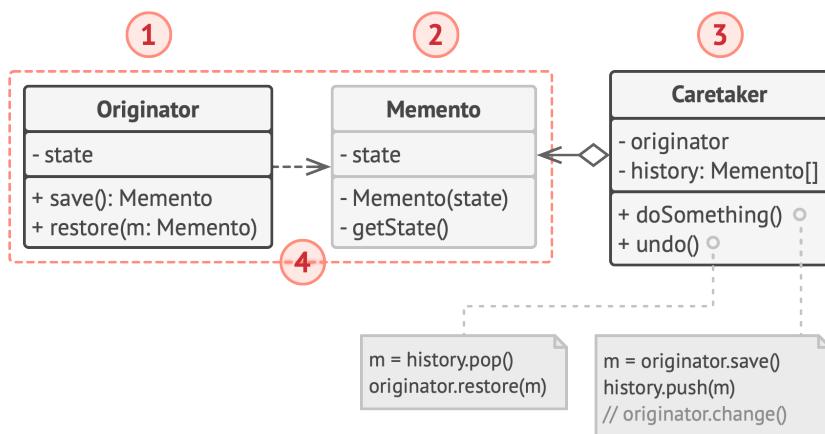
En nuestro ejemplo del editor de texto, podemos crear una clase separada de historial que actúe como cuidadora. Una pila de mementos almacenados dentro de la cuidadora crecerá cada vez que el editor vaya a ejecutar una operación. Puedes incluso presentar esta pila dentro de la UI de la aplicación, mostrando a un usuario el historial de operaciones previamente realizadas.

Cuando un usuario activa la función Deshacer, el historial toma el memento más reciente de la pila y lo pasa de vuelta al editor, solicitando una restauración. Debido a que el editor tiene pleno acceso al memento, cambia su propio estado con los valores tomados del memento.

Estructura

Implementación basada en clases anidadas

La implementación clásica del patrón se basa en el soporte de clases anidadas, disponible en varios lenguajes de programación populares (como C++, C# y Java).



1. La clase **Originadora** puede producir instantáneas de su propio estado, así como restaurar su estado a partir de instantáneas cuando lo necesita.
2. El **Memento** es un objeto de valor que actúa como instantánea del estado del originador. Es práctica común hacer el memento inmutable y pasarle los datos solo una vez, a través del constructor.

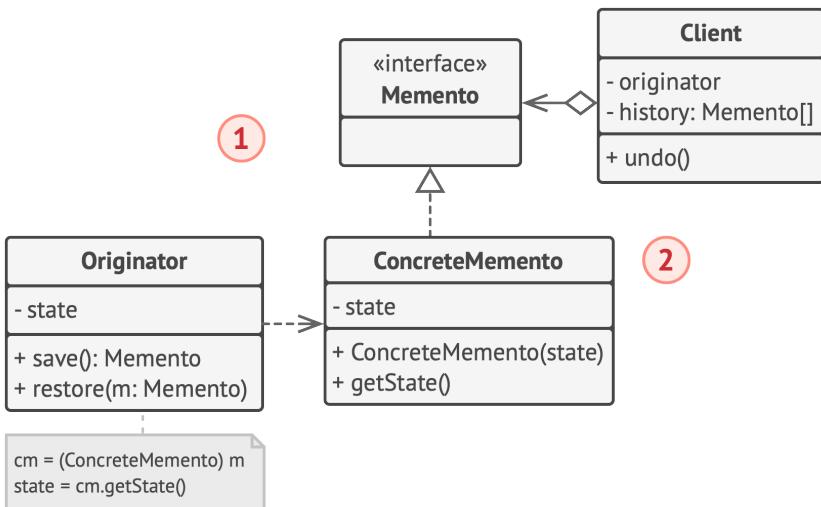
3. La **Cuidadora** sabe no solo “cuándo” y “por qué” capturar el estado de la originadora, sino también cuándo debe restaurarse el estado.

Una cuidadora puede rastrear el historial de la originadora almacenando una pila de mementos. Cuando la originadora deba retroceder en el historial, la cuidadora extraerá el memento de más arriba de la pila y lo pasará al método de restauración de la originadora.

4. En esta implementación, la clase memento se anida dentro de la originadora. Esto permite a la originadora acceder a los campos y métodos de la clase memento, aunque se declaren privados. Por otro lado, la cuidadora tiene un acceso muy limitado a los campos y métodos de la clase memento, lo que le permite almacenar mementos en una pila pero no alterar su estado.

Implementación basada en una interfaz intermedia

Existe una implementación alternativa, adecuada para lenguajes de programación que no soportan clases anidadas (sí, PHP, estoy hablando de ti).

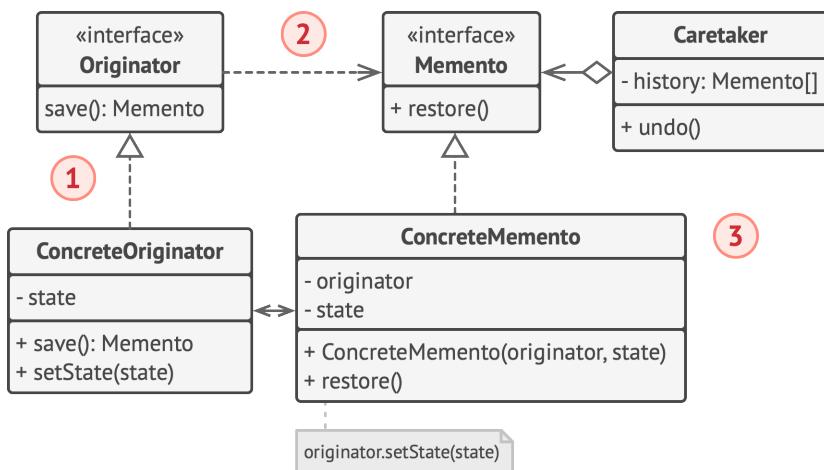


1. En ausencia de clases anidadas, puedes restringir el acceso a los campos de la clase memento estableciendo una convención de que las cuidadoras sólo pueden trabajar con una memento a través de una interfaz intermediaria explícitamente declarada, que sólo declarará métodos relacionados con los metadatos del memento.
2. Por otro lado, las originadoras pueden trabajar directamente con un objeto memento, accediendo a campos y métodos declarados en la clase memento. El inconveniente de esta solu-

ción es que debes declarar públicos todos los miembros de la clase memento.

Implementación con una encapsulación más estricta

Existe otra implementación que resulta útil cuando no queremos dejar la más mínima opción a que otras clases accedan al estado de la originadora a través del memento.

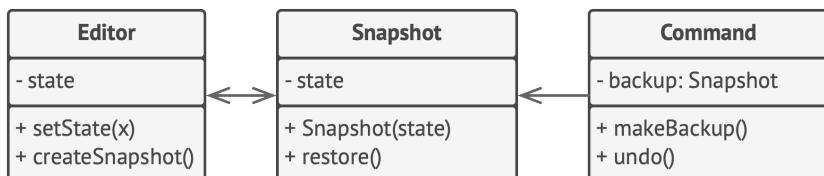


1. Esta implementación permite tener varios tipos de originadoras y mementos. Cada originadora trabaja con una clase memento correspondiente. Ninguna de las dos expone su estado a nadie.
2. Las cuidadoras tienen ahora explícitamente restringido cambiar el estado almacenado en los mementos. Además, la clase cuidadora se vuelve independiente de la originadora porque el método de restauración se define ahora en la clase memento.

3. Cada memento queda vinculado a la originadora que lo produce. La originadora se pasa al constructor del memento, junto con los valores de su estado. Gracias a la estrecha relación entre estas clases, un memento puede restaurar el estado de su originadora, siempre que esta última haya definido los modificadores (setters) adecuados.

Pseudocódigo

Este ejemplo utiliza el patrón Memento junto al patrón **Command** para almacenar instantáneas del estado complejo del editor de texto y restaurar un estado previo a partir de estas instantáneas cuando sea necesario.



Guardar instantáneas del estado del editor de texto.

Los objetos de comando actúan como cuidadores. Buscan el memento del editor antes de ejecutar operaciones relacionadas con los comandos. Cuando un usuario intenta deshacer el comando más reciente, el editor puede utilizar el memento almacenado en ese comando para revertirse a sí mismo al estado previo.

La clase memento no declara ningún campo, consultor (getter) o modificador (setter) como público. Por lo tanto, ningún obje-

to puede alterar sus contenidos. Los mementos se vinculan al objeto del editor que los creó. Esto permite a un memento restaurar el estado del editor vinculado pasando los datos a través de modificadores en el objeto editor. Ya que los mementos están vinculados a objetos de editor específicos, puedes hacer que tu aplicación soporte varias ventanas de editor independientes con una pila centralizada para deshacer.

```
1 // El originador contiene información importante que puede
2 // cambiar con el paso del tiempo. También define un método para
3 // guardar su estado dentro de un memento, y otro método para
4 // restaurar el estado a partir de él.
5 class Editor is
6     private field text, curX, curY, selectionWidth
7
8     method setText(text) is
9         this.text = text
10
11    method setCursor(x, y) is
12        this.curX = curX
13        this.curY = curY
14
15    method setSelectionWidth(width) is
16        this.selectionWidth = width
17
18    // Guarda el estado actual dentro de un memento.
19    method createSnapshot():Snapshot is
20        // El memento es un objeto inmutable; ese es el motivo
21        // por el que el originador pasa su estado a los
22        // parámetros de su constructor.
23        return new Snapshot(this, text, curX, curY, selectionWidth)
```

```
24
25 // La clase memento almacena el estado pasado del editor.
26 class Snapshot is
27     private field editor: Editor
28     private field text, curX, curY, selectionWidth
29
30 constructor Snapshot(editor, text, curX, curY, selectionWidth) is
31     this.editor = editor
32     this.text = text
33     this.curX = curX
34     this.curY = curY
35     this.selectionWidth = selectionWidth
36
37 // En cierto punto, puede restaurarse un estado previo del
38 // editor utilizando un objeto memento.
39 method restore() is
40     editor.setText(text)
41     editor.setCursor(curX, curY)
42     editor.setSelectionWidth(selectionWidth)
43
44 // Un objeto de comando puede actuar como cuidador. En este
45 // caso, el comando obtiene un memento justo antes de cambiar el
46 // estado del originador. Cuando se solicita deshacer, restaura
47 // el estado del originador a partir del memento.
48 class Command is
49     private field backup: Snapshot
50
51 method makeBackup() is
52     backup = editor.createSnapshot()
53
54 method undo() is
55     if (backup != null)
```

```
56     backup.restore()  
57     // ...
```

💡 Aplicabilidad

- ⚡ Utiliza el patrón Memento cuando quieras producir instantáneas del estado del objeto para poder restaurar un estado previo del objeto.
- ⚡ El patrón Memento te permite realizar copias completas del estado de un objeto, incluyendo campos privados, y almacenarlos independientemente del objeto. Aunque la mayoría de la gente recuerda este patrón gracias al caso de la función Deshacer, también es indispensable a la hora de tratar con transacciones (por ejemplo, si debes volver atrás sobre un error en una operación).
- ⚡ Utiliza el patrón cuando el acceso directo a los campos, consumidores o modificadores del objeto viole su encapsulación.
- ⚡ El Memento hace al propio objeto responsable de la creación de una instantánea de su estado. Ningún otro objeto puede leer la instantánea, lo que hace que los datos del estado del objeto original queden seguros.

Cómo implementarlo

1. Determina qué clase jugará el papel de la originadora. Es importante saber si el programa utiliza un objeto central de este tipo o varios más pequeños.
2. Crea la clase memento. Uno a uno, declara un grupo de campos que reflejen los campos declarados dentro de la clase originadora.
3. Haz la clase memento inmutable. Una clase memento debe aceptar los datos sólo una vez, a través del constructor. La clase no debe tener modificadores.
4. Si tu lenguaje de programación soporta clases anidadas, anida la clase memento dentro de la originadora. Si no es así, extrae una interfaz en blanco de la clase memento y haz que el resto de objetos la utilicen para remitirse a ella. Puedes añadir operaciones de metadatos a la interfaz, pero nada que exponga el estado de la originadora.
5. Añade un método para producir mementos a la clase originadora. La originadora debe pasar su estado a la clase memento a través de uno o varios argumentos del constructor del memento.

El tipo de retorno del método debe ser del mismo que la interfaz que extrajiste en el paso anterior (asumiendo que lo hi-

ciste). Básicamente, el método productor del memento debe trabajar directamente con la clase memento.

6. Añade un método para restaurar el estado del originador a su clase. Debe aceptar un objeto memento como argumento. Si extrajiste una interfaz en el paso previo, haz que sea el tipo del parámetro. En este caso, debes especificar el tipo del objeto entrante al de la clase memento, ya que la originadora necesita pleno acceso a ese objeto.
7. La cuidadora, independientemente de que represente un objeto de comando, un historial, o algo totalmente diferente, debe saber cuándo solicitar nuevos mementos de la originadora, cómo almacenarlos y cuándo restaurar la originadora con un memento particular.
8. El vínculo entre cuidadoras y originadoras puede moverse dentro de la clase memento. En este caso, cada memento debe conectarse a la originadora que lo creó. El método de restauración también se moverá a la clase memento. No obstante, todo esto sólo tendrá sentido si la clase memento está anidada dentro de la originadora o la clase originadora proporciona suficientes modificadores para sobrescribir su estado.

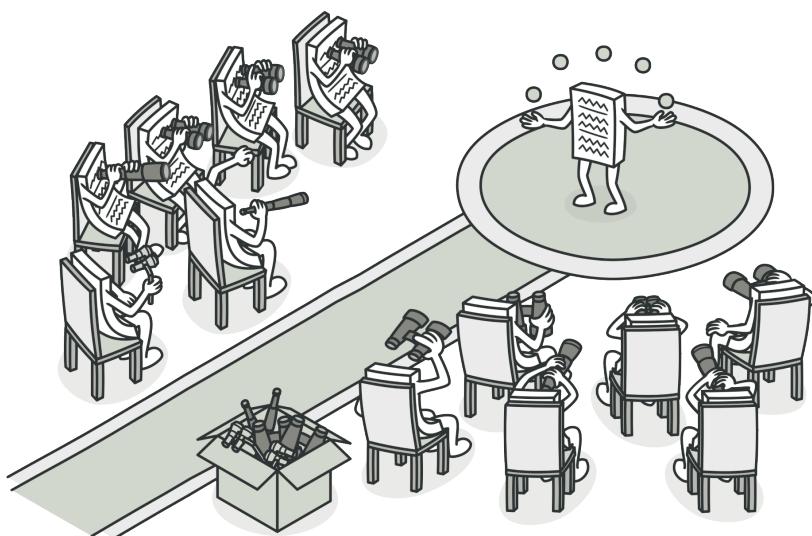
⚠️ Pros y contras

- ✓ Puedes producir instantáneas del estado del objeto sin violar su encapsulación.

- ✓ Puedes simplificar el código de la originadora permitiendo que la cuidadora mantenga el historial del estado de la originadora.
- ✗ La aplicación puede consumir mucha memoria RAM si los clientes crean mementos muy a menudo.
- ✗ Las cuidadoras deben rastrear el ciclo de vida de la originadora para poder destruir mementos obsoletos.
- ✗ La mayoría de los lenguajes de programación dinámicos, como PHP, Python y JavaScript, no pueden garantizar que el estado dentro del memento se mantenga intacto.

↔ Relaciones con otros patrones

- Puedes utilizar **Command** y **Memento** juntos cuando implementes “deshacer”. En este caso, los comandos son responsables de realizar varias operaciones sobre un objeto destino, mientras que los mementos guardan el estado de ese objeto justo antes de que se ejecute el comando.
- Puedes usar **Memento** junto con **Iterator** para capturar el estado de la iteración actual y reanudarla si fuera necesario.
- En ocasiones, **Prototype** puede ser una alternativa más simple al patrón **Memento**. Esto funciona si el objeto cuyo estado quieras almacenar en el historial es suficientemente sencillo y no tiene enlaces a recursos externos, o estos son fáciles de restablecer.



OBSERVER

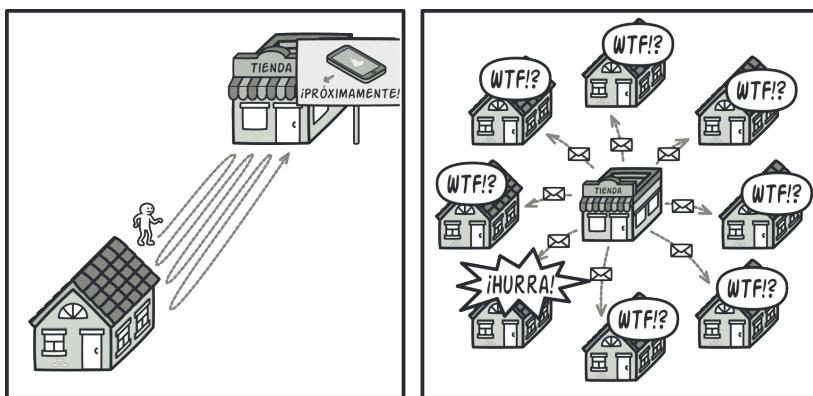
También llamado: Observador, Publicación-Suscripción, Modelo-patrón, Event-Subscriber, Listener

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

(:() Problema

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

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



Visita a la tienda vs. envío de spam

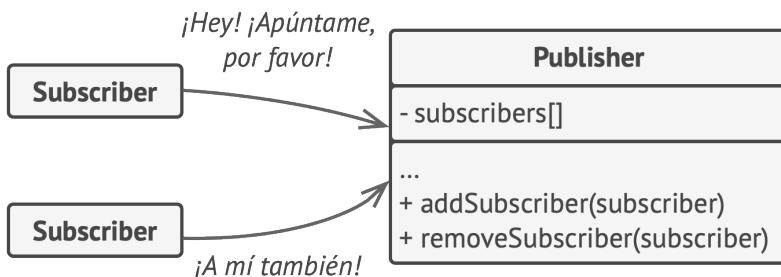
Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados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

Parece que nos encontramos ante un conflicto. O el cliente pierde tiempo comprobando la disponibilidad del producto, o bien la tienda desperdicia recursos notificando a los clientes equivocados.

😊 Solución

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

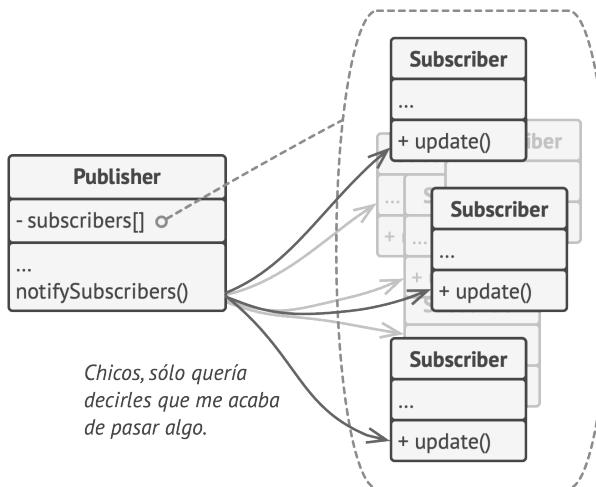
El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora. ¡No temas! No es tan complicado como parece. En realidad, este mecanismo consiste en: 1) un campo matriz para almacenar una lista de referencias a objetos suscriptores y 2) varios métodos públicos que permiten añadir suscriptores y eliminarlos de esa lista.



Un mecanismo de suscripción permite a los objetos individuales suscribirse a notificaciones de eventos.

Ahora, cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.

Las aplicaciones reales pueden tener decenas de clases suscriptoras diferentes interesadas en seguir los eventos de la misma clase notificadora. No querrás acoplar la notificadora a todas esas clases. Además, puede que no conozcas algunas de ellas de antemano si se supone que otras personas pueden utilizar tu clase notificadora.



El notificador notifica a los suscriptores invocando el método de notificación específico de sus objetos.

Por eso es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comunique con ellos a través de esa interfaz. Esta interfaz debe declarar el método de notificación junto con un grupo de pa-

rámetros que el notificador puede utilizar para pasar cierta información contextual con la notificación.

Si tu aplicación tiene varios tipos diferentes de notificadores y quieres hacer a tus suscriptores compatibles con todos ellos, puedes ir más allá y hacer que todos los notificadores sigan la misma interfaz. Esta interfaz sólo tendrá que describir algunos métodos de suscripción. La interfaz permitirá a los suscriptores observar los estados de los notificadores sin acoplarse a sus clases concretas.

🚗 Analogía en el mundo real

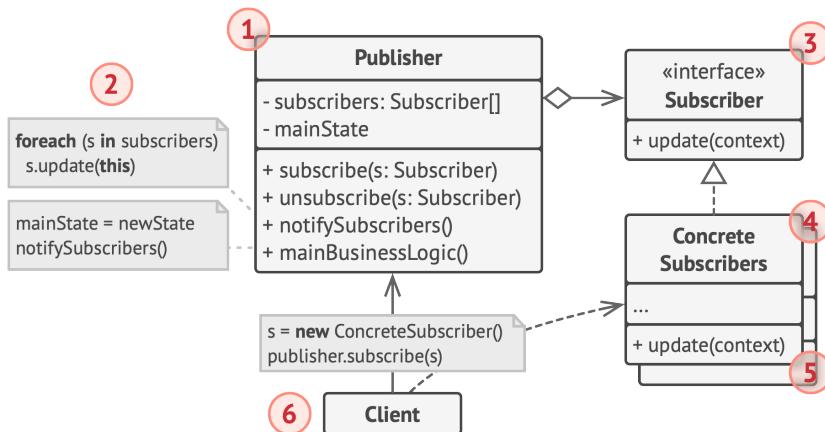


Suscripciones a revistas y periódicos.

Si te suscribes a un periódico o una revista, ya no necesitarás ir a la tienda a comprobar si el siguiente número está disponible. En lugar de eso, el notificador envía nuevos números directamente a tu buzón justo después de la publicación, o incluso antes.

El notificador mantiene una lista de suscriptores y sabe qué revistas les interesan. Los suscriptores pueden abandonar la lista en cualquier momento si quieren que el notificador deje de enviarles nuevos números.

Estructura



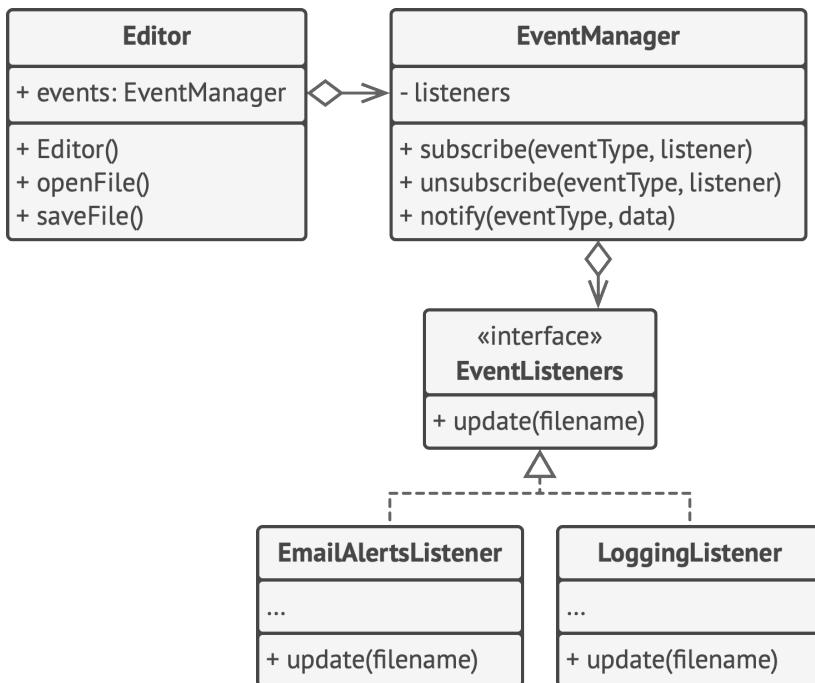
1. El **Notificador** envía eventos de interés a otros objetos. Esos eventos ocurren cuando el notificador cambia su estado o ejecuta algunos comportamientos. Los notificadores contienen una infraestructura de suscripción que permite a nuevos y antiguos suscriptores abandonar la lista.
2. Cuando sucede un nuevo evento, el notificador recorre la lista de suscripción e invoca el método de notificación declarado en la interfaz suscriptora en cada objeto suscriptor.
3. La interfaz **Suscriptora** declara la interfaz de notificación. En la mayoría de los casos, consiste en un único método

actualizar . El método puede tener varios parámetros que permitan al notificador pasar algunos detalles del evento junto a la actualización.

4. Los **Suscriptores Concretos** realizan algunas acciones en respuesta a las notificaciones emitidas por el notificador. Todas estas clases deben implementar la misma interfaz de forma que el notificador no esté acoplado a clases concretas.
5. Normalmente, los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por este motivo, a menudo los notificadores pasan cierta información de contexto como argumentos del método de notificación. El notificador puede pasarse a sí mismo como argumento, dejando que los suscriptores extraigan la información necesaria directamente.
6. El **Cliente** crea objetos tipo notificador y suscriptor por separado y después registra a los suscriptores para las actualizaciones del notificador.

Pseudocódigo

En este ejemplo, el patrón **Observer** permite al objeto editor de texto notificar a otros objetos tipo servicio sobre los cambios en su estado.



Notificar a objetos sobre eventos que suceden a otros objetos.

La lista de suscriptores se compila dinámicamente: los objetos pueden empezar o parar de escuchar notificaciones durante el tiempo de ejecución, dependiendo del comportamiento que deseas para tu aplicación.

En esta implementación, la clase editora no mantiene la lista de suscripción por sí misma. Delega este trabajo al objeto ayudante especial dedicado justo a eso. Puedes actualizar ese objeto para que sirva como despachador centralizado de eventos, dejando que cualquier objeto actúe como notificador.

Añadir nuevos suscriptores al programa no requiere cambios en clases notificadoras existentes, siempre y cuando trabajen con todos los suscriptores a través de la misma interfaz.

```
1 // La clase notificadora base incluye código de gestión de
2 // suscripciones y métodos de notificación.
3 class EventManager is
4     private field listeners: hash map of event types and listeners
5
6     method subscribe(eventType, listener) is
7         listeners.add(eventType, listener)
8
9     method unsubscribe(eventType, listener) is
10        listeners.remove(eventType, listener)
11
12    method notify(eventType, data) is
13        foreach (listener in listeners.of(eventType)) do
14            listener.update(data)
15
16    // El notificador concreto contiene lógica de negocio real, de
17    // interés para algunos suscriptores. Podemos derivar esta clase
18    // de la notificadora base, pero esto no siempre es posible en
19    // el mundo real porque puede que la notificadora concreta sea
20    // ya una subclase. En este caso, puedes modificar la lógica de
21    // la suscripción con composición, como hicimos aquí.
22 class Editor is
23     public field events: EventManager
24     private field file: File
25
26     constructor Editor() is
27         events = new EventManager()
```

```
28
29 // Los métodos de la lógica de negocio pueden notificar los
30 // cambios a los suscriptores.
31 method openFile(path) is
32     this.file = new File(path)
33     events.notify("open", file.name)
34
35 method saveFile() is
36     file.write()
37     events.notify("save", file.name)
38
39 // ...
40
41
42 // Aquí está la interfaz suscriptora. Si tu lenguaje de
43 // programación soporta tipos funcionales, puedes sustituir toda
44 // la jerarquía suscriptora por un grupo de funciones.
45
46
47 interface EventListener is
48     method update(filename)
49
50 // Los suscriptores concretos reaccionan a las actualizaciones
51 // emitidas por el notificador al que están unidos.
52 class LoggingListener implements EventListener is
53     private field log: File
54     private field message
55
56     constructor LoggingListener(log_filename, message) is
57         this.log = new File(log_filename)
58         this.message = message
59
```

```
60   method update(filename) is
61     log.write(replace('%s',filename,message))
62
63 class EmailAlertsListener implements EventListener is
64   private field email: string
65
66 constructor EmailAlertsListener(email, message) is
67   this.email = email
68   this.message = message
69
70 method update(filename) is
71   system.email(email, replace('%s',filename,message))
72
73
74 // Una aplicación puede configurar notificadores y suscriptores
75 // durante el tiempo de ejecución.
76 class Application is
77   method config() is
78     editor = new Editor()
79
80     logger = new LoggingListener(
81       "/path/to/log.txt",
82       "Someone has opened the file: %s")
83     editor.events.subscribe("open", logger)
84
85     emailAlerts = new EmailAlertsListener(
86       "admin@example.com",
87       "Someone has changed the file: %s")
88     editor.events.subscribe("save", emailAlerts)
```

Aplicabilidad

-  Utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.
-  Puedes experimentar este problema a menudo al trabajar con clases de la interfaz gráfica de usuario. Por ejemplo, si creas clases personalizadas de botón y quieras permitir al cliente colgar código cliente de tus botones para que se active cuando un usuario pulse un botón.

El patrón Observer permite que cualquier objeto que implemente la interfaz suscriptora pueda suscribirse a notificaciones de eventos en objetos notificadores. Puedes añadir el mecanismo de suscripción a tus botones, permitiendo a los clientes acoplar su código personalizado a través de clases suscriptoras personalizadas.

-  Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.
-  La lista de suscripción es dinámica, por lo que los suscriptores pueden unirse o abandonar la lista cuando lo deseen.



Cómo implementarlo

1. Repasa tu lógica de negocio e intenta dividirla en dos partes: la funcionalidad central, independiente del resto de código, actuará como notificador; el resto se convertirá en un grupo de clases suscriptoras.
2. Declara la interfaz suscriptora. Como mínimo, deberá declarar un único método `actualizar`.
3. Declara la interfaz notificadora y describe un par de métodos para añadir y eliminar de la lista un objeto suscriptor. Recuerda que los notificadores deben trabajar con suscriptores únicamente a través de la interfaz suscriptora.
4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción. Normalmente, este código tiene el mismo aspecto para todos los tipos de notificadores, por lo que el lugar obvio para colocarlo es en una clase abstracta derivada directamente de la interfaz notificadora. Los notificadores concretos extienden esa clase, heredando el comportamiento de suscripción.

Sin embargo, si estás aplicando el patrón a una jerarquía de clases existentes, considera una solución basada en la composición: coloca la lógica de la suscripción en un objeto separado y haz que todos los notificadores reales la utilicen.

5. Crea clases notificadoras concretas. Cada vez que suceda algo importante dentro de una notificadora, deberá notificar a todos sus suscriptores.
6. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas. La mayoría de las suscriptoras necesitarán cierta información de contexto sobre el evento, que puede pasarse como argumento del método de notificación.

Pero hay otra opción. Al recibir una notificación, el suscriptor puede extraer la información directamente de ella. En este caso, el notificador debe pasarse a sí mismo a través del método de actualización. La opción menos flexible es vincular un notificador con el suscriptor de forma permanente a través del constructor.

7. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

Δ Δ Pros y contras

- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevas clases suscriptoras sin tener que cambiar el código de la notificadora (y viceversa si hay una interfaz notificadora).
- ✓ Puedes establecer relaciones entre objetos durante el tiempo de ejecución.
- ✗ Los suscriptores son notificados en un orden aleatorio.

↔ Relaciones con otros patrones

- Chain of Responsibility, Command, Mediator y Observer abordan distintas formas de conectar emisores y receptores de solicitudes:
 - *Chain of Responsibility* pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la gestiona.
 - *Command* establece conexiones unidireccionales entre emisores y receptores.
 - *Mediator* elimina las conexiones directas entre emisores y receptores, forzándolos a comunicarse indirectamente a través de un objeto mediador.
 - *Observer* permite a los receptores suscribirse o darse de baja dinámicamente a la recepción de solicitudes.
- La diferencia entre Mediator y Observer a menudo resulta difusa. En la mayoría de los casos, puedes implementar uno de estos dos patrones; pero en ocasiones puedes aplicarlos ambos a la vez. Veamos cómo podemos hacerlo.

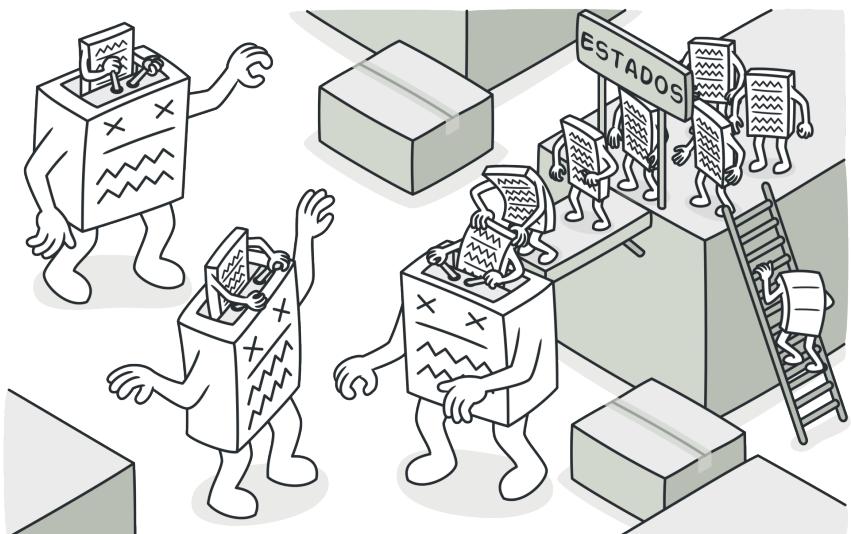
La meta principal del patrón *Mediator* consiste en eliminar las dependencias mutuas entre un grupo de componentes del sistema. En su lugar, estos componentes se vuelven dependientes de un único objeto mediador. La meta del patrón *Observer* es establecer conexiones dinámicas de un único sentido

entre objetos, donde algunos objetos actúan como subordinados de otros.

Hay una implementación popular del patrón *Mediator* que se basa en el *Observer*. El objeto mediador juega el papel de notificador, y los componentes actúan como suscriptores que se suscriben o se dan de baja de los eventos del mediador. Cuando se implementa el *Mediator* de esta forma, puede asemejarse mucho al *Observer*.

Cuando te sientas confundido, recuerda que puedes implementar el patrón Mediator de otras maneras. Por ejemplo, puedes vincular permanentemente todos los componentes al mismo objeto mediador. Esta implementación no se parece al *Observer*, pero aún así será una instancia del patrón Mediator.

Ahora, imagina un programa en el que todos los componentes se hayan convertido en notificadores, permitiendo conexiones dinámicas entre sí. No hay un objeto mediador centralizado, tan solo un grupo distribuido de observadores.



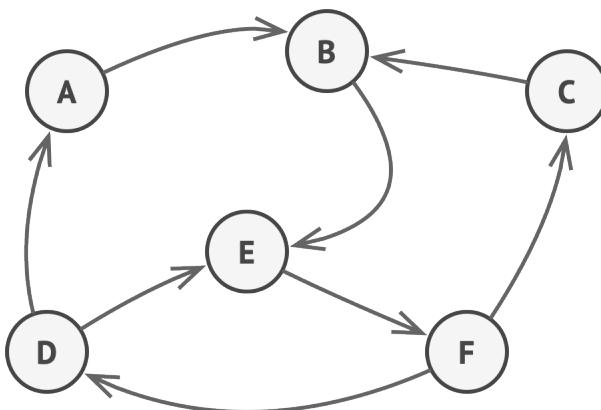
STATE

También llamado: Estado

State es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

Problema

El patrón State está estrechamente relacionado con el concepto de la Máquina de estados finitos.



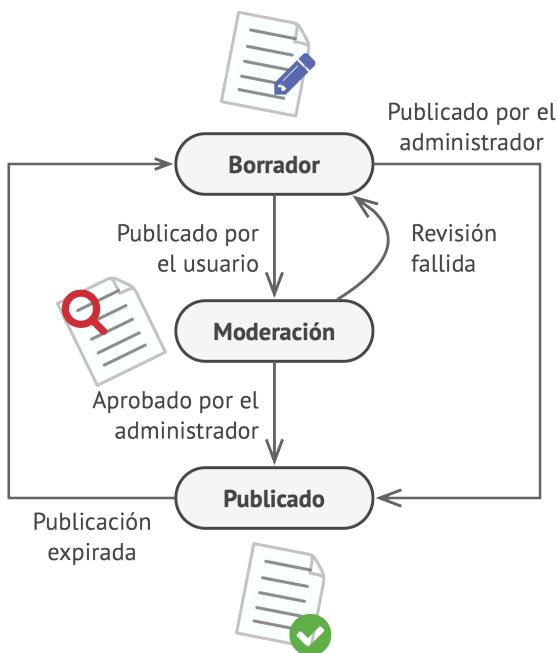
Máquina de estados finitos.

La idea principal es que, en cualquier momento dado, un programa puede encontrarse en un número *finito* de *estados*. Dentro de cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo de un estado actual, el programa puede cambiar o no a otros estados. Estas normas de cambio llamadas *transiciones* también son finitas y predeterminadas.

También puedes aplicar esta solución a los objetos. Imagina que tienes una clase `Documento`. Un documento puede encontrarse en uno de estos tres estados: `Borrador`, `Moderación`

y `Publicado`. El método `publicar` del documento funciona de forma ligeramente distinta en cada estado:

- En `Borrador`, mueve el documento a moderación.
- En `Moderación`, hace público el documento, pero sólo si el usuario actual es un administrador.
- En `Publicado`, no hace nada en absoluto.



Possibles estados y transiciones de un objeto de documento.

Las máquinas de estado se implementan normalmente con muchos operadores condicionales (`if` o `switch`) que seleccionan el comportamiento adecuado dependiendo del estado actual del objeto. Normalmente, este “estado” es tan solo

un grupo de valores de los campos del objeto. Aunque nunca hayas oído hablar de máquinas de estados finitos, probablemente hayas implementado un estado al menos alguna vez. ¿Te suena esta estructura de código?

```
1 class Document is
2     field state: string
3     // ...
4     method publish() is
5         switch (state)
6             "draft":
7                 state = "moderation"
8                 break
9             "moderation":
10                if (currentUser.role == 'admin')
11                    state = "published"
12                    break
13                "published":
14                    // No hacer nada.
15                    break
16        // ...
```

La mayor debilidad de una máquina de estado basada en condicionales se revela una vez que empezamos a añadir más y más estados y comportamientos dependientes de estados a la clase `Documento`. La mayoría de los métodos contendrán condicionales monstruosos que eligen el comportamiento adecuado de un método de acuerdo con el estado actual. Un código así es muy difícil de mantener, porque cualquier cam-

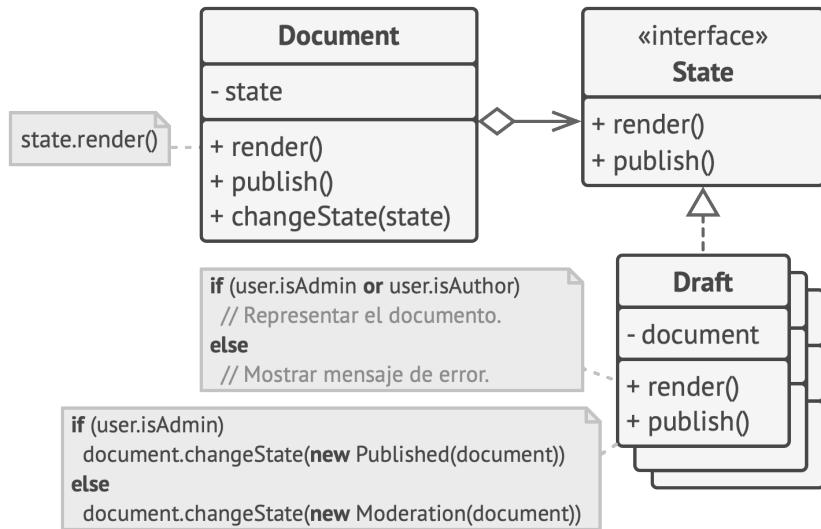
bio en la lógica de transición puede requerir cambiar los condicionales de estado de cada método.

El problema tiende a empeorar con la evolución del proyecto. Es bastante difícil predecir todos los estados y transiciones posibles en la etapa de diseño. Por ello, una máquina de estados esbelta, creada con un grupo limitado de condicionales, puede crecer hasta convertirse en un abotargado desastre con el tiempo.

Solución

El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.

En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado *contexto*, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.



Documento delega el trabajo a un objeto de estado.

Para la transición del contexto a otro estado, sustituye el objeto de estado activo por otro objeto que represente ese nuevo estado. Esto sólo es posible si todas las clases de estado siguen la misma interfaz y el propio contexto funciona con esos objetos a través de esa interfaz.

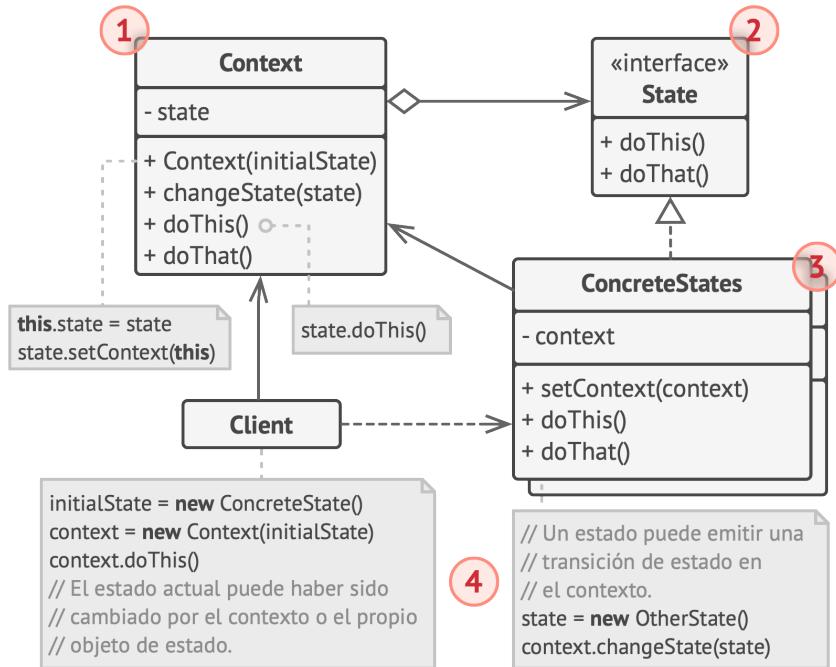
Esta estructura puede resultar similar al patrón **Strategy**, pero hay una diferencia clave. En el patrón State, los estados particulares pueden conocerse entre sí e iniciar transiciones de un estado a otro, mientras que las estrategias casi nunca se conocen.

🚗 Analogía en el mundo real

Los botones e interruptores de tu smartphone se comportan de forma diferente dependiendo del estado actual del dispositivo:

- Cuando el teléfono está desbloqueado, al pulsar botones se ejecutan varias funciones.
- Cuando el teléfono está bloqueado, pulsar un botón desbloquea la pantalla.
- Cuando la batería del teléfono está baja, pulsar un botón muestra la pantalla de carga.

📲 Estructura



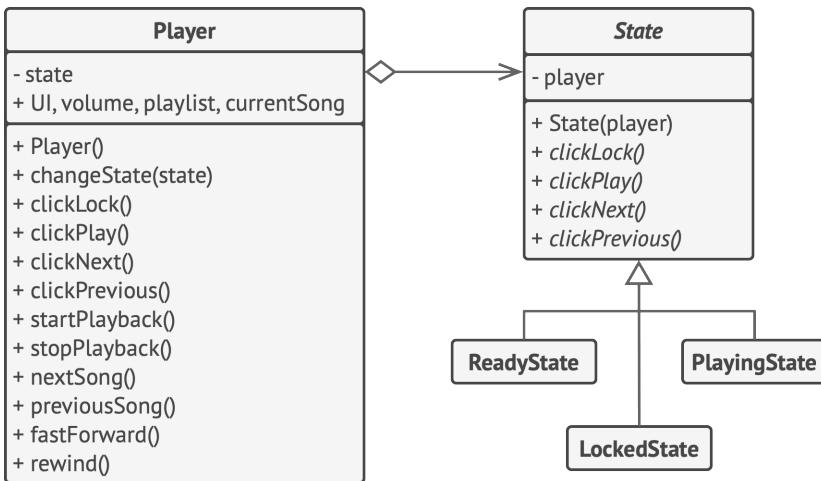
1. La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.
2. La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.
3. Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

4. Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

Pseudocódigo

En este ejemplo, el patrón **State** permite a los mismos controles del reproductor de medios comportarse de forma diferente, dependiendo del estado actual de reproducción.



Ejemplo de cambio del comportamiento de un objeto con objetos de estado.

El objeto principal del reproductor siempre está vinculado a un objeto de estado que realiza la mayor parte del trabajo del reproductor. Algunas acciones sustituyen el objeto de estado actual del reproductor por otro, lo cual cambia la forma en la que el reproductor reacciona a las interacciones del usuario.

```
1 // La clase ReproductordeAudio actúa como un contexto. También
2 // mantiene una referencia a una instancia de una de las clases
3 // estado que representa el estado actual del reproductor de
4 // audio.
5 class AudioPlayer is
6     field state: State
7     field UI, volume, playlist, currentSong
8
9 constructor AudioPlayer() is
10    this.state = new ReadyState(this)
11
12    // El contexto delega la gestión de entradas del usuario
13    // a un objeto de estado. Naturalmente, el resultado
14    // depende del estado que esté activo ahora, ya que cada
15    // estado puede gestionar las entradas de manera
16    // diferente.
17    UI = new UserInterface()
18    UI.lockButton.onClick(this.clickLock)
19    UI.playButton.onClick(this.clickPlay)
20    UI.nextButton.onClick(this.clickNext)
21    UI.prevButton.onClick(this.clickPrevious)
22
23    // Otros objetos deben ser capaces de cambiar el estado
24    // activo del reproductor.
25 method changeState(state: State) is
26    this.state = state
27
28    // Los métodos UI delegan la ejecución al estado activo.
29 method clickLock() is
30    state.clickLock()
31 method clickPlay() is
32    state.clickPlay()
```

```
33 method clickNext() is
34     state.clickNext()
35 method clickPrevious() is
36     state.clickPrevious()
37
38 // Un estado puede invocar algunos métodos del servicio en
39 // el contexto.
40 method startPlayback() is
41     // ...
42 method stopPlayback() is
43     // ...
44 method nextSong() is
45     // ...
46 method previousSong() is
47     // ...
48 method fastForward(time) is
49     // ...
50 method rewind(time) is
51     // ...
52
53
54 // La clase Estado base declara métodos que todos los estados
55 // concretos deben implementar, y también proporciona una
56 // referencia inversa al objeto de contexto asociado con el
57 // estado. Los estados pueden utilizar la referencia inversa
58 // para dirigir el contexto a otro estado.
59 abstract class State is
60     protected field player: AudioPlayer
61
62     // El contexto se pasa a sí mismo a través del constructor
63     // del estado. Esto puede ayudar al estado a extraer
64     // información de contexto útil si la necesita.
```

```
65  constructor State(player) is
66      this.player = player
67
68  abstract method clickLock()
69  abstract method clickPlay()
70  abstract method clickNext()
71  abstract method clickPrevious()
72
73
74 // Los estados concretos implementan varios comportamientos
75 // asociados a un estado del contexto.
76 class LockedState extends State is
77
78 // Cuando desbloqueas a un jugador bloqueado, puede asumir
79 // uno de dos estados.
80 method clickLock() is
81     if (player.playing)
82         player.changeState(new PlayingState(player))
83     else
84         player.changeState(new ReadyState(player))
85
86 method clickPlay() is
87     // Bloqueado, no hace nada.
88
89 method clickNext() is
90     // Bloqueado, no hace nada.
91
92 method clickPrevious() is
93     // Bloqueado, no hace nada.
94
95 // También pueden disparar transiciones de estado en el
96 // contexto.
```

```
97 class ReadyState extends State is
98     method clickLock() is
99         player.changeState(new LockedState(player))
100
101    method clickPlay() is
102        player.startPlayback()
103        player.changeState(new PlayingState(player))
104
105    method clickNext() is
106        player.nextSong()
107
108    method clickPrevious() is
109        player.previousSong()
110
111
112 class PlayingState extends State is
113     method clickLock() is
114         player.changeState(new LockedState(player))
115
116     method clickPlay() is
117         player.stopPlayback()
118         player.changeState(new ReadyState(player))
119
120     method clickNext() is
121         if (event.doubleclick)
122             player.nextSong()
123         else
124             player.fastForward(5)
125
126     method clickPrevious() is
127         if (event.doubleclick)
128             player.previous()
```

```
129     else  
130         player.rewind(5)
```

💡 Aplicabilidad

- ⚡ Utiliza el patrón State cuando tengas un objeto que se comporta de forma diferente dependiendo de su estado actual, el número de estados sea enorme y el código específico del estado cambie con frecuencia.
- ⚡ El patrón sugiere que extraigas todo el código específico del estado y lo metas dentro de un grupo de clases específicas. Como resultado, puedes añadir nuevos estados o cambiar los existentes independientemente entre sí, reduciendo el costo de mantenimiento.
- ⚡ Utiliza el patrón cuando tengas una clase contaminada con enormes condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase.
- ⚡ El patrón State te permite extraer ramas de esos condicionales a métodos de las clases estado correspondientes. Al hacerlo, también puedes limpiar campos temporales y métodos de ayuda implicados en código específico del estado de fuera de tu clase principal.

 **Utiliza el patrón State cuando tengas mucho código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.**

 El patrón State te permite componer jerarquías de clases de estado y reducir la duplicación, extrayendo el código común y metiéndolo en clases abstractas base.

Cómo implementarlo

1. Decide qué clase actuará como contexto. Puede ser una clase existente que ya tiene el código dependiente del estado, o una nueva clase, si el código específico del estado está distribuido a lo largo de varias clases.
2. Declara la interfaz de estado. Aunque puede replicar todos los métodos declarados en el contexto, cántrate en los que pueden contener comportamientos específicos del estado.
3. Para cada estado actual, crea una clase derivada de la interfaz de estado. Despues repasa los métodos del contexto y extrae todo el código relacionado con ese estado para meterlo en tu clase recién creada.

Al mover el código a la clase estado, puede que descubras que depende de miembros privados del contexto. Hay varias soluciones alternativas:

- Haz públicos esos campos o métodos.

- Convierte el comportamiento que estás extrayendo para ponerlo en un método público en el contexto e invócalo desde la clase de estado. Esta forma es desagradable pero rápida y siempre podrás arreglarlo más adelante.
 - Anida las clases de estado en la clase contexto, pero sólo si tu lenguaje de programación soporta clases anidadas.
4. En la clase contexto, añade un campo de referencia del tipo de interfaz de estado y un modificador (*setter*) público que permita sobrescribir el valor de ese campo.
 5. Vuelve a repasar el método del contexto y sustituye los condicionales de estado vacíos por llamadas a métodos correspondientes del objeto de estado.
 6. Para cambiar el estado del contexto, crea una instancia de una de las clases de estado y pásala a la clase contexto. Puedes hacer esto dentro de la propia clase contexto, en distintos estados, o en el cliente. Se haga de una forma u otra, la clase se vuelve dependiente de la clase de estado concreto que instancia.

ΔΔ Pros y contras

- ✓ *Principio de responsabilidad única.* Organiza el código relacionado con estados particulares en clases separadas.
- ✓ *Principio de abierto/cerrado.* Introduce nuevos estados sin cambiar clases de estado existentes o la clase contexto.

- ✓ Simplifica el código del contexto eliminando voluminosos condicionales de máquina de estados.
- ✗ Aplicar el patrón puede resultar excesivo si una máquina de estados sólo tiene unos pocos estados o raramente cambia.

↔ Relaciones con otros patrones

- **Bridge, State, Strategy** (y, hasta cierto punto, **Adapter**) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, todos ellos solucionan problemas diferentes. Un patrón no es simplemente una receta para estructurar tu código de una forma específica. También puede comunicar a otros desarrolladores el problema que resuelve.
- **State** puede considerarse una extensión de **Strategy**. Ambos patrones se basan en la composición: cambian el comportamiento del contexto delegando parte del trabajo a objetos ayudantes. *Strategy* hace que estos objetos sean completamente independientes y no se conozcan entre sí. Sin embargo, *State* no restringe las dependencias entre estados concretos, permitiéndoles alterar el estado del contexto a voluntad.



STRATEGY

También llamado: Estrategia

Strategy es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

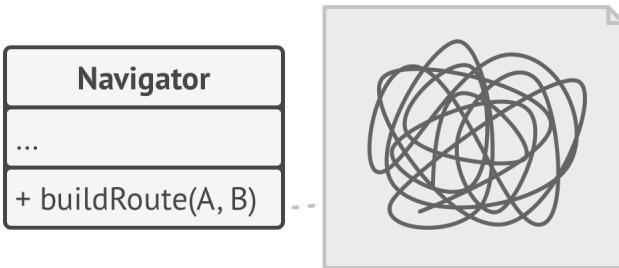
Problema

Un día decidiste crear una aplicación de navegación para viajeros ocasionales. La aplicación giraba alrededor de un bonito mapa que ayudaba a los usuarios a orientarse rápidamente en cualquier ciudad.

Una de las funciones más solicitadas para la aplicación era la planificación automática de rutas. Un usuario debía poder introducir una dirección y ver la ruta más rápida a ese destino mostrado en el mapa.

La primera versión de la aplicación sólo podía generar las rutas sobre carreteras. Las personas que viajaban en coche estaban locas de alegría. Pero, aparentemente, no a todo el mundo le gusta conducir durante sus vacaciones. De modo que, en la siguiente actualización, añadiste una opción para crear rutas a pie. Después, añadiste otra opción para permitir a las personas utilizar el transporte público en sus rutas.

Sin embargo, esto era sólo el principio. Más tarde planeaste añadir la generación de rutas para ciclistas, y más tarde, otra opción para trazar rutas por todas las atracciones turísticas de una ciudad.



El código del navegador se saturó.

Aunque desde una perspectiva comercial la aplicación era un éxito, la parte técnica provocaba muchos dolores de cabeza. Cada vez que añadías un nuevo algoritmo de enrutamiento, la clase principal del navegador doblaba su tamaño. En cierto momento, la bestia se volvió demasiado difícil de mantener.

Cualquier cambio en alguno de los algoritmos, ya fuera un sencillo arreglo de un error o un ligero ajuste de la representación de la calle, afectaba a toda la clase, aumentando las probabilidades de crear un error en un código ya funcional.

Además, el trabajo en equipo se volvió ineficiente. Tus compañeros, contratados tras el exitoso lanzamiento, se quejaban de que dedicaban demasiado tiempo a resolver conflictos de integración. Implementar una nueva función te exige cambiar la misma clase enorme, entrando en conflicto con el código producido por otras personas.

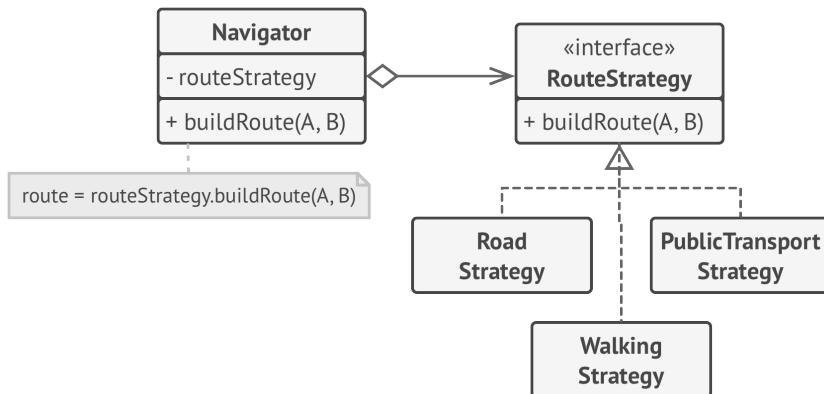
Solución

El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas *estrategias*.

La clase original, llamada *contexto*, debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En lugar de eso, el cliente pasa la estrategia deseada a la clase contexto. De hecho, la clase contexto no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que sólo expone un único método para disparar el algoritmo encapsulado dentro de la estrategia seleccionada.

De esta forma, el contexto se vuelve independiente de las estrategias concretas, así que puedes añadir nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.

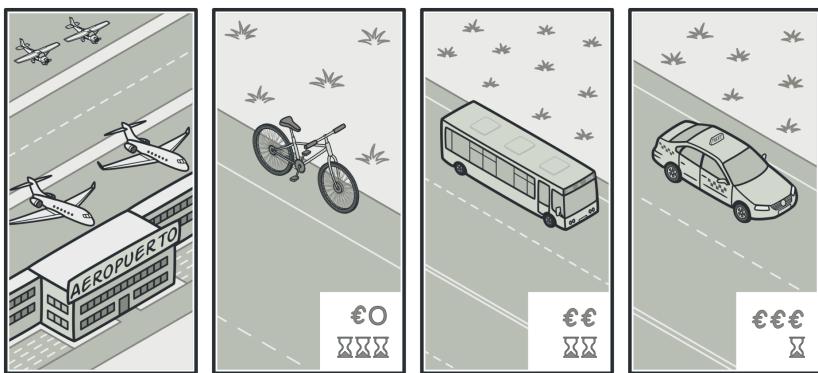


Estrategias de planificación de rutas.

En nuestra aplicación de navegación, cada algoritmo de enrutamiento puede extraerse y ponerse en su propia clase con un único método `crearRuta`. El método acepta un origen y un destino y devuelve una colección de puntos de control de la ruta.

Incluso contando con los mismos argumentos, cada clase de enrutamiento puede crear una ruta diferente. A la clase navegador principal no le importa qué algoritmo se selecciona ya que su labor principal es representar un grupo de puntos de control en el mapa. La clase tiene un método para cambiar la estrategia activa de enrutamiento, de modo que sus clientes, como los botones en la interfaz de usuario, pueden sustituir el comportamiento seleccionado de enrutamiento por otro.

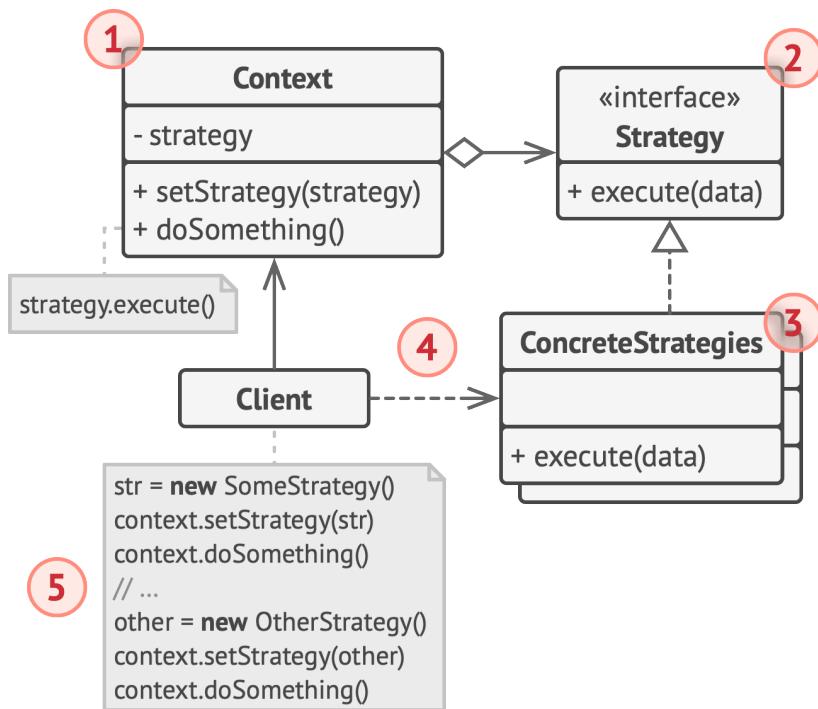
🚗 Analogía en el mundo real



Varias estrategias para llegar al aeropuerto.

Imagina que tienes que llegar al aeropuerto. Puedes tomar el autobús, pedir un taxi o ir en bicicleta. Éstas son tus estrategias de transporte. Puedes elegir una de las estrategias, dependiendo de factores como el presupuesto o los límites de tiempo.

Estructura



1. La clase **Contexto** mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz estrategia.
2. La interfaz **Estrategia** es común a todas las estrategias concretas. Declara un método que la clase contexto utiliza para ejecutar una estrategia.
3. Las **Estrategias Concretas** implementan distintas variaciones de un algoritmo que la clase contexto utiliza.

4. La clase contexto invoca el método de ejecución en el objeto de estrategia vinculado cada vez que necesita ejecutar el algoritmo. La clase contexto no sabe con qué tipo de estrategia funciona o cómo se ejecuta el algoritmo.
5. El **Cliente** crea un objeto de estrategia específico y lo pasa a la clase contexto. La clase contexto expone un modificador *set* que permite a los clientes sustituir la estrategia asociada al contexto durante el tiempo de ejecución.

Pseudocódigo

En este ejemplo, el contexto utiliza varias **estrategias** para ejecutar diversas operaciones aritméticas.

```
1 // La interfaz estrategia declara operaciones comunes a todas
2 // las versiones soportadas de algún algoritmo. El contexto
3 // utiliza esta interfaz para invocar el algoritmo definido por
4 // las estrategias concretas.
5 interface Strategy is
6     method execute(a, b)
7
8     // Las estrategias concretas implementan el algoritmo mientras
9     // siguen la interfaz estrategia base. La interfaz las hace
10    // intercambiables en el contexto.
11 class ConcreteStrategyAdd implements Strategy is
12     method execute(a, b) is
13         return a + b
14
15 class ConcreteStrategySubtract implements Strategy is
```

```
16  method execute(a, b) is
17      return a - b
18
19 class ConcreteStrategyMultiply implements Strategy is
20     method execute(a, b) is
21         return a * b
22
23 // El contexto define la interfaz de interés para los clientes.
24 class Context is
25     // El contexto mantiene una referencia a uno de los objetos
26     // de estrategia. El contexto no conoce la clase concreta de
27     // una estrategia. Debe trabajar con todas las estrategias a
28     // través de la interfaz estrategia.
29     private strategy: Strategy
30
31     // Normalmente, el contexto acepta una estrategia a través
32     // del constructor y también proporciona un setter
33     // (modificador) para poder cambiar la estrategia durante el
34     // tiempo de ejecución.
35     method setStrategy(Strategy strategy) is
36         this.strategy = strategy
37
38     // El contexto delega parte del trabajo al objeto de
39     // estrategia en lugar de implementar varias versiones del
40     // algoritmo por su cuenta.
41     method executeStrategy(int a, int b) is
42         return strategy.execute(a, b)
43
44
45 // El código cliente elige una estrategia concreta y la pasa al
46 // contexto. El cliente debe conocer las diferencias entre
47 // estrategias para elegir la mejor opción.
```

```
48 class ExampleApplication is
49     method main() is
50         Create context object.
51
52         Read first number.
53         Read last number.
54         Read the desired action from user input.
55
56         if (action == addition) then
57             context.setStrategy(new ConcreteStrategyAdd())
58
59         if (action == subtraction) then
60             context.setStrategy(new ConcreteStrategySubtract())
61
62         if (action == multiplication) then
63             context.setStrategy(new ConcreteStrategyMultiply())
64
65         result = context.executeStrategy(First number, Second number)
66
67         Print result.
```

💡 Aplicabilidad

- 💡 Utiliza el patrón Strategy cuando quieras utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- ⚡ El patrón Strategy te permite alterar indirectamente el comportamiento del objeto durante el tiempo de ejecución asociando

ndolo con distintos subobjetos que pueden realizar subtareas específicas de distintas maneras.

- ⚡ **Utiliza el patrón Strategy cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.**
- ⚡ El patrón Strategy te permite extraer el comportamiento variante para ponerlo en una jerarquía de clases separada y combinar las clases originales en una, reduciendo con ello el código duplicado.
- ⚡ **Utiliza el patrón para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.**
- ⚡ El patrón Strategy te permite aislar el código, los datos internos y las dependencias de varios algoritmos, del resto del código. Los diversos clientes obtienen una interfaz simple para ejecutar los algoritmos y cambiarlos durante el tiempo de ejecución.
- ⚡ **Utiliza el patrón cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.**
- ⚡ El patrón Strategy te permite suprimir dicho condicional extrayendo todos los algoritmos para ponerlos en clases separadas,

las cuales implementan la misma interfaz. El objeto original delega la ejecución a uno de esos objetos, en lugar de implementar todas las variantes del algoritmo.

Cómo implementarlo

1. En la clase contexto, identifica un algoritmo que tienda a sufrir cambios frecuentes. También puede ser un enorme condicional que seleccione y ejecute una variante del mismo algoritmo durante el tiempo de ejecución.
2. Declara la interfaz estrategia común a todas las variantes del algoritmo.
3. Uno a uno, extrae todos los algoritmos y ponlos en sus propias clases. Todas deben implementar la misma interfaz estrategia.
4. En la clase contexto, añade un campo para almacenar una referencia a un objeto de estrategia. Proporciona un modificador *set* para sustituir valores de ese campo. La clase contexto debe trabajar con el objeto de estrategia únicamente a través de la interfaz estrategia. La clase contexto puede definir una interfaz que permita a la estrategia acceder a sus datos.
5. Los clientes de la clase contexto deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

⚠️ Pros y contras

- ✓ Puedes intercambiar algoritmos usados dentro de un objeto durante el tiempo de ejecución.
- ✓ Puedes aislar los detalles de implementación de un algoritmo del código que lo utiliza.
- ✓ Puedes sustituir la herencia por composición.
- ✓ *Principio de abierto/cerrado.* Puedes introducir nuevas estrategias sin tener que cambiar el contexto.
- ✗ Si sólo tienes un par de algoritmos que raramente cambian, no hay una razón real para complicar el programa en exceso con nuevas clases e interfaces que vengan con el patrón.
- ✗ Los clientes deben conocer las diferencias entre estrategias para poder seleccionar la adecuada.
- ✗ Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que te permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas. Entonces puedes utilizar estas funciones exactamente como habrías utilizado los objetos de estrategia, pero sin saturar tu código con clases e interfaces adicionales.

➡ Relaciones con otros patrones

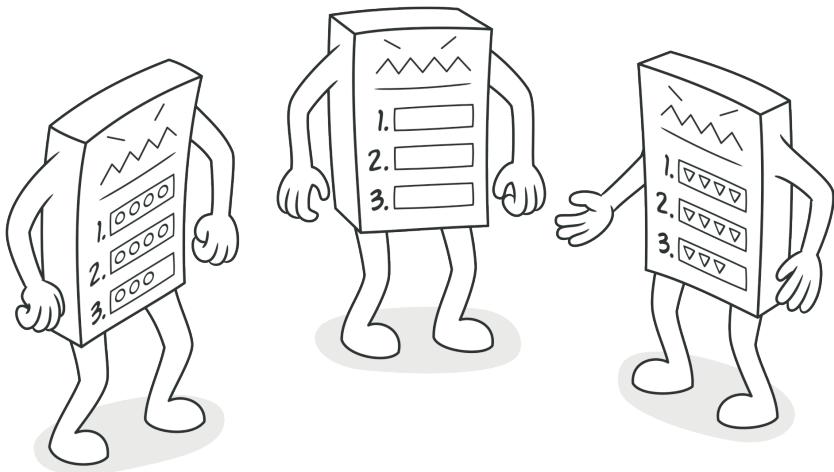
- **Bridge, State, Strategy** (y, hasta cierto punto, **Adapter**) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, todos ellos solucionan problemas

diferentes. Un patrón no es simplemente una receta para estructurar tu código de una forma específica. También puede comunicar a otros desarrolladores el problema que resuelve.

- **Command** y **Strategy** pueden resultar similares porque puedes usar ambos para parametrizar un objeto con cierta acción. No obstante, tienen propósitos muy diferentes.
 - Puedes utilizar *Command* para convertir cualquier operación en un objeto. Los parámetros de la operación se convierten en campos de ese objeto. La conversión te permite aplazar la ejecución de la operación, ponerla en cola, almacenar el historial de comandos, enviar comandos a servicios remotos, etc.
 - Por su parte, *Strategy* normalmente describe distintas formas de hacer lo mismo, permitiéndote intercambiar estos algoritmos dentro de una única clase contexto.
- **Decorator** te permite cambiar la piel de un objeto, mientras que **Strategy** te permite cambiar sus entrañas.
- **Template Method** se basa en la herencia: te permite alterar partes de un algoritmo extendiendo esas partes en subclases. **Strategy** se basa en la composición: puedes alterar partes del comportamiento del objeto suministrándole distintas estrategias que se correspondan con ese comportamiento. *Template Method* trabaja al nivel de la clase, por lo que es estático. *Strat-*

*t*egy trabaja al nivel del objeto, permitiéndote cambiar los comportamientos durante el tiempo de ejecución.

- **State** puede considerarse una extensión de **Strategy**. Ambos patrones se basan en la composición: cambian el comportamiento del contexto delegando parte del trabajo a objetos ayudantes. *Strategy* hace que estos objetos sean completamente independientes y no se conozcan entre sí. Sin embargo, *State* no restringe las dependencias entre estados concretos, permitiéndoles alterar el estado del contexto a voluntad.



TEMPLATE METHOD

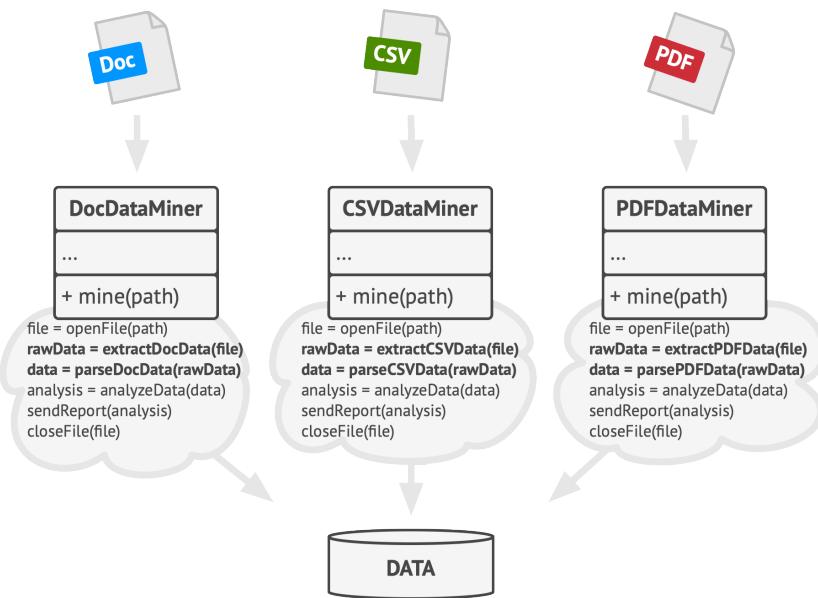
También llamado: Método plantilla

Template Method es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura.

(:() Problema

Imagina que estás creando una aplicación de minería de datos que analiza documentos corporativos. Los usuarios suben a la aplicación documentos en varios formatos (PDF, DOC, CSV) y ésta intenta extraer la información relevante de estos documentos en un formato uniforme.

La primera versión de la aplicación sólo funcionaba con archivos DOC. La siguiente versión podía soportar archivos CSV. Un mes después, le “enseñaste” a extraer datos de archivos PDF.



Las clases de minería de datos contenían mucho código duplicado.

En cierto momento te das cuenta de que las tres clases tienen mucho código similar. Aunque el código para gestionar disti-

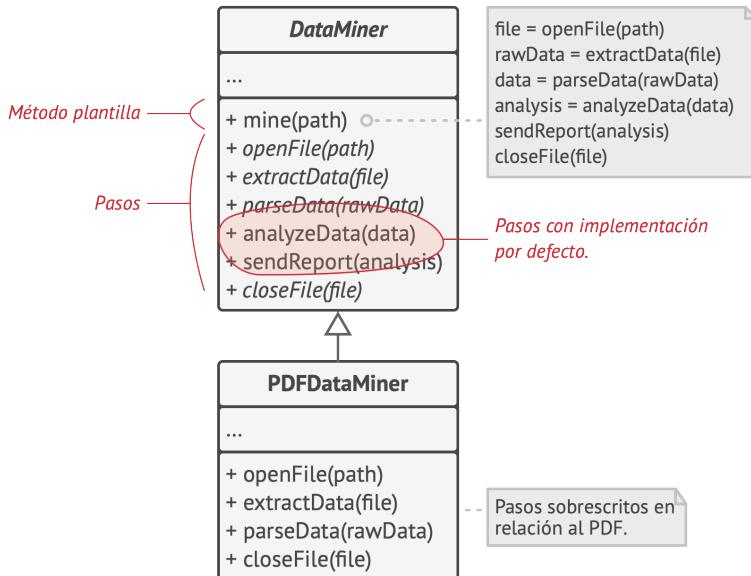
ntos formatos de datos es totalmente diferente en todas las clases, el código para procesar y analizar los datos es casi idéntico. ¿No sería genial deshacerse de la duplicación de código, dejando intacta la estructura del algoritmo?

Hay otro problema relacionado con el código cliente que utiliza esas clases. Tiene muchos condicionales que eligen un curso de acción adecuado dependiendo de la clase del objeto de procesamiento. Si las tres clases de procesamiento tienen una interfaz común o una clase base, puedes eliminar los condicionales en el código cliente y utilizar el polimorfismo al invocar métodos en un objeto de procesamiento.

Solución

El patrón Template Method sugiere que dividas un algoritmo en una serie de pasos, conviertas estos pasos en métodos y coloques una serie de llamadas a esos métodos dentro de un único *método plantilla*. Los pasos pueden ser abstractos, o contar con una implementación por defecto. Para utilizar el algoritmo, el cliente debe aportar su propia subclase, implementar todos los pasos abstractos y sobrescribir algunos de los opcionales si es necesario (pero no el propio método plantilla).

Veamos cómo funciona en nuestra aplicación de minería de datos. Podemos crear una clase base para los tres algoritmos de análisis. Esta clase define un método plantilla consistente en una serie de llamadas a varios pasos de procesamiento de documentos.



El método plantilla divide el algoritmo en pasos, permitiendo a las subclases sobreescribir estos pasos pero no el método en sí.

Al principio, podemos declarar todos los pasos como abstractos, forzando a las subclases a proporcionar sus propias implementaciones para estos métodos. En nuestro caso, las subclases ya cuentan con todas las implementaciones necesarias, por lo que lo único que tendremos que hacer es ajustar las firmas de los métodos para que coincidan con los métodos de la superclase.

Ahora, veamos lo que podemos hacer para deshacernos del código duplicado. Parece que el código para abrir/cerrar archivos y extraer/analizar información es diferente para varios formatos de datos, por lo que no tiene sentido tocar estos métodos. No obstante, la implementación de otros pasos, como analizar los datos sin procesar y generar informes, es muy similar, por

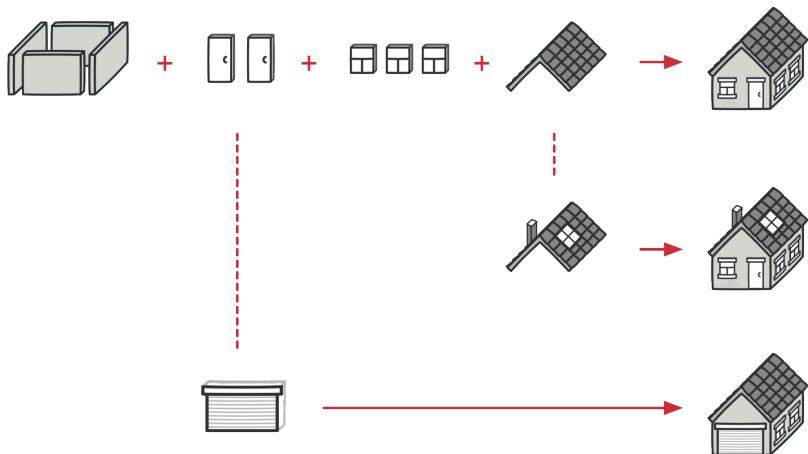
lo que puede meterse en la clase base, donde las subclases pueden compartir ese código.

Como puedes ver, tenemos dos tipos de pasos:

- Los *pasos abstractos* deben ser implementados por todas las subclases
- Los *pasos opcionales* ya tienen cierta implementación por defecto, pero aún así pueden sobrescribirse si es necesario

Hay otro tipo de pasos, llamados ganchos (*hooks*). Un gancho es un paso opcional con un cuerpo vacío. Un método plantilla funcionará aunque el gancho no se sobrescriba. Normalmente, los ganchos se colocan antes y después de pasos cruciales de los algoritmos, suministrando a las subclases puntos adicionales de extensión para un algoritmo.

🚗 Analogía en el mundo real

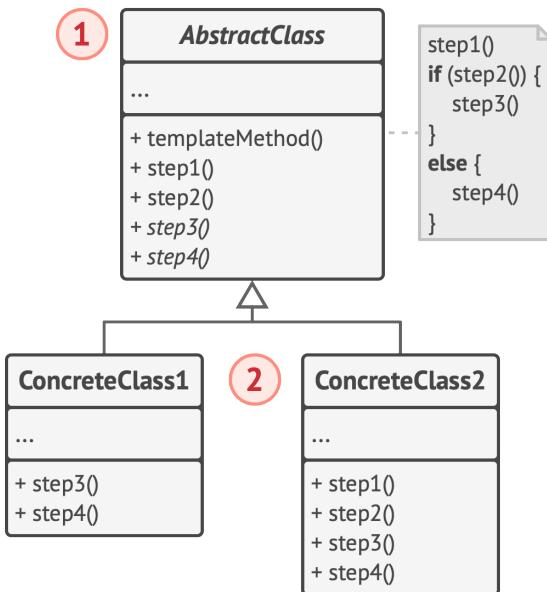


Un plan arquitectónico típico puede alterarse ligeramente para que encaje mejor con las necesidades del cliente.

El enfoque del método plantilla puede emplearse en la construcción de viviendas en masa. El plan arquitectónico para construir una casa estándar puede contener varios puntos de extensión que permitirán a un potencial propietario ajustar algunos detalles de la casa resultante.

Cada paso de la construcción, como colocar los cimientos, el armazón, construir las paredes, instalar las tuberías para el agua y el cableado para la electricidad, etc., puede cambiarse ligeramente para que la casa resultante sea un poco diferente de las demás.

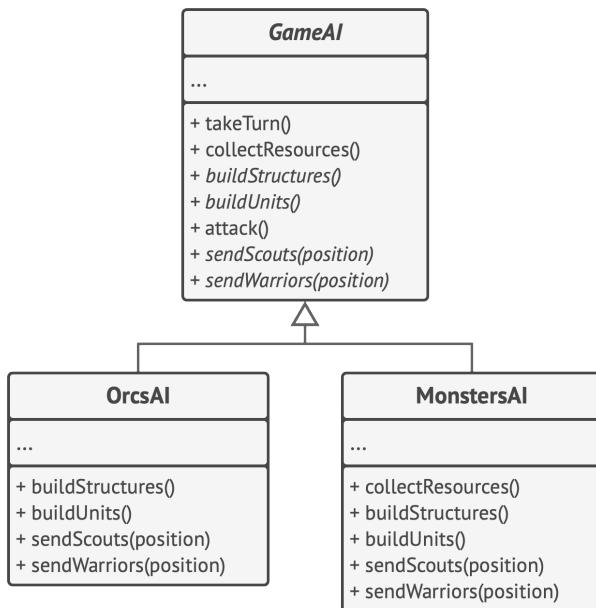
Estructura



1. La **Clase Abstracta** declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse **abstractos** o contar con una implementación por defecto.
2. Las **Clases Concretas** pueden sobrescribir todos los pasos, pero no el propio método plantilla.

Pseudocódigo

En este ejemplo, el patrón **Template Method** proporciona un “esqueleto” para varias ramas de inteligencia artificial (IA) en un sencillo videojuego de estrategia.



Clases IA de un sencillo videojuego.

Todas las razas del juego tienen tipos de unidades y edificios casi iguales. Por lo tanto, puedes reutilizar la misma estructura IA para varias de ellas, a la vez que puedes sobrescribir algunos de los detalles. Con esta solución, puedes sobrescribir la IA de los orcos para que sean más agresivos, hacer que los humanos tengan una actitud más defensiva y hacer que los monstruos no puedan construir nada. Para añadir una nueva raza

al juego habría que crear una nueva subclase IA y sobrescribir los métodos por defecto declarados en la clase IA base.

```
1 // La clase abstracta define un método plantilla que contiene un
2 // esqueleto de algún algoritmo compuesto por llamadas,
3 // normalmente a operaciones primitivas abstractas. Las
4 // subclases concretas implementan estas operaciones, pero dejan
5 // el propio método plantilla intacto.
6 class GameAI is
7     // El método plantilla define el esqueleto de un algoritmo.
8     method turn() is
9         collectResources()
10        buildStructures()
11        buildUnits()
12        attack()
13
14    // Algunos de los pasos se pueden implementar directamente
15    // en una clase base.
16    method collectResources() is
17        foreach (s in this.builtStructures) do
18            s.collect()
19
20    // Y algunos de ellos pueden definirse como abstractos.
21    abstract method buildStructures()
22    abstract method buildUnits()
23
24    // Una clase puede tener varios métodos plantilla.
25    method attack() is
26        enemy = closestEnemy()
27        if (enemy == null)
28            sendScouts(map.center)
```

```
29     else
30         sendWarriors(enemy.position)
31
32     abstract method sendScouts(position)
33     abstract method sendWarriors(position)
34
35 // Las clases concretas tienen que implementar todas las
36 // operaciones abstractas de la clase base, pero no deben
37 // sobrescribir el propio método plantilla.
38 class OrcsAI extends GameAI is
39     method buildStructures() is
40         if (there are some resources) then
41             // Construye granjas, después cuarteles y después
42             // fortaleza.
43
44     method buildUnits() is
45         if (there are plenty of resources) then
46             if (there are no scouts)
47                 // Crea peón y añádelo al grupo de exploradores.
48             else
49                 // Crea soldado, añádelo al grupo de guerreros.
50
51     // ...
52
53     method sendScouts(position) is
54         if (scouts.length > 0) then
55             // Envía exploradores a posición.
56
57     method sendWarriors(position) is
58         if (warriors.length > 5) then
59             // Envía guerreros a posición.
```

```
61 // Las subclases también pueden sobrescribir algunas operaciones
62 // con una implementación por defecto.
63 class MonstersAI extends GameAI is
64     method collectResources() is
65         // Los monstruos no recopilan recursos.
66
67     method buildStructures() is
68         // Los monstruos no construyen estructuras.
69
70     method buildUnits() is
71         // Los monstruos no construyen unidades.
```

💡 Aplicabilidad

- ⚡ Utiliza el patrón Template Method cuando quieras permitir a tus clientes que extiendan únicamente pasos particulares de un algoritmo, pero no todo el algoritmo o su estructura.
- ⚡ El patrón Template Method te permite convertir un algoritmo monolítico en una serie de pasos individuales que se pueden extender fácilmente con subclases, manteniendo intacta la estructura definida en una superclase.
- ⚡ Utiliza el patrón cuando tengas muchas clases que contengan algoritmos casi idénticos, pero con algunas diferencias mínimas. Como resultado, puede que tengas que modificar todas las clases cuando el algoritmo cambie.

⚡ Cuando conviertes un algoritmo así en un método plantilla, también puedes elevar los pasos con implementaciones similares a una superclase, eliminando la duplicación del código. El código que varía entre subclases puede permanecer en las subclases.

Cómo implementarlo

1. Analiza el algoritmo objetivo para ver si puedes dividirlo en pasos. Considera qué pasos son comunes a todas las subclases y cuáles siempre serán únicos.
2. Crea la clase base abstracta y declara el método plantilla y un grupo de métodos abstractos que representen los pasos del algoritmo. Perfila la estructura del algoritmo en el método plantilla ejecutando los pasos correspondientes. Considera declarar el método plantilla como `final` para evitar que las subclases lo sobreescrbían.
3. No hay problema en que todos los pasos acaben siendo abstractos. Sin embargo, a algunos pasos les vendría bien tener una implementación por defecto. Las subclases no tienen que implementar esos métodos.
4. Piensa en añadir ganchos entre los pasos cruciales del algoritmo.

5. Para cada variación del algoritmo, crea una nueva subclase concreta. Ésta *debe* implementar todos los pasos abstractos, pero también *puede* sobrescribir algunos de los opcionales.

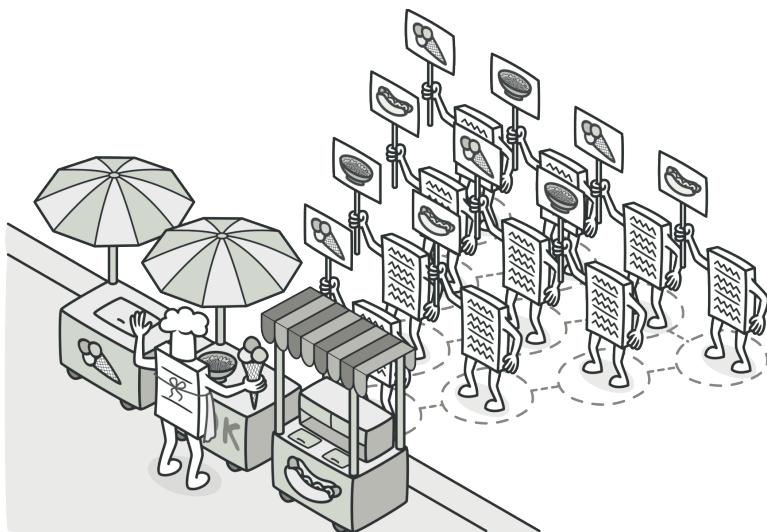
⚠️ Pros y contras

- ✓ Puedes permitir a los clientes que sobrescriban tan solo ciertas partes de un algoritmo grande, para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo.
- ✓ Puedes colocar el código duplicado dentro de una superclase.
- ✗ Algunos clientes pueden verse limitados por el esqueleto proporcionado de un algoritmo.
- ✗ Puede que violes el *principio de sustitución de Liskov* suprimiendo una implementación por defecto de un paso a través de una subclase.
- ✗ Los métodos plantilla tienden a ser más difíciles de mantener cuantos más pasos tengan.

➡️ Relaciones con otros patrones

- **Factory Method** es una especialización del **Template Method**. Al mismo tiempo, un *Factory Method* puede servir como paso de un gran *Template Method*.
- **Template Method** se basa en la herencia: te permite alterar partes de un algoritmo extendiendo esas partes en subclases. **Strategy** se basa en la composición: puedes alterar partes del

comportamiento del objeto suministrándole distintas estrategias que se correspondan con ese comportamiento. *Template Method* trabaja al nivel de la clase, por lo que es estático. *Strategy* trabaja al nivel del objeto, permitiéndote cambiar los comportamientos durante el tiempo de ejecución.



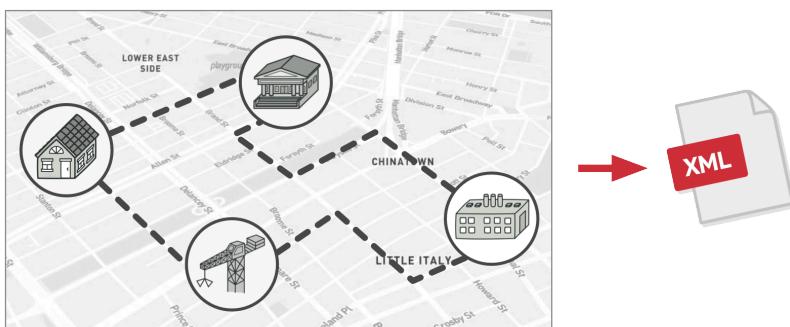
VISITOR

También llamado: Visitante

Visitor es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.

Problema

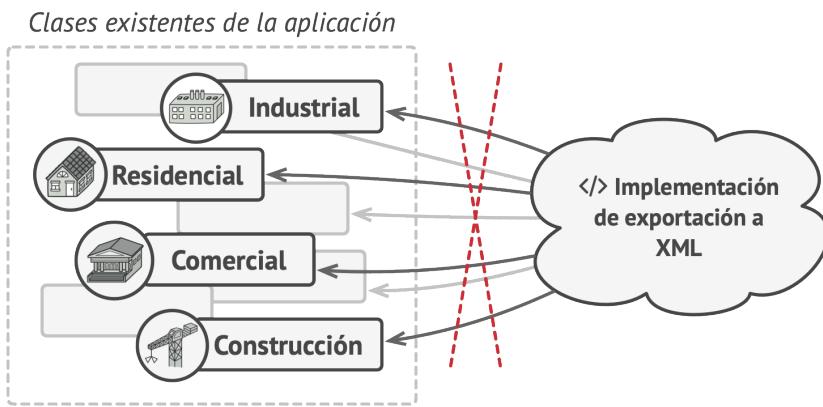
Imagina que tu equipo desarrolla una aplicación que funciona con información geográfica estructurada como un enorme grafo. Cada nodo del grafo puede representar una entidad compleja, como una ciudad, pero también cosas más específicas, como industrias, áreas turísticas, etc. Los nodos están conectados con otros si hay un camino entre los objetos reales que representan. Técnicamente, cada tipo de nodo está representado por su propia clase, mientras que cada nodo específico es un objeto.



Exportando el grafo a XML.

En cierto momento, te surge la tarea de implementar la exportación del grafo a formato XML. Al principio, el trabajo parece bastante sencillo. Planificaste añadir un método de exportación a cada clase de nodo y después aprovechar la recursión para recorrer cada nodo del grafo, ejecutando el método de exportación. La solución era sencilla y elegante: gracias al polimorfismo, no acoplabas el código que invocaba el método de exportación a clases concretas de nodos.

Lamentablemente, el arquitecto del sistema no te permitió alterar las clases de nodo existentes. Dijo que el código ya estaba en producción y no quería arriesgarse a que se descompusiera por culpa de un potencial error en tus cambios.



El método de exportación XML tuvo que añadirse a todas las clases de nodo, lo que supuso el riesgo de descomponer la aplicación si se introducía algún error con el cambio.

Además, cuestionó si tenía sentido tener el código de exportación XML dentro de las clases de nodo. El trabajo principal de estas clases era trabajar con geodatos. El comportamiento de la exportación XML resultaría extraño ahí.

Había otra razón para el rechazo. Era muy probable que, una vez que se implementara esta función, alguien del departamento de marketing te pidiera que incluyeras la capacidad de exportar a otros formatos, o te pidiera alguna otra cosa extraña. Esto te forzaría a cambiar de nuevo esas preciadas y frágiles clases.

Solución

El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada *visitante*, en lugar de intentar integrarlo dentro de clases existentes. El objeto que originalmente tenía que realizar el comportamiento se pasa ahora a uno de los métodos del visitante como argumento, de modo que el método accede a toda la información necesaria contenida dentro del objeto.

Ahora, ¿qué pasa si ese comportamiento puede ejecutarse sobre objetos de clases diferentes? Por ejemplo, en nuestro caso con la exportación XML, la implementación real probablemente sería un poco diferente en varias clases de nodo. Por lo tanto, la clase visitante puede definir un grupo de métodos en lugar de uno solo, y cada uno de ellos podría tomar argumentos de distintos tipos, así:

```
1 class ExportVisitor implements Visitor is
2     method doForCity(City c) { ... }
3     method doForIndustry(Industry f) { ... }
4     method doForSightSeeing(SightSeeing ss) { ... }
5     // ...
```

Pero, ¿cómo llamaríamos exactamente a estos métodos, sobre todo al lidiar con el grafo completo? Estos métodos tienen distintas firmas, por lo que no podemos utilizar el polimorfismo. Para elegir un método visitante adecuado que sea capaz de

procesar un objeto dado, debemos revisar su clase. ¿No suena esto como una pesadilla?

```
1 foreach (Node node in graph)
2     if (node instanceof City)
3         exportVisitor.doForCity((City) node)
4     if (node instanceof Industry)
5         exportVisitor.doForIndustry((Industry) node)
6     // ...
7 }
```

Puede que te preguntes, ¿por qué no utilizar la sobrecarga de métodos? Eso es cuando le das a todos los métodos el mismo nombre, incluso cuando soportan distintos grupos de parámetros. Lamentablemente, incluso asumiendo que nuestro lenguaje de programación la soportara (como Java y C#), no nos ayudaría. Debido a que la clase exacta de un objeto tipo nodo es desconocida de antemano, el mecanismo de sobrecarga no será capaz de determinar el método correcto a ejecutar. Recurriría por defecto al método que toma un objeto de la clase base `Nodo`.

Sin embargo, el patrón Visitor ataja este problema. Utiliza una técnica llamada **Double Dispatch**, que ayuda a ejecutar el método adecuado sobre un objeto sin complicados condicionales. En lugar de permitir al cliente seleccionar una versión adecuada del método a llamar, ¿qué tal si delegamos esta opción a los objetos que pasamos al visitante como argumento? Como estos objetos conocen sus propias clases, podrán elegir un mé-

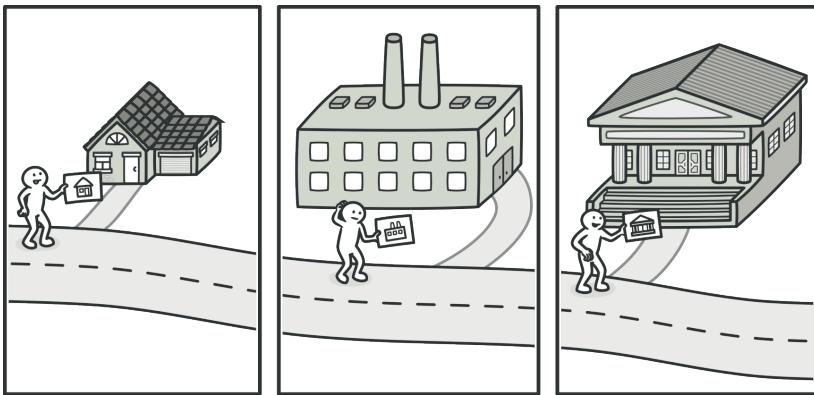
todo adecuado en el visitante más fácilmente. “Aceptan” un visitante y le dicen qué método visitante debe ejecutarse.

```
1 // Código cliente
2 foreach (Node node in graph)
3     node.accept(exportVisitor)
4
5 // Ciudad
6 class City is
7     method accept(Visitor v) is
8         v.doForCity(this)
9         // ...
10
11 // Industria
12 class Industry is
13     method accept(Visitor v) is
14         v.doForIndustry(this)
15         // ...
```

Lo confieso. Hemos tenido que cambiar las clases de nodo, después de todo. Pero al menos el cambio es trivial y nos permite añadir más comportamientos sin alterar el código otra vez.

Ahora, si extraemos una interfaz común para todos los visitantes, todos los nodos existentes pueden funcionar con cualquier visitante que introduzcas en la aplicación. Si te encuentras introduciendo un nuevo comportamiento relacionado con los nodos, todo lo que tienes que hacer es implementar una nueva clase visitante.

🚗 Analogía en el mundo real

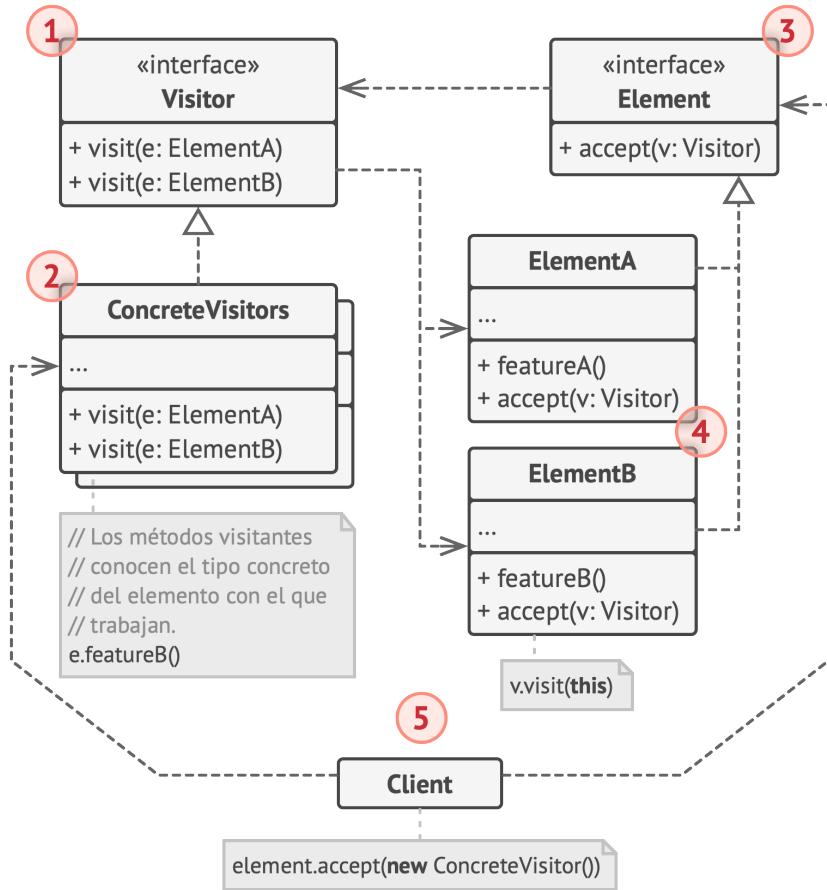


Un buen agente de seguros siempre está listo para ofrecer pólizas diferentes a los distintos tipos de organizaciones.

Imagina un experimentado agente de seguros que está deseoso de conseguir nuevos clientes. Puede visitar todos los edificios de un barrio, intentando vender seguros a todo aquel que se va encontrando. Dependiendo del tipo de organización que ocupe el edificio, puede ofrecer pólizas de seguro especializadas:

- Si es un edificio residencial, vende seguros médicos.
- Si es un banco, vende seguros contra robos.
- Si es una cafetería, vende seguros contra incendios e inundaciones.

Estructura

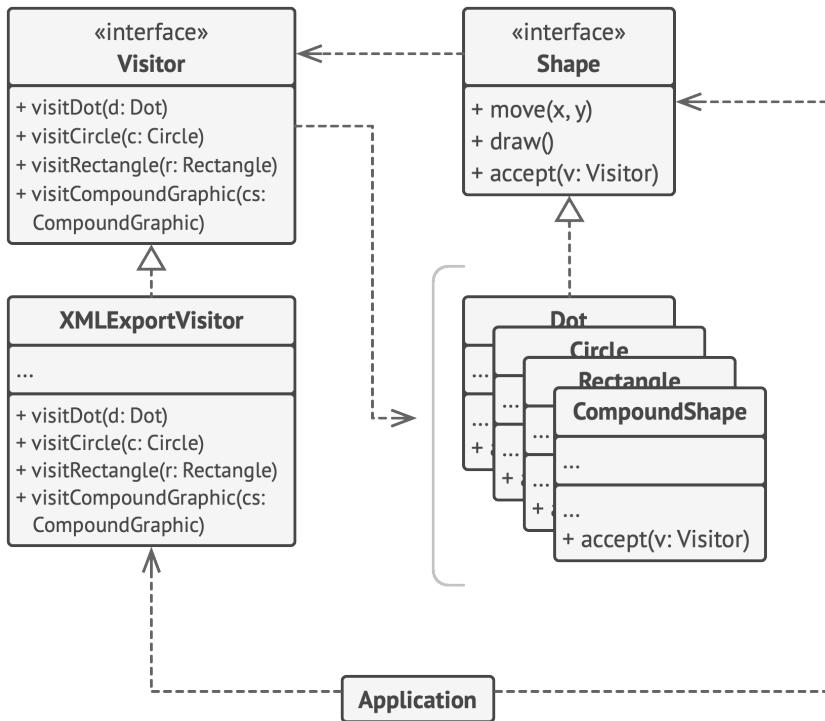


1. La interfaz **Visitante** declara un grupo de métodos visitantes que pueden tomar elementos concretos de una estructura de objetos como argumentos. Estos métodos pueden tener los mismos nombres si el programa está escrito en un lenguaje que soporte la sobrecarga, pero los tipos de sus parámetros deben ser diferentes.

2. Cada **Visitante Concreto** implementa varias versiones de los mismos comportamientos, personalizadas para las distintas clases de elemento concreto.
3. La interfaz **Elemento** declara un método para “aceptar” visitantes. Este método deberá contar con un parámetro declarado con el tipo de la interfaz visitante.
4. Cada **Elemento Concreto** debe implementar el método de aceptación. El propósito de este método es redirigir la llamada al método adecuado del visitante correspondiente a la clase de elemento actual. Piensa que, aunque una clase base de elemento implemente este método, todas las subclases deben sobre escribir este método en sus propias clases e invocar el método adecuado en el objeto visitante.
5. El **Cliente** representa normalmente una colección o algún otro objeto complejo (por ejemplo, un árbol **Composite**). A menudo, los clientes no son conscientes de todas las clases de elemento concreto porque trabajan con objetos de esa colección a través de una interfaz abstracta.

Pseudocódigo

En este ejemplo, el patrón **Visitor** añade soporte de exportación XML a la jerarquía de clases de formas geométricas.



Exportar varios tipos de objetos a formato XML a través de un objeto visitante.

```

1 // La interfaz elemento declara un método `accept` (aceptar) que
2 // toma la interfaz visitante base como argumento.
3 interface Shape is
4     method move(x, y)
5     method draw()
6     method accept(v: Visitor)
7
8 // Cada clase de elemento concreto debe implementar el método
9 // `accept` de tal manera que invoque el método del visitante
10 // que corresponde a la clase del elemento.
11 class Dot implements Shape is
  
```

```
12 // ...
13
14 // Observa que invocamos `visitDot`, que coincide con el
15 // nombre de la clase actual. De esta forma, hacemos saber
16 // al visitante la clase del elemento con el que trabaja.
17 method accept(v: Visitor) is
18     v.visitDot(this)
19
20 class Circle implements Shape is
21 // ...
22 method accept(v: Visitor) is
23     v.visitCircle(this)
24
25 class Rectangle implements Shape is
26 // ...
27 method accept(v: Visitor) is
28     v.visitRectangle(this)
29
30 class CompoundShape implements Shape is
31 // ...
32 method accept(v: Visitor) is
33     v.visitCompoundShape(this)
34
35
36 // La interfaz Visitor declara un grupo de métodos de visita que
37 // se corresponden con clases de elemento. La firma de un método
38 // de visita permite al visitante identificar la clase exacta
39 // del elemento con el que trata.
40 interface Visitor is
41     method visitDot(d: Dot)
42     method visitCircle(c: Circle)
43     method visitRectangle(r: Rectangle)
```

```
44     method visitCompoundShape(cs: CompoundShape)
45
46     // Los visitantes concretos implementan varias versiones del
47     // mismo algoritmo, que puede funcionar con todas las clases de
48     // elementos concretos.
49     //
50     // Puedes disfrutar de la mayor ventaja del patrón Visitor si lo
51     // utilizas con una estructura compleja de objetos, como un
52     // árbol Composite. En este caso, puede ser de ayuda almacenar
53     // algún estado intermedio del algoritmo mientras ejecutas los
54     // métodos del visitante sobre varios objetos de la estructura.
55 class XMLExportVisitor implements Visitor is
56     method visitDot(d: Dot) is
57         // Exporta la ID del punto (dot) y centra las
58         // coordenadas.
59
60     method visitCircle(c: Circle) is
61         // Exporta la ID del círculo y centra las coordenadas y
62         // el radio.
63
64     method visitRectangle(r: Rectangle) is
65         // Exporta la ID del rectángulo, las coordenadas de
66         // arriba a la izquierda, la anchura y la altura.
67
68     method visitCompoundShape(cs: CompoundShape) is
69         // Exporta la ID de la forma, así como la lista de las
70         // ID de sus hijos.
71
72
73     // El código cliente puede ejecutar operaciones del visitante
74     // sobre cualquier grupo de elementos sin conocer sus clases
75     // concretas. La operación `accept` dirige una llamada a la
```

```
76 // operación adecuada del objeto visitante.  
77 class Application is  
78     field allShapes: array of Shapes  
79  
80     method export() is  
81         exportVisitor = new XMLExportVisitor()  
82  
83     foreach (shape in allShapes) do  
84         shape.accept(exportVisitor)
```

Si te preguntas por qué necesitamos el método `aceptar` en este ejemplo, mi artículo [Visitor y Double Dispatch](#) aborda esta cuestión en detalle.

Aplicabilidad

-  Utiliza el patrón Visitor cuando necesites realizar una operación sobre todos los elementos de una compleja estructura de objetos (por ejemplo, un árbol de objetos).
-  El patrón Visitor te permite ejecutar una operación sobre un grupo de objetos con diferentes clases, haciendo que un objeto visitante implemente distintas variantes de la misma operación que correspondan a todas las clases objetivo.
-  Utiliza el patrón Visitor para limpiar la lógica de negocio de comportamientos auxiliares.

- ⚡ El patrón te permite hacer que las clases primarias de tu aplicación estén más centradas en sus trabajos principales extrayendo el resto de los comportamientos y poniéndolos dentro de un grupo de clases visitantes.

- 💡 Utiliza el patrón cuando un comportamiento solo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.

- ⚡ Puedes extraer este comportamiento y ponerlo en una clase visitante separada e implementar únicamente aquellos métodos visitantes que acepten objetos de clases relevantes, dejando el resto vacíos.

Cómo implementarlo

1. Declara la interfaz visitante con un grupo de métodos “visitantes”, uno por cada clase de elemento concreto existente en el programa.

2. Declara la interfaz de elemento. Si estás trabajando con una jerarquía de clases de elemento existente, añade el método abstracto de “aceptación” a la clase base de la jerarquía. Este método debe aceptar un objeto visitante como argumento.

3. Implementa los métodos de aceptación en todas las clases de elemento concreto. Estos métodos simplemente deben redirigir la llamada a un método visitante en el objeto visitante entrante que coincida con la clase del elemento actual.

4. Las clases de elemento sólo deben funcionar con visitantes a través de la interfaz visitante. Los visitantes, sin embargo, deben conocer todas las clases de elemento concreto, referenciadas como tipos de parámetro de los métodos de visita.
5. Por cada comportamiento que no pueda implementarse dentro de la jerarquía de elementos, crea una nueva clase concreta visitante e implementa todos los métodos visitantes.

Puede que te encuentres una situación en la que el visitante necesite acceso a algunos miembros privados de la clase elemento. En este caso, puedes hacer estos campos o métodos públicos, violando la encapsulación del elemento, o anidar la clase visitante en la clase elemento. Esto último sólo es posible si tienes la suerte de trabajar con un lenguaje de programación que soporte clases anidadas.

6. El cliente debe crear objetos visitantes y pasarlo dentro de elementos a través de métodos de “aceptación”.

⚠️ Pros y contras

- ✓ *Principio de abierto/cerrado.* Puedes introducir un nuevo comportamiento que puede funcionar con objetos de clases diferentes sin cambiar esas clases.
- ✓ *Principio de responsabilidad única.* Puedes tomar varias versiones del mismo comportamiento y ponerlas en la misma clase.
- ✓ Un objeto visitante puede acumular cierta información útil mientras trabaja con varios objetos. Esto puede resultar útil

cuando quieras atravesar una compleja estructura de objetos, como un árbol de objetos, y aplicar el visitante a cada objeto de esa estructura.

- ✗ Debes actualizar todos los visitantes cada vez que una clase se añada o elimine de la jerarquía de elementos.
- ✗ Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos con los que se supone que deben trabajar.

↔ Relaciones con otros patrones

- Puedes tratar a **Visitor** como una versión potente del patrón **Command**. Sus objetos pueden ejecutar operaciones sobre varios objetos de distintas clases.
- Puedes utilizar el patrón **Visitor** para ejecutar una operación sobre un árbol **Composite** entero.
- Puedes utilizar **Visitor** junto con **Iterator** para recorrer una estructura de datos compleja y ejecutar alguna operación sobre sus elementos, incluso aunque todos tengan clases distintas.