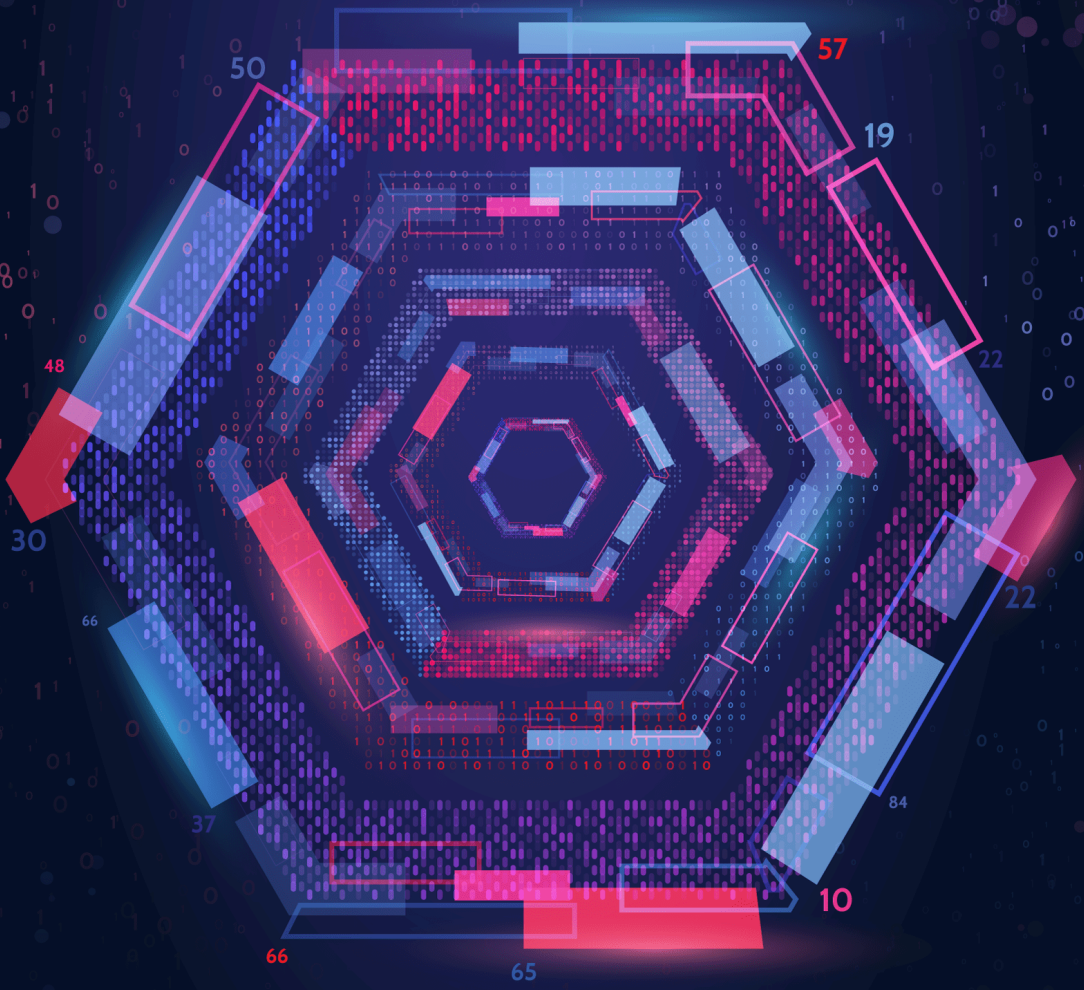
Sumérgete en los PATRONES DE DISEÑO



Alexander Shvets

Sumérgete en los

PATRONES DE DISEÑO

v2021-1.7



El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

Unas palabras sobre derechos de autor



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

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

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

© Alexander Shvets, Refactoring.Guru, 2019

support@refactoring.guru

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

www.detodoprogramacion.org

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

www.detodoprogramacion.org

Índice de contenido

Índice de contenido **4**

Cómo leer este libro **6**

FUNDAMENTOS DE LA POO **7**

Conceptos básicos de POO 8
Los pilares de la POO 13
Relaciones entre objetos 21

INTRODUCCIÓN A LOS PATRONES DE DISEÑO **27**

¿Qué es un patrón de diseño? 28
¿Por qué debería aprender sobre patrones? 33

PRINCIPIOS DE DISEÑO DE SOFTWARE **34**

Características del buen diseño 35

Principios del diseño **39**

§ Encapsula lo que varía 40
§ Programa a una interfaz, no a una implementación 45
§ Favorece la composición sobre la herencia 50

Principios SOLID **54**

§ S: Principio de responsabilidad única 55
§ O: Principio de abierto/cerrado 57
§ L: Principio de sustitución de Liskov 61
§ I: Principio de segregación de la interfaz 68
§ D: Principio de inversión de la dependencia 71

EL CATÁLOGO DE PATRONES DE DISEÑO 75

Patrones creacionales 76

§ Factory Method / *Método fábrica* 78
§ Abstract Factory / *Fábrica abstracta* 95
§ Builder / *Constructor* 111
§ Prototype / *Prototipo* 132
§ Singleton / *Instancia única* 148

Patrones estructurales 158

§ Adapter / *Adaptador* 161
§ Bridge / *Puente* 175
§ Composite / *Objeto compuesto* 192
§ Decorator / *Decorador* 206
§ Facade / *Fachada* 226
§ Flyweight / *Peso mosca* 237
§ Proxy 252

Patrones de comportamiento 266

§ Chain of Responsibility / *Cadena de responsabilidad* 270
§ Command / *Comando* 289
§ Iterator / *Iterador* 310
§ Mediator / *Mediador* 326
§ Memento / *Recuerdo* 340
§ Observer / *Observador* 358
§ State / *Estado* 374
§ Strategy / *Estrategia* 391
§ Template Method / *Método plantilla* 406
§ Visitor / *Visitante* 420

Conclusión 436

Cómo leer este libro

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

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

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

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

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

www.detodoprogramacion.org

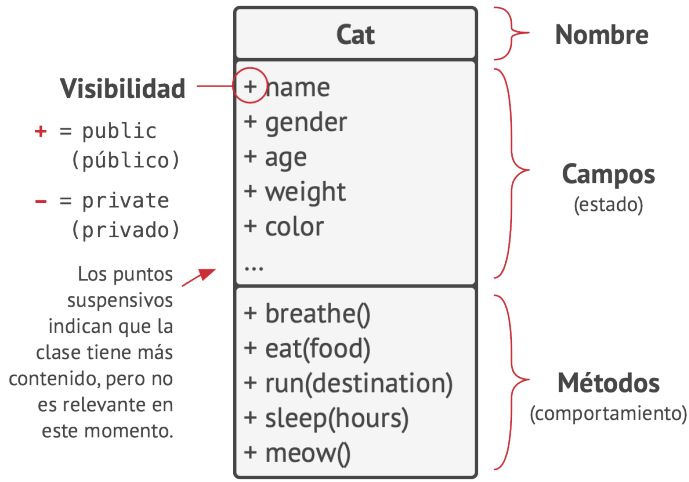
FUNDAMENTOS DE LA POO

Conceptos básicos de POO

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

Objetos, clases

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



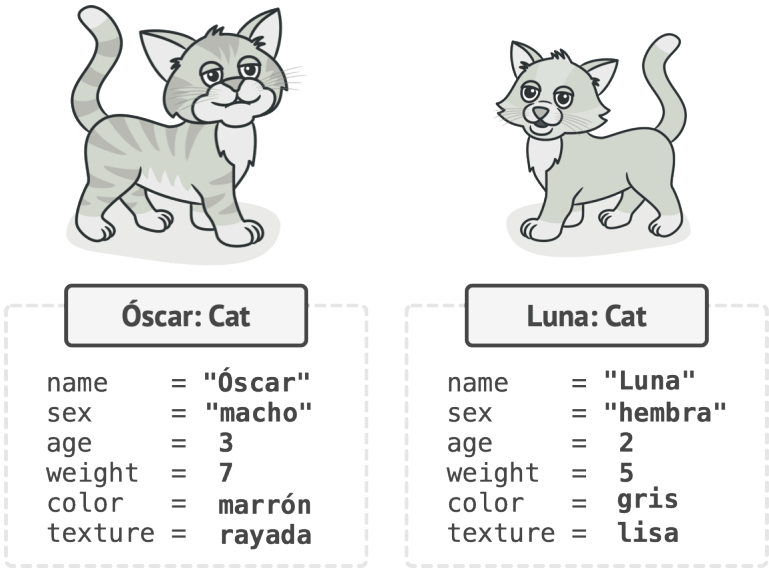
Esto es un diagrama de clases en UML. Encontrarás muchos diagramas como éste en el libro. Los nombres de las cosas en los diagramas están en inglés, como lo estarían en un código real. Sin embargo, los comentarios y las notas pueden estar en español.

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

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

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

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



Los objetos son instancias de clases.

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

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

Jerarquías de clase

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

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

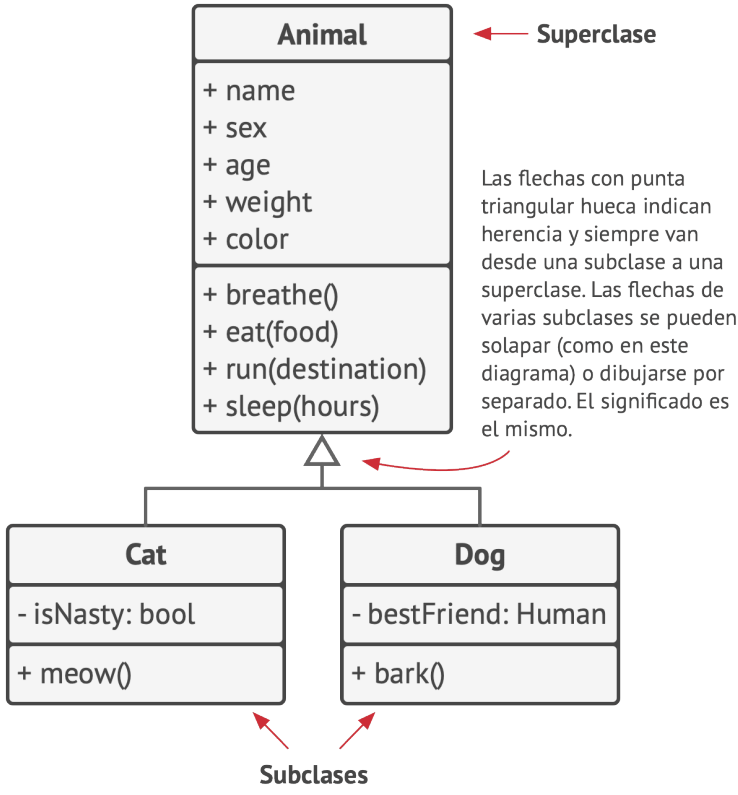
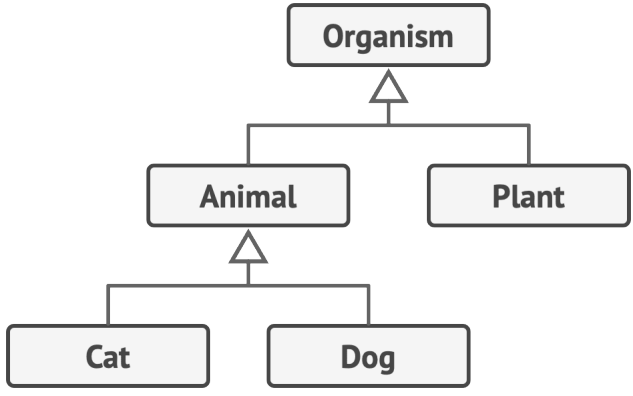


Diagrama UML de una jerarquía de clases. Todas las clases de este diagrama son parte de la jerarquía de clases `Animal`.

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

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

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



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

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

Los pilares de la POO

La programación orientada a objetos se basa en cuatro pilares, conceptos que la diferencian de otros paradigmas de programación.

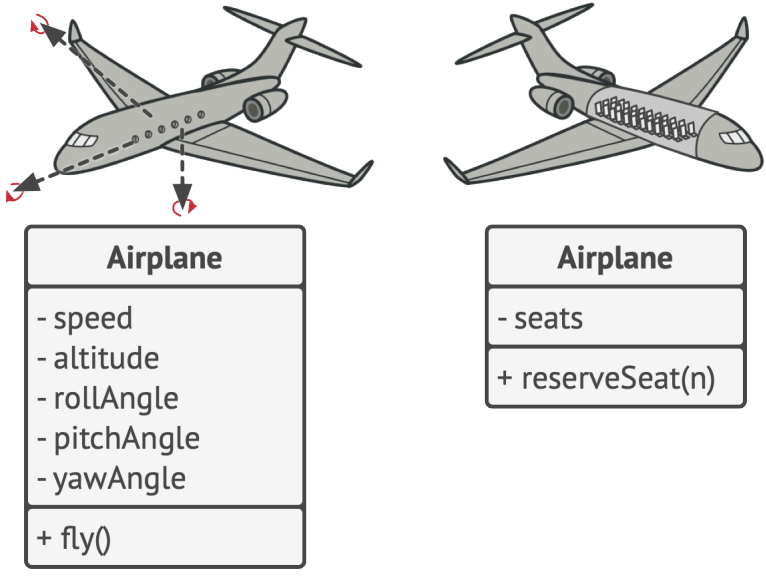


Abstracción

La mayoría de las veces, cuando creas un programa con POO, das forma a los objetos del programa con base a objetos del mundo real. Sin embargo, los objetos del programa no representan a los originales con una precisión del 100 % (y rara vez es necesario que lo hagan). En su lugar, tus objetos tan solo *copian* atributos y comportamientos de objetos reales en un contexto específico, ignorando el resto.

Por ejemplo, una clase `Avión` probablemente podría existir en un simulador de vuelo y en una aplicación de reserva de vuelos. Pero, en el primer caso, contendría información rela-

cionada con el propio vuelo, mientras que en la segunda clase sólo habría que preocuparse del mapa de asientos y de los asientos que estén disponibles.



Distintos modelos del mismo objeto del mundo real.

La *Abstracción* es el modelo de un objeto o fenómeno del mundo real, limitado a un contexto específico, que representa todos los datos relevantes a este contexto con gran precisión, omitiendo el resto.

Encapsulación

Para arrancar el motor de un auto, tan solo debes girar una llave o pulsar un botón. No necesitas conectar cables bajo el capó, rotar el cigüeñal y los cilindros, e iniciar el ciclo de po-

tencia del motor. Estos detalles se esconden bajo el capó del auto. Sólo tienes una interfaz simple: un interruptor de encendido, un volante y unos pedales. Esto ilustra el modo en que cada objeto cuenta con una **interfaz**: una parte pública de un objeto, abierta a interacciones con otros objetos.

La *encapsulación* es la capacidad que tiene un objeto de esconder partes de su estado y comportamiento de otros objetos, exponiendo únicamente una interfaz limitada al resto del programa.

Encapsular algo significa hacerlo `privado` y, por ello, accesible únicamente desde dentro de los métodos de su propia clase. Existe un modelo un poco menos restrictivo llamado `protegido` que hace que un miembro de una clase también esté disponible para las subclases.

Las interfaces y las clases y métodos abstractos de la mayoría de los lenguajes de programación se basan en conceptos de abstracción y encapsulación. En los lenguajes modernos de programación orientada a objetos, el mecanismo de la interfaz (declarado normalmente con la palabra clave `interface` o `protocol`) te permite definir contratos de interacción entre objetos. Ésta es una de las razones por las que las interfaces sólo se interesan por los comportamientos de los objetos, y también el motivo por el que no puedes declarar un campo en una interfaz.

www.detodoprogramacion.org

> El hecho de que la palabra *interfaz* se refiera a la parte pública de un objeto, mientras que también existe el tipo `interface` en la mayoría de los lenguajes de programación, resulta muy confuso. Estoy contigo.

Imagina que tienes una interfaz `TransporteAéreo` con un método `vuelo(origen, destino, pasajeros)`.

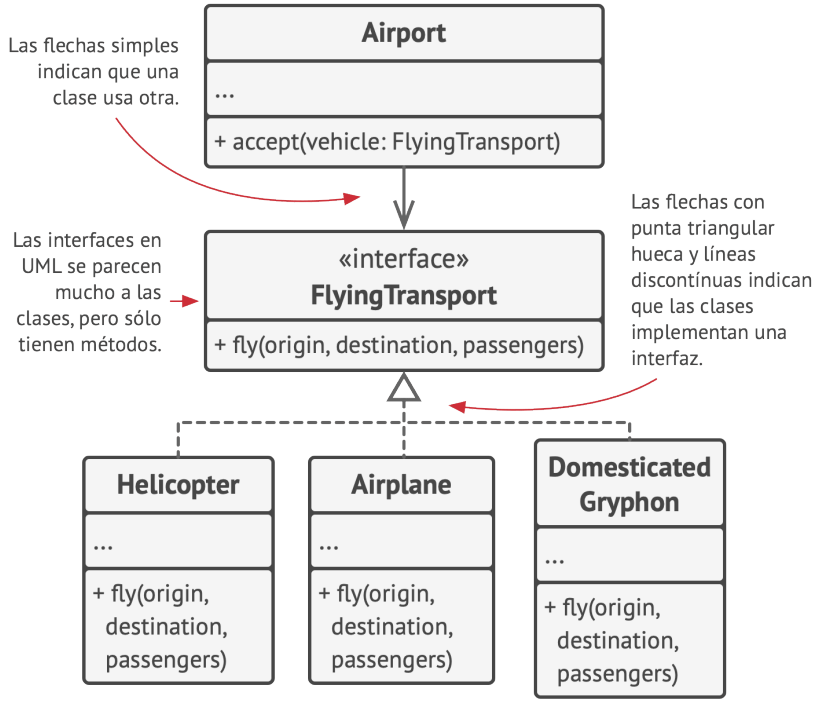


Diagrama UML de varias clases implementando una interfaz.

Al diseñar un simulador de transporte aéreo, puedes restringir la clase `Aeropuerto` para que sólo funcione con objetos que

implementan la interfaz `TransporteAéreo`. Después de esto, tendrás la certeza de que cualquier objeto pasado a un objeto del aeropuerto, ya sea un `Avión`, un `Helicóptero` o hasta un maldito `GrifoDomesticado` si quieres, podrá aterrizar o despegar de este tipo de aeropuerto.

Puedes cambiar la implementación del método `volar` en esas clases del modo que quieras. Siempre y cuando la firma del método sea la misma que se declaró en la interfaz, todas las instancias de la clase `Aeropuerto` pueden funcionar bien con tus objetos voladores.

Herencia

La *herencia* es la capacidad de crear nuevas clases sobre otras existentes. La principal ventaja de la herencia es la reutilización de código. Si quieres crear una clase ligeramente diferente a una ya existente, no hay necesidad de duplicar el código. En su lugar, extiendes la clase existente y colocas la funcionalidad adicional dentro de una subclase resultante que hereda los campos y métodos de la superclase.

La consecuencia del uso de la herencia es que las subclases tienen la misma interfaz que su clase padre. No puedes esconder un método en una subclase si se declaró en la superclase. También debes implementar todos los métodos abstractos, aunque no tengan sentido en tu subclase.

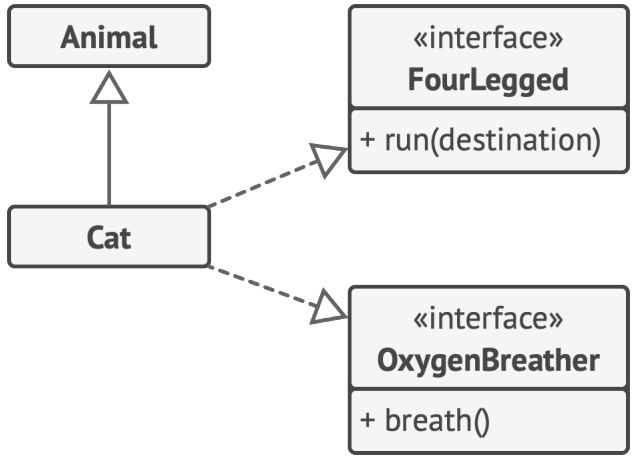


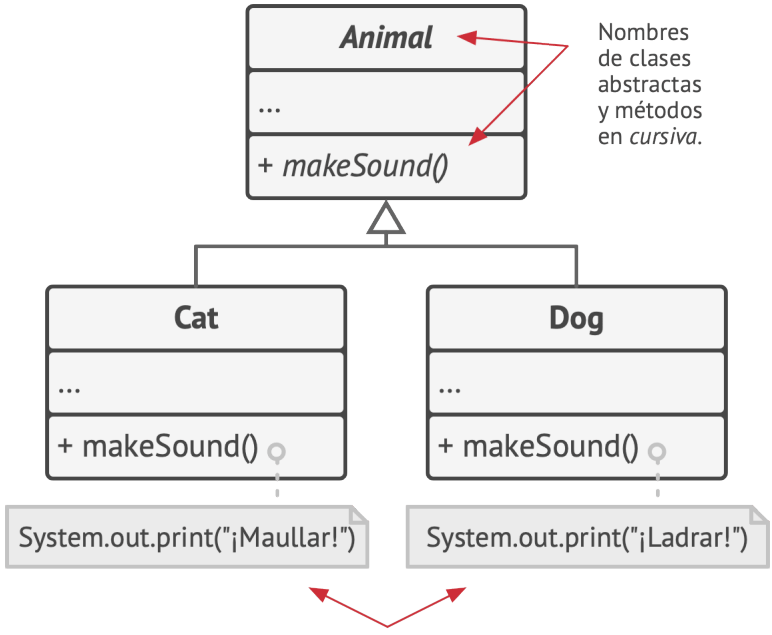
Diagrama UML de extensión de una única clase en comparación con la implementación de múltiples interfaces al mismo tiempo.

En la mayoría de los lenguajes de programación una subclase puede extender una única superclase. Por otro lado, cualquier clase puede implementar varias interfaces al mismo tiempo. Pero, como expliqué antes, si una superclase implementa una interfaz, todas sus subclases deben implementarla también.

Polimorfismo

Veamos algunos ejemplos con animales. La mayoría de los `Animal` puede emitir sonidos. Podemos anticipar que todas las subclases necesitarán sobrescribir el método base `emitirSonido` para que cada subclase pueda emitir el sonido correcto; por lo tanto, podemos declararlo *abstracto* directamente. Esto nos permite omitir cualquier implementación por defecto del método en la superclase, pero fuerza a todas las subclases a establecer las suyas propias.

www.detodoprogramacion.org



Imagina que ponemos varios gatos y perros dentro de una gran bolsa. Después, con los ojos tapados, vamos sacando los animales de la bolsa, de uno en uno. Al sacar un animal, no sabemos con seguridad lo que es. Pero si lo abrazamos lo suficiente, el animal emitirá un sonido específico de alegría, dependiendo de su clase concreta.

```
bag = [new Cat(), new Dog()];

foreach (Animal a : bag)
  a.makeSound()

// ¡Miau!
// ¡Guau!
```

El programa no conoce el tipo concreto del objeto contenido dentro de la variable `a`, pero, gracias al mecanismo especial llamado *polimorfismo*, el programa puede rastrear la subclase del objeto cuyo método está siendo ejecutado, y ejecutar el comportamiento adecuado.

El *polimorfismo* es la capacidad que tiene un programa de detectar la verdadera clase de un objeto e invocar su implementación, incluso aunque su tipo real sea desconocido en el contexto actual.

También puedes pensar en el polimorfismo como la capacidad de un objeto para "fingir" ser otra cosa, normalmente una clase que extiende o una interfaz que implementa. En nuestro ejemplo, los perros y gatos de la bolsa fingen ser animales genéricos.

Relaciones entre objetos

Además de la *herencia* y la *implementación*, que ya vimos, hay otros tipos de relaciones entre objetos de las que aún no hemos hablado.

Dependencia



Dependencia en UML. El profesor depende de los materiales del curso.

La *dependencia* es el tipo de relación más básica y débil entre clases. Existe una dependencia entre dos clases cuando ciertos cambios en la definición de una clase puede provocar modificaciones en otra. La dependencia ocurre normalmente cuando utilizas nombres de clases concretas en tu código. Por ejemplo, al especificar tipos en las firmas de un método, al instanciar objetos mediante llamadas al constructor, etc. Puedes hacer más débil una dependencia haciendo que tu código dependa de interfaces o clases abstractas en lugar de clases concretas.

Normalmente, un diagrama UML no muestra todas las dependencias; hay demasiadas en cualquier código real. En lugar de contaminar el diagrama con dependencias, debes ser muy selectivo y mostrar únicamente aquellas que son importantes para lo que sea que estás comunicando.

www.detodoprogramacion.org

Asociación



Asociación en UML. El profesor se comunica con los estudiantes.

La *asociación* es una relación en la que un objeto utiliza o interactúa con otro. En diagramas UML, la relación de asociación se muestra mediante una flecha simple desde un objeto y apuntando hacia el objeto que utiliza. Por cierto, es totalmente normal tener una asociación bidireccional. En este caso, la flecha tiene una punta en cada extremo. La asociación puede verse como un tipo especializado de dependencia, en la que un objeto siempre tiene acceso a los objetos con los que interactúa, mientras que la dependencia simple no establece un vínculo permanente entre los objetos.

En general, se utiliza la asociación para representar algo como un campo en una clase. El vínculo siempre está ahí, en cuanto a que siempre puedes pedir una orden para su cliente. Pero no siempre tiene que ser un campo. Si estás modelando tus clase desde una perspectiva de interfaz, puede simplemente indicar la presencia de un método que devolverá el cliente de la orden.

Para asentar tu comprensión de la diferencia entre asociación y dependencia, veamos un ejemplo combinado. Imagina que tenemos una clase `Profesor`:

www.detodoprogramacion.org

```
class Professor is
  field Student student
  // ...
  method teach(Course c) is
    // ...
    this.student.remember(c.getKnowledge())
```

Observa el método `enseñar`. Toma un argumento de la clase `Curso`, que después se utiliza en el cuerpo del método. Si alguien cambia el método `obtenerConocimientos` de la clase `Curso` (es decir, si cambia su nombre o añade algún parámetro requerido, etc.) nuestro código se descompondrá. Esto se llama dependencia.

Ahora observa el campo `Estudiante` y cómo se utiliza en el método `enseñar`. Podemos afirmar con seguridad que la clase `Estudiante` también es una dependencia para el `Profesor`: si el método `recordar` cambia, el código de `Profesor` se romperá. No obstante, como el campo `Estudiante` siempre está accesible para cualquier método de `Profesor`, la clase `Estudiante` no es sólo una dependencia, sino también una asociación.

Agregación



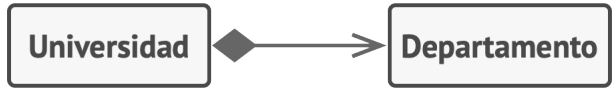
Agregación en UML. Los departamentos contienen profesores.

La *agregación* es un tipo especializado de asociación que representa relaciones "uno a muchos", "muchos a muchos" o "todo a parte" entre múltiples objetos, mientras que una asociación simple describe relaciones entre un par de objetos.

Normalmente, con la agregación, un objeto "tiene" un grupo de otros objetos y sirve como contenedor o colección. El componente puede existir sin el contenedor y puede vincularse a varios contenedores al mismo tiempo. En UML, la relación de agregación se muestra por una línea con un diamante vacío en el lado del contenedor y una flecha en el extremo apuntando hacia el componente.

Mientras hablamos de relaciones entre objetos, ten en cuenta que UML representa relaciones entre *clases*. Esto significa que un objeto universidad puede consistir en varios departamentos aunque solo veas un "bloque" por cada entidad en el diagrama. La notación UML puede representar cantidades en ambos lados de las relaciones, pero no hay problema en omitirlas si están claras por el contexto.

Composición



Composición en UML. La universidad consta de departamentos.

La *composición* es un tipo específico de agregación en la que un objeto se compone de una o más instancias del otro. La diferencia entre ésta y otras relaciones está en que el componente sólo puede existir como parte del contenedor. En UML, la relación de composición se representa igual que la de agregación, pero con un diamante relleno en la base de la flecha.

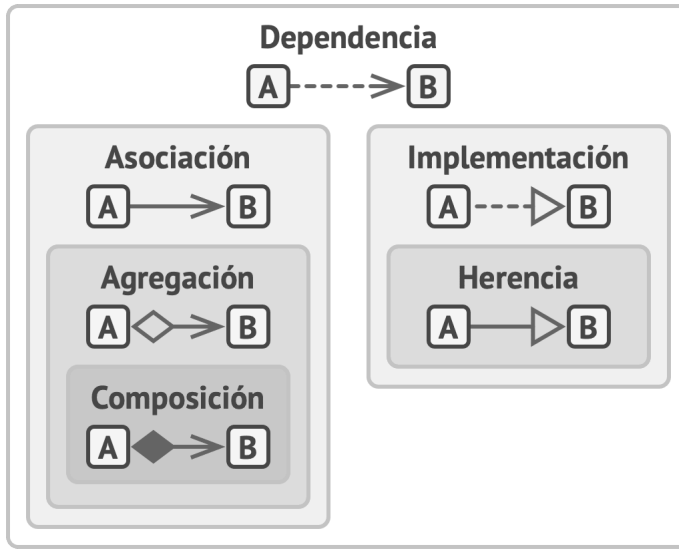
Observa que muchas personas utilizan a menudo el término "composición" cuando en realidad se refieren tanto a la agregación como a la composición. El ejemplo más notorio es el famoso principio "escoge composición antes que herencia". No es que la gente ignore la diferencia, sino más bien que la palabra "composición" (por ejemplo, "composición de objetos") suena más natural en el idioma inglés.

La visión global

Ahora que conocemos todos los tipos de relaciones entre objetos, veamos cómo se conectan entre sí. Esperemos que esto te ayude a responder preguntas como: "¿cuál es la diferencia entre agregación y composición?" o "¿la herencia es un tipo de dependencia?".

www.detodoprogramacion.org

- **Dependencia:** La clase A puede verse afectada por cambios en la clase B.
- **Asociación:** El objeto A conoce el objeto B. La clase A depende de B.
- **Agregación:** El objeto A conoce el objeto B y consiste en B. La clase A depende de B.
- **Composición:** El objeto A conoce el objeto B, consiste en B y gestiona el ciclo vital de B. La clase A depende de B.
- **Implementación:** La clase A define métodos declarados en la interfaz B. Los objetos A pueden tratarse como B. La clase A depende de B.
- **Herencia:** La clase A hereda la interfaz y la implementación de la clase B, pero puede extenderla. El objeto A puede tratarse como B. La clase A depende de B.



Relaciones entre objetos y clases: de la más débil a la más fuerte.

INTRODUCCIÓN
A LOS PATRONES

¿Qué es un patrón de diseño?

Los **patrones de diseño** son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. Son como planos prefabricados que se pueden personalizar para resolver un problema de diseño recurrente en tu código.

No se puede elegir un patrón y copiarlo en el programa como si se tratara de funciones o bibliotecas ya preparadas. El patrón no es una porción específica de código, sino un concepto general para resolver un problema particular. Puedes seguir los detalles del patrón e implementar una solución que encaje con las realidades de tu propio programa.

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución. El código del mismo patrón aplicado a dos programas distintos puede ser diferente.

Una analogía de un algoritmo sería una receta de cocina: ambos cuentan con pasos claros para alcanzar una meta. Por su parte, un patrón es más similar a un plano, ya que puedes observar cómo son su resultado y sus funciones, pero el orden exacto de la implementación depende de ti.

www.detodoprogramacion.org

¿En qué consiste el patrón?

La mayoría de los patrones se describe con mucha formalidad para que la gente pueda reproducirlos en muchos contextos. Aquí tienes las secciones que suelen estar presentes en la descripción de un patrón:

- El **propósito** del patrón explica brevemente el problema y la solución.
- La **motivación** explica en más detalle el problema y la solución que brinda el patrón.
- La **estructura** de las clases muestra cada una de las partes del patrón y el modo en que se relacionan.
- El **ejemplo de código** en uno de los lenguajes de programación populares facilita la asimilación de la idea que se esconde tras el patrón.

Algunos catálogos de patrones enumeran otros detalles útiles, como la aplicabilidad del patrón, los pasos de implementación y las relaciones con otros patrones.

Clasificación de los patrones

Los patrones de diseño varían en su complejidad, nivel de detalle y escala de aplicabilidad al sistema completo que se diseña. Me gusta la analogía de la construcción de carreteras: puedes hacer más segura una intersección instalando semáfo-

www.detodoprogramacion.org

ros o construyendo un intercambiador completo de varios niveles con pasajes subterráneos para peatones.

Los patrones más básicos y de más bajo nivel suelen llamarse *idioms*. Normalmente se aplican a un único lenguaje de programación.

Los patrones más universales y de más alto nivel son los *patrones de arquitectura*. Los desarrolladores pueden implementar estos patrones prácticamente en cualquier lenguaje. Al contrario que otros patrones, pueden utilizarse para diseñar la arquitectura de una aplicación completa.

Además, todos los patrones pueden clasificarse por su *propósito*. Este libro cubre tres grupos generales de patrones:

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

- Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.

- Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

www.detodoprogramacion.org

¿Quién inventó los patrones de diseño?

Esa es una buena, aunque imprecisa pregunta. Los patrones de diseño no son conceptos opacos y sofisticados, al contrario. Los patrones son soluciones habituales a problemas comunes en el diseño orientado a objetos. Cuando una solución se repite una y otra vez en varios proyectos, al final alguien le pone un nombre y explica la solución en detalle. Básicamente, así es como se descubre un patrón.

El concepto de los patrones fue descrito por Christopher Alexander en *El lenguaje de patrones* [1]. El libro habla de un "lenguaje" para diseñar el entorno urbano. Las unidades de este lenguaje son los patrones. Pueden describir lo altas que tienen que ser las ventanas, cuántos niveles debe tener un edificio, cuan grandes deben ser las zonas verdes de un barrio, etcétera.

La idea fue recogida por cuatro autores: Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm. En 1995, publicaron *Patrones de diseño* [2], en el que aplicaron el concepto de los patrones de diseño a la programación. El libro presentaba 23 patrones que resolvían varios problemas del diseño orientado a objetos y se convirtió en un éxito de ventas con rapidez. Al tener un título tan largo en inglés, la gente empezó a llamarlo "el libro de la 'gang of four' (banda de los cuatro)", lo que pronto se abrevió a "el libro GoF".

1. *El lenguaje de patrones*: **https://refactoring.guru/es/pattern-language-book**
2. *Patrones de diseño*: **https://refactoring.guru/es/gof-book**

Desde entonces se han descubierto decenas de nuevos patrones orientados a objetos. La "metodología del patrón" se hizo muy popular en otros campos de la programación, por lo que hoy en día existen muchos otros patrones no relacionados con el diseño orientado a objetos.

www.detodoprogramacion.org

¿Por qué debería aprender sobre patrones?

La realidad es que podrías trabajar durante años como programador sin conocer un solo patrón. Mucha gente lo hace. Incluso en ese caso, podrías estar implementando patrones sin saberlo. Así que, ¿por qué dedicar tiempo a aprenderlos?

- Los patrones de diseño son un juego de herramientas de **soluciones comprobadas** a problemas habituales en el diseño de software. Incluso aunque nunca te encuentres con estos problemas, conocer los patrones sigue siendo de utilidad, porque te enseña a resolver todo tipo de problemas utilizando principios del diseño orientado a objetos.

- Los patrones de diseño definen un lenguaje común que puedes utilizar con tus compañeros de equipo para comunicaros de forma más eficiente. Podrías decir: "Oh, utiliza un singleton para eso", y todos entenderían la idea de tu sugerencia. No habría necesidad de explicar qué es un singleton si conocen el patrón y su nombre.

www.detodoprogramacion.org

PRINCIPIOS DE DISEÑO DE SOFTWARE

Características del buen diseño

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

Reutilización de código

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

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

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

www.detodoprogramacion.org

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

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

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

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

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

Extensibilidad

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

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

www.detodoprogramacion.org

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

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

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

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

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

www.detodoprogramacion.org

Principios del diseño

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

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

www.detodoprogramacion.org

Encapsula lo que varía

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

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

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

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

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

www.detodoprogramacion.org

Encapsulación a nivel del método

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

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

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

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

  return total
```

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

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

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

  total += total * getTaxRate(order.country)

  return total

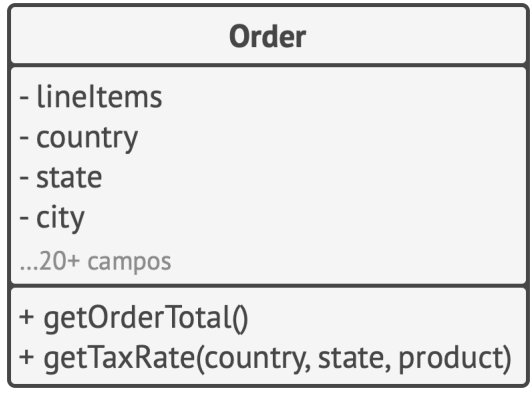
method getTaxRate(country) is
  if (country == "US")
    return 0.07 // Impuesto sobre la venta de EUA
  else if (country == "EU")
    return 0.20 // IVA europeo
  else
    return 0
```

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

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

Encapsulación a nivel de la clase

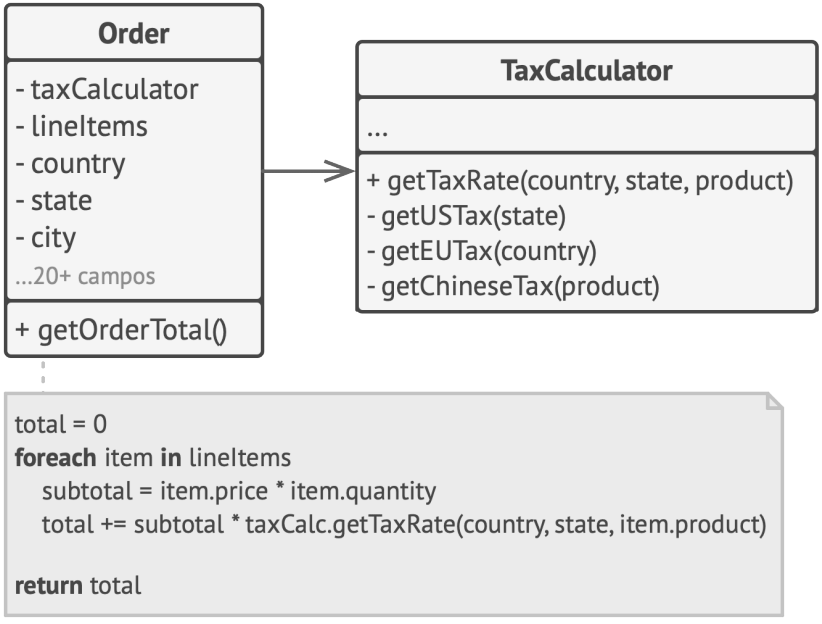
Con el tiempo puedes añadir más y más responsabilidades a un método que solía hacer algo sencillo. Estos comportamientos añadidos suelen venir con sus propios campos y métodos de ayuda que acaban nublando la responsabilidad principal de la clase contenedora. Si se extrae todo a una nueva clase se puede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



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

www.detodoprogramacion.org

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



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

Programa a una interfaz, no a una implementación

> Programa a una interfaz, no a una implementación. Depende de abstracciones, no de clases concretas.

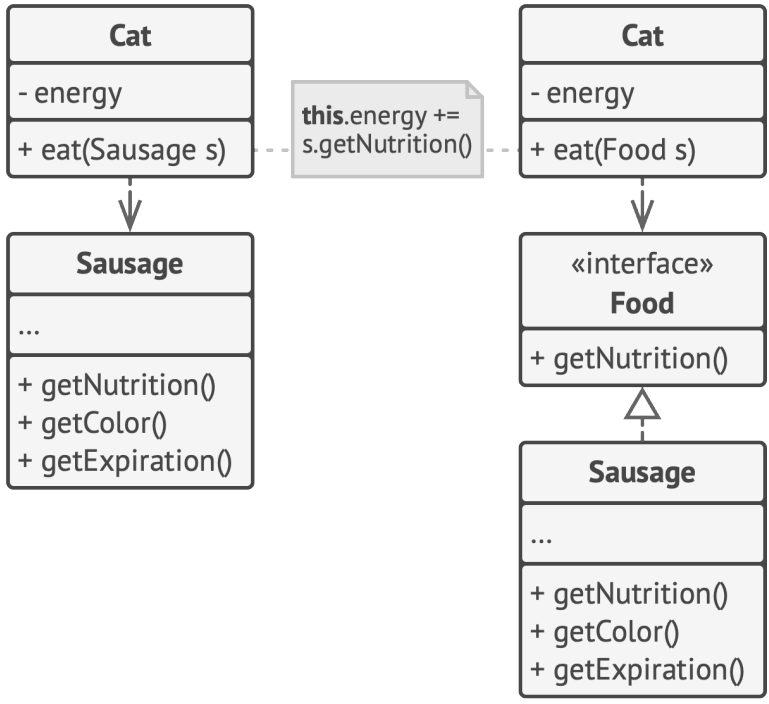
Sabrás que el diseño es lo suficientemente flexible si puedes extenderlo con facilidad y sin descomponer el código existente. Vamos a asegurarnos de que esta afirmación es correcta viendo un nuevo ejemplo con gatos. Un `Gato` que puede comer cualquier comida es más flexible que uno que sólo puede comer salchichas. Al primer gato le puedes dar salchichas porque éstas son un subgrupo de "cualquier comida", pero puedes extender el menú de ese gato con cualquier otra comida.

Cuando quieres hacer que dos clases colaboren, puedes comenzar por hacer una dependiente de la otra. Caramba, yo mismo suelo empezar haciendo eso. Sin embargo, hay otra forma más flexible de establecer la colaboración entre objetos:

1. Determina lo que necesita exactamente un objeto del otro: ¿qué métodos ejecuta?
2. Describe esos métodos en una nueva interfaz o clase abstracta.
3. Haz que la clase que es una dependencia implemente esta interfaz.

www.detodoprogramacion.org

4. Ahora, haz la segunda clase dependiente de esta interfaz en lugar de la clase concreta. Todavía puedes hacerlo funcionar con objetos de la clase original, pero ahora la conexión es mucho más flexible.



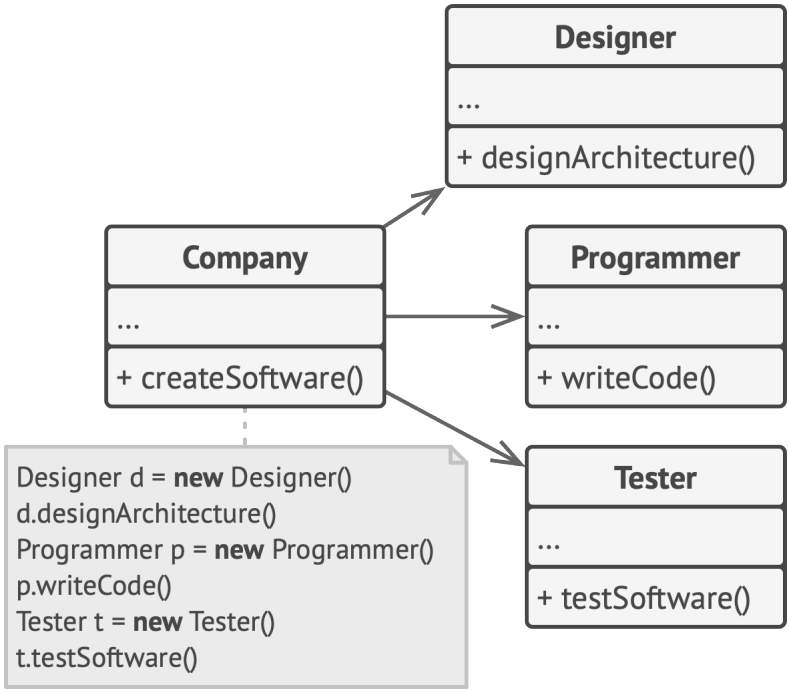
Antes y después de extraer la interfaz. El código de la derecha es más flexible que el de la izquierda, pero también es más complicado.

Tras hacer este cambio, probablemente no notarás un beneficio inmediato. Por el contrario, el código se ha vuelto más complicado de lo que era antes. No obstante, si consideras que éste puede ser un buen punto de extensión para alguna funcionalidad adicional, o que otras personas que utilizan tu código pueden desear extenderlo aquí, entonces adelante con ello.

www.detodoprogramacion.org

Ejemplo

Veamos otro ejemplo que ilustra que trabajar con objetos a través de interfaces puede ser más beneficioso que depender de sus clases concretas. Imagina que estás creando un simulador de empresa de desarrollo de software. Tienes distintas clases que representan varios tipos de empleados.

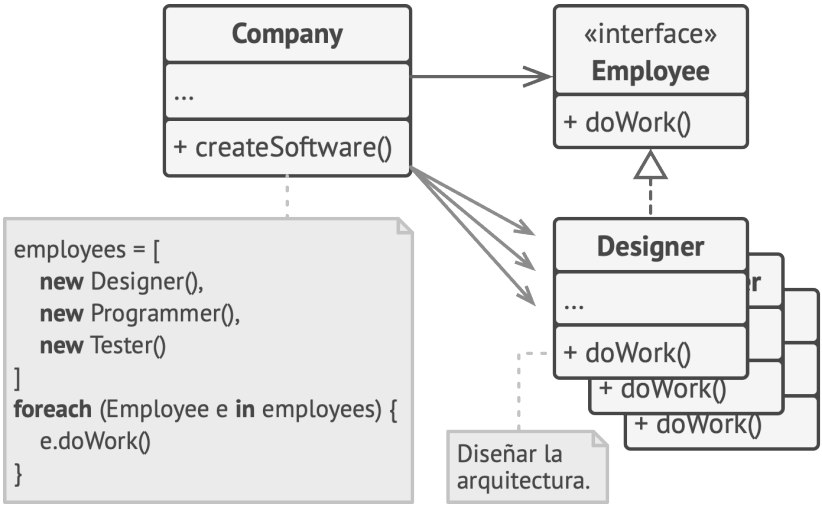


ANTES: todos las clases están fuertemente acopladas.

Al principio, la clase `Empresa` está fuertemente acoplada a clases de empleados concretos. Sin embargo, a pesar de la diferencia en sus implementaciones, podemos generalizar varios

métodos relacionados con el trabajo y extraer después una interfaz común para todas las clases de empleado.

Después de hacer esto, podemos aplicar el polimorfismo dentro de la clase `Empresa`, tratando varios objetos de empleado a través de la interfaz `Empleado`.



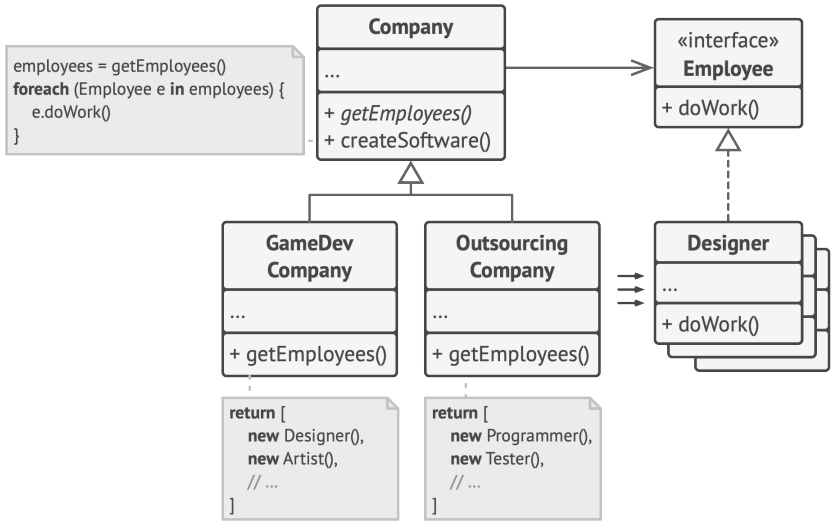
MEJOR: el polimorfismo nos ayudó a simplificar el código, pero el resto de la clase `Empresa` aún depende de las clases de empleados concretos.

La clase `Empresa` sigue acoplada a las clases de empleado. Esto no es bueno, porque si introducimos nuevos tipos de empresas que funcionan con otros tipos de empleados, necesitaremos sobrescribir la mayor parte de la clase `Empresa` en lugar de reutilizar ese código.

Para resolver este problema, podemos declarar el método para obtener empleados como *abstractos*. Cada empresa concreta

www.detodoprogramacion.org

implementará este método de forma diferente, creando únicamente los empleados que necesita.



DESPUÉS: el método primario de la clase `Empresa` es independiente de clases de empleado concretos. Los objetos de empleado se crean en subclases de empresas concretas.

Tras este cambio, la clase `Empresa` se vuelve independiente de varias clases de empleado. Ahora puedes extender esta clase e introducir nuevos tipos de empresas y empleados, a la vez que reutilizas una parte de la clase base empresa. Extender la clase base empresa no descompone el código existente que ya se basa en ella.

Por cierto, ¡acabas de ver la aplicación de un patrón de diseño en acción! Esto es un ejemplo del patrón *Factory Method*. No te preocupes: lo explicaremos en detalle más adelante.

www.detodoprogramacion.org

Favorece la composición sobre la herencia

La herencia es probablemente la forma más obvia y sencilla de reutilizar código entre clases. Tienes dos clases con el mismo código. Creas una clase base común para estas dos clases y colocas dentro el código similar. ¡Pan comido!

Lamentablemente, la herencia tiene sus contras, que a menudo no resultan aparentes hasta que tu programa tiene toneladas de clases y cambiarlo todo resulta muy complicado. Aquí tienes una lista de esos problemas.

- **Una subclase no puede reducir la interfaz de la superclase**. Tienes que implementar todos los métodos abstractos de la clase padre, incluso aunque no vayas a usarlos.

- **Al sobrescribir métodos debes asegurarte de que el nuevo comportamiento sea compatible con el de base**. Es importante porque los objetos de la subclase pueden pasarse a cualquier código que espere objetos de la superclase y no quieres que ese código se rompa.

- **La herencia rompe la encapsulación de la superclase** porque los detalles internos de la clase padre se hacen disponibles para la subclase. Puede darse una situación opuesta en la que un programador hace que una superclase conozca algunos de-

www.detodoprogramacion.org

talles de las subclases, con el objetivo de que las siguientes extensiones sean más sencillas.

- **Las subclases están fuertemente acopladas a superclases**. Cualquier cambio en una superclase puede descomponer la funcionalidad de las subclases.

- **Intentar reutilizar código mediante la herencia puede conducir a la creación de jerarquías de herencia paralelas**. Normalmente, la herencia sucede en una única dimensión. Pero cuando hay dos o más dimensiones, debes crear muchas combinaciones de clases, hinchando la jerarquía de clases hasta un tamaño ridículo.

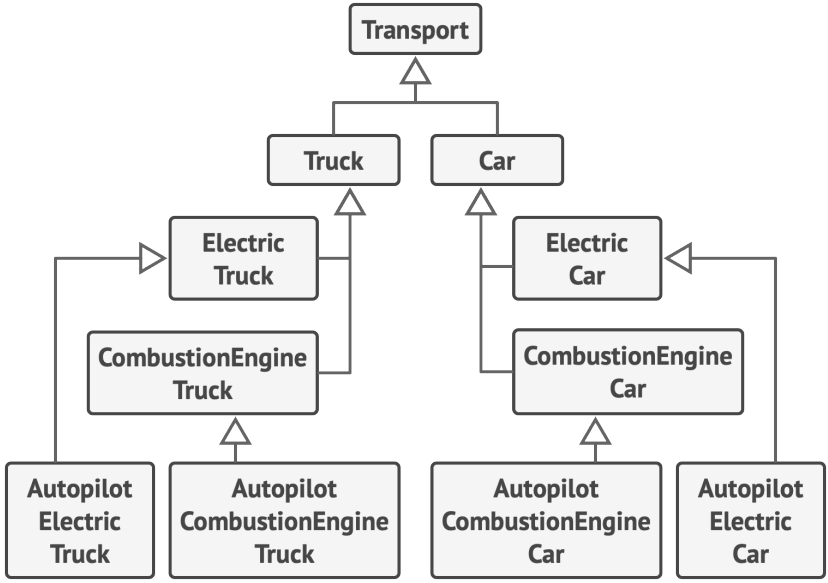
  Existe una alternativa a la herencia llamada *composición*. Mientras que la herencia representa la relación "is a" (es un/a) entre clases (un auto *es un* medio de transporte), la composición se basa en la relación "tiene un/a" (un auto *tiene un* motor).

  Debo mencionar que este principio también se aplica a la agregación, una variante más relajada de la composición en la que un objeto puede contener una referencia al otro pero no gestiona su ciclo vital. Aquí tienes un ejemplo: un auto *tiene un* conductor pero éste puede utilizar otro auto o caminar *sin el auto*.

www.detodoprogramacion.org

Ejemplo

Imagina que debes crear una aplicación de un catálogo para un fabricante de automóviles. La empresa fabrica autos y camiones; pueden ser eléctricos o de gasolina; todos los modelos pueden tener controles manuales o piloto automático.



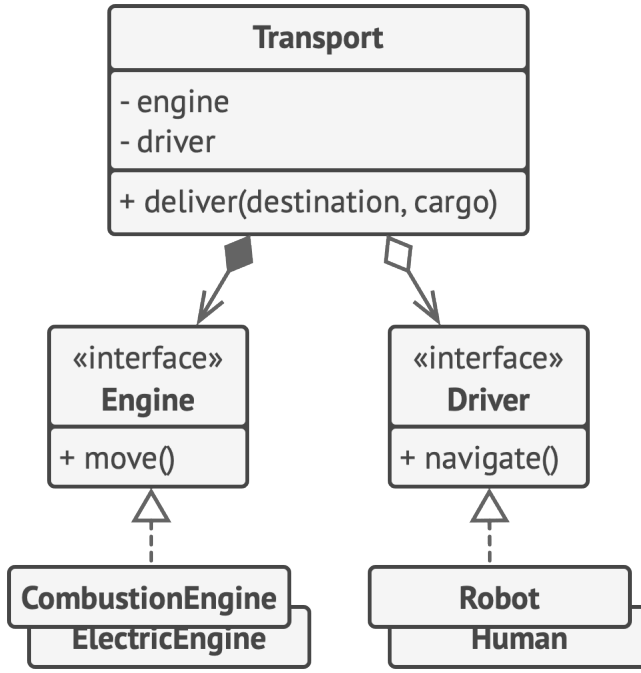
HERENCIA: extender una clase en varias dimensiones (tipo de carga × tipo de motor × tipo de navegación) puede provocar una explosión combinatoria de subclases.

Como puedes ver, cada parámetro adicional resulta en la multiplicación del número de subclases. Hay mucho código duplicado entre subclases porque una subclase no puede extender dos clases al mismo tiempo.

www.detodoprogramacion.org

Puedes resolver este problema con la composición. En lugar de que los objetos de auto implementen un comportamiento por su cuenta, pueden delegarlo a otros objetos.

La ventaja adicional es que puedes sustituir un comportamiento durante el tiempo de ejecución. Por ejemplo, puedes sustituir un objeto de motor vinculado a un objeto de auto asignando simplemente un objeto de motor distinto al auto.



COMPOSICIÓN: diferentes "dimensiones" de funcionalidad extraídas para ponerse dentro de sus propias jerarquías de clases..

Esta estructura de clases se parece al patrón *Strategy*, que explicaremos más adelante en este libro.

www.detodoprogramacion.org

Principios SOLID

Ahora que conoces los principios básicos de diseño, veamos cinco que se conocen popularmente como los principios SOLID. Robert Martin los presentó en el libro *Desarrollo ágil de software: principios, patrones y prácticas*[1].

SOLID es una regla mnemotécnica para cinco principios de diseño ideados para hacer que los diseños de software sean más comprensibles, flexibles y fáciles de mantener.

Como con todo en la vida, utilizar estos principios de forma descuidada puede hacer más mal que bien. El costo de aplicar estos principios en la arquitectura de un programa es que puede hacerlo más complicado de lo que debería. Dudo que exista un producto de software exitoso en el que se apliquen todos estos principios al mismo tiempo. Aspirar a estos principios es bueno, pero intenta siempre ser pragmático y no tomes todo lo escrito aquí como un dogma.

1. *Agile Software Development, Principles, Patterns, and Practices*: **https://refactoring.guru/es/principles-book**

www.detodoprogramacion.org

S Principio de responsabilidad única
ingle Responsibility Principle

> Una clase sólo debe tener una razón para cambiar.

Intenta hacer a cada clase responsable de una única parte de la funcionalidad proporcionada por el software, y haz que esa responsabilidad quede totalmente encapsulada por (también puedes decir *escondida dentro de*) la clase.

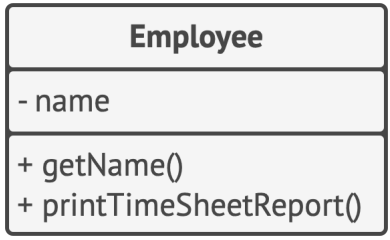
El principal objetivo de este principio es reducir la complejidad. No hace falta que inventes un diseño sofisticado para un programa que tiene unas 200 líneas de código. Con un puñado de buenos métodos te irá bien. Los verdaderos problemas surgen cuando tu programa crece y cambia constantemente. En cierto punto, las clases se vuelven tan grandes que ya no puedes recordar sus detalles. La navegación del código se ralentiza hasta ir a paso de tortuga y tienes que recorrer clases enteras o incluso un programa completo para encontrar algo específico. El número de entidades en el programa supera tu capacidad cerebral y sientes que pierdes el control sobre el código.

Hay más: si una clase hace demasiadas cosas, tienes que cambiarla cada vez que una de esas cosas cambia. Al hacerlo, te arriesgas a descomponer otras partes de la clase que no pretendías cambiar. Si sientes que te resulta difícil centrarte en un aspecto específico del programa cada vez, recuerda el principio de responsabilidad única y comprueba si es el momento de dividir algunas clases en partes.

www.detodoprogramacion.org

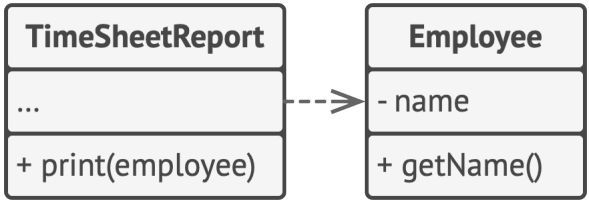
Ejemplo

La clase `Empleado` tiene varias razones para cambiar. La primera razón puede estar relacionada con el trabajo principal de la clase: gestionar información de los empleados. Pero hay otra razón: el formato del informe de horas de trabajo puede cambiar con el tiempo, lo que te obliga a cambiar el código de la clase.



ANTES: la clase contiene varios comportamientos diferentes.

Resuelve el problema moviendo el comportamiento relacionado con la impresión de informes de horas de trabajo a una clase separada. Este cambio te permite mover otros elementos relacionados con el informe a la nueva clase.



DESPUÉS: el comportamiento adicional está en su propia clase.

www.detodoprogramacion.org

O Principio de abierto/cerrado
pen/Closed Principle

> Las clases deben estar abiertas a la extensión pero cerradas a la modificación.

La idea fundamental de este principio es evitar que el código existente se descomponga cuando implementas nuevas funciones.

Una clase está *abierta* si puedes extenderla, crear una subclase y hacer lo que quieras con ella (añadir nuevos métodos o campos, sobrescribir el comportamiento base, etc.). Algunos lenguajes de programación te permiten restringir en mayor medida la extensión de una clase con palabras clave como `final`. Después de esto, la clase ya no está abierta. Al mismo tiempo, la clase está *cerrada* (también puedes decir *completa*) si está lista al 100 % para que otras clases la utilicen; su interfaz está claramente definida y no se cambiará en el futuro.

La primera vez que estudié este principio me confundieron las palabras *abierta* y *cerrada*, porque suenan mutuamente excluyentes. Pero en lo que respecta a este principio, una clase puede estar al mismo tiempo abierta (para la extensión) y cerrada (para la modificación).

Cuando una clase se ha desarrollado, probado, revisado e incluido en un framework o utilizada en una aplicación de cualquier otro modo, es arriesgado intentar juguetear con su código. En lugar de cambiar directamente el código de la clase, puedes crear una sub-

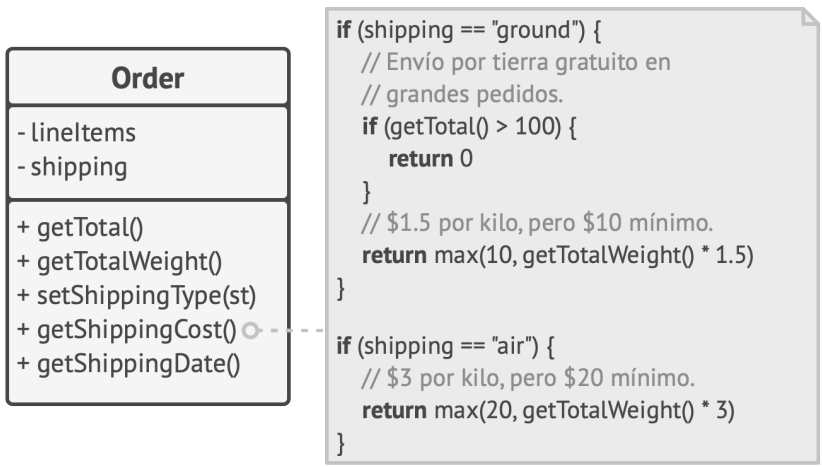
www.detodoprogramacion.org

clase y sobrescribir las partes de la clase original que quieras que se comporten de otra forma. Lograrás tu objetivo sin descomponer clientes existentes de la clase original.

Este principio no está pensado para aplicarse a todos los cambios de una clase. Si sabes que hay un error en la clase, debes arreglarlo; no crees una subclase para ello. Una clase hija no debe ser responsable de los problemas de la clase padre.

Ejemplo

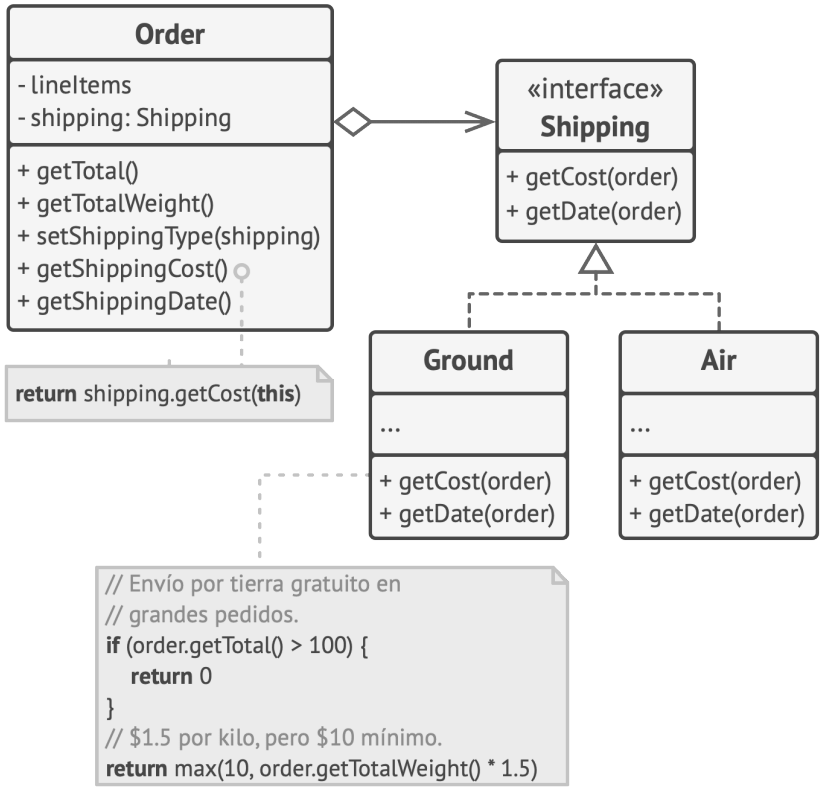
Tienes una aplicación de comercio electrónico con una clase `Pedido` que calcula los costos de envío, y todos los métodos de envío están incrustados dentro de la clase. Si necesitas añadir un nuevo método de envío, tienes que cambiar el código de la clase `Pedido`, arriesgándote a descomponerlo.



ANTES: tienes que cambiar la clase `Pedido` siempre que añades un nuevo método de envío a la aplicación.

www.detodoprogramacion.org

Puedes resolver el problema aplicando el patrón *Strategy*. Empieza extrayendo métodos de envío y colocándolos dentro de clases separadas con una interfaz común.



DESPUÉS: añadir un nuevo método de envío no requiere cambiar clases existentes.

Ahora, cuando necesites implementar un nuevo método de envío, puedes derivar una nueva clase de la interfaz `Envíos` sin tocar el código de la clase `Pedido`. El código cliente de la clase `Pedido` vinculará los pedidos con un objeto de envío de

la nueva clase cuando el usuario seleccione estos métodos de envío en la UI.

Esta solución tiene la ventaja adicional de que te permite mover el cálculo del tiempo de entrega a clases más relevantes, de acuerdo con el *principio de responsabilidad única*.

www.detodoprogramacion.org

Principio de sustitución de Liskov[1]
Liskov Substitution Principle

> Al extender una clase, recuerda que debes tener la capacidad de pasar objetos de las subclases en lugar de objetos de la clase padre, sin descomponer el código cliente.

Esto significa que la subclase debe permanecer compatible con el comportamiento de la superclase. Al sobrescribir un método, extiende el comportamiento base en lugar de sustituirlo con algo totalmente distinto.

El principio de sustitución es un grupo de comprobaciones que ayudan a predecir si una subclase permanece compatible con el código que pudo funcionar con objetos de la superclase. Este concepto es fundamental al desarrollar bibliotecas y *frameworks*, porque otras personas utilizarán tus clases y no podrás acceder directamente ni cambiar su código.

Al contrario que otros principios de diseño que están muy abiertos a la interpretación, el principio de sustitución incluye un grupo de requisitos formales para las subclases y, específicamente, para sus métodos. Repasemos esta lista en detalle.

1. Este principio debe su nombre a Barbara Liskov, que lo definió en 1987 en su trabajo *Data abstraction and hierarchy*: https://refactoring.guru/liskov/dah

- **Los tipos de parámetros en el método de una subclase deben *coincidir* o ser *más abstractos* que los tipos de parámetros del método de la superclase**. ¿Suena confuso? Veamos un ejemplo.

    - Digamos que hay una clase con un método que debe alimentar gatos: `alimentar(Gato c)`. El código cliente siempre pasa objetos de gatos a este método.

    - **Bien:** Digamos que creaste una subclase que sobrescribió el método para que pueda alimentar a cualquier animal (una superclase de gatos): `alimentar(Animal c)`. Ahora, si pasas un objeto a esta subclase en lugar de un objeto de la superclase al código cliente, todo funcionará bien. El método puede alimentar a todos los animales, por lo que alimentará a todos los gatos pasados por el cliente.

    - **Mal:** Creaste otras subclase y restringiste el método de alimentación para que acepte únicamente gatos de Bengala (una subclase de gatos): `alimentar(GatoDeBengala c)`. ¿Qué le pasará al código cliente si lo vinculas con un objeto como éste, en lugar de hacerlo con la clase original? Debido a que el método sólo puede alimentar una raza específica de gatos, no servirá a los gatos genéricos pasados por el cliente, descomponiendo toda la funcionalidad relacionada.

- **El tipo de retorno en el método de una subclase debe *coincidir* o ser un *subtipo* del tipo de retorno del método de la superclase**. Como puedes ver, los requisitos para un tipo de retorno son inversos a los requisitos para los tipos de parámetros.

  - Digamos que tienes una clase con el método `comprarGato(): Gato`. El código cliente espera recibir cualquier gato como resultado de ejecutar este método.

  - **Bien:** Una subclase sobrescribe el método de esta forma: `comprarGato(): GatoDeBengala`. El cliente obtiene un gato de Bengala, que sigue siendo un gato, por lo que todo está bien.

  - **Mal:** Una subclase sobrescribe el método de esta forma: `comprarGato(): Animal`. Ahora el código cliente se descompone porque recibe un animal genérico desconocido (¿Un lagarto? ¿Un oso?) que no encaja con la estructura diseñada para un gato.

  Del mundo de los lenguajes de programación de tipado dinámico podemos extraer otro ejemplo negativo: el método base devuelve una cadena, pero el método sobrescrito devuelve un número.

- **Un método de una subclase no debe arrojar tipos de excepciones que no se espere que arroje el método base**. En otras palabras, los tipos de excepciones deben *coincidir* o ser *subtipos* de los que el método base es capaz de arrojar. Esta regla proviene del hecho de que los bloques `try–catch` en el código cliente se dirigen a los tipos específicos de excepciones que más probablemente arrojará el método base. Por lo tanto, una excepción inesperada puede colarse entre las líneas defensivas del código cliente y destrozar la aplicación.

www.detodoprogramacion.org

> En la mayoría de lenguajes de programación modernos, sobre todo los de tipado estático (Java, C# y otros), estas reglas vienen integradas en el lenguaje. No podrás compilar un programa que viola estas reglas.

- **Una subclase no debe fortalecer las condiciones previas**. Por ejemplo, el método base tiene un parámetro con el tipo `int`. Si una subclase sobrescribe este método y requiere que el valor de un argumento pasado al método sea positivo (lanzando una excepción si el valor es negativo), esto amplía las condiciones previas. El código cliente, que solía funcionar bien pasando números negativos al método, ahora se descompone si empieza a funcionar con un objeto de esta subclase.

- **Una subclase no debe debilitar las condiciones posteriores**. Digamos que tienes una clase con un método que trabaja con una base de datos. Se supone que un método de la clase siempre debe cerrar todas las conexiones de las bases de datos abiertas al devolver un valor.

  Creaste una subclase y la cambiaste, de modo que las conexiones de la base de datos permanezcan abiertas para poder reutilizarlas. Pero puede que el cliente no sepa nada de tus intenciones. Debido a que espera que los métodos cierren todas las conexiones, puede que simplemente finalice el programa después de invocar el método, contaminando un sistema con conexiones de bases de datos fantasma.

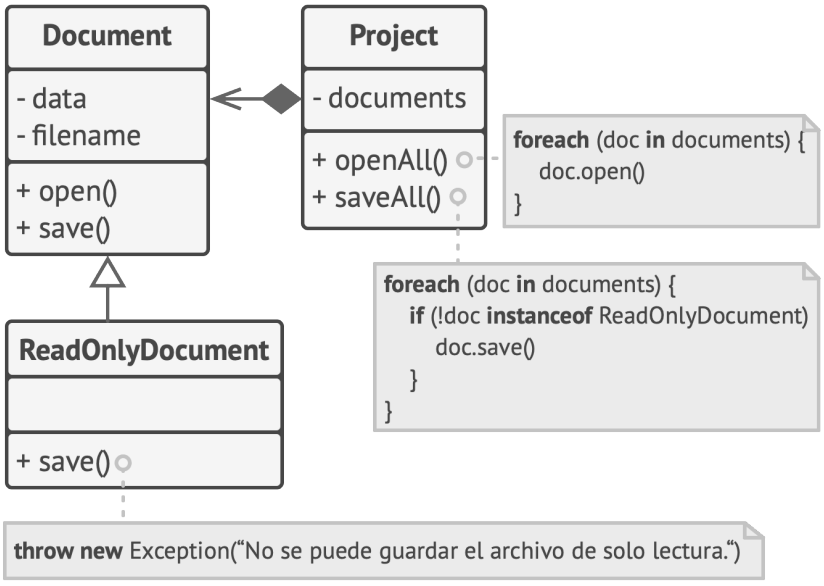
- **Los invariantes de una superclase deben preservarse**. Probablemente esta sea la regla menos formal de todas. Los *invariantes* son condiciones en las cuales un objeto tiene sentido. Por ejemplo, los invariantes de un gato son tener cuatro patas, una cola, la capacidad de maullar, etc. La parte confusa sobre los invariantes es que, aunque pueden definirse explícitamente en la forma de contratos de interfaz o un grupo de aserciones dentro de los métodos, también pueden resultar implícitos por ciertas pruebas de unidad y expectativas del código cliente.

  La regla de los invariantes es la más fácil de vulnerar, porque puedes no comprender o no ser consciente de todos los invariantes de una clase compleja. Por lo tanto, la forma más segura de extender una clase consiste en introducir nuevos campos y métodos y no meterse con miembros existentes de la superclase. Po supuesto, esto no siempre es posible en la vida real.

- **Una subclase no debe cambiar los valores de campos privados de la superclase**. *¿Qué? ¿Cómo es posible?* Resulta que algunos lenguajes de programación te permiten acceder a miembros privados de una clase mediante mecanismos reflexivos. Otros lenguajes (Python, JavaScript) no tienen protección en absoluto para los miembros privados.

Ejemplo

Veamos un ejemplo de una jerarquía de clases de documento que violan el principio de sustitución.

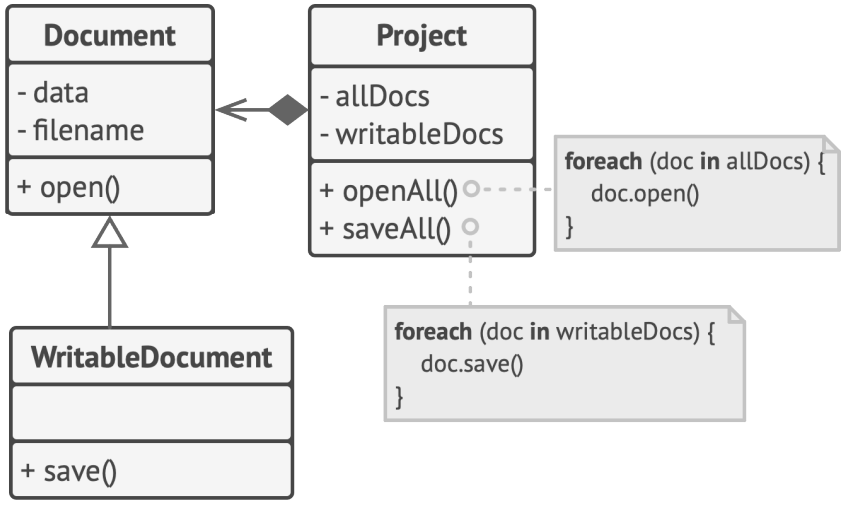


ANTES: el método guardar no tiene sentido en un documento de solo lectura, por lo que la subclase intenta resolverlo reiniciando el comportamiento base en el método sobrescrito.

El método `guardar` de la subclase `DocumentosDeSoloLectura` arroja una excepción si alguien intenta invocarlo. El método base no tiene esta restricción. Esto significa que el código cliente se descompondrá si no comprobamos el tipo de documento antes de guardarlo.

El código resultante también viola el principio de abierto/cerrado, ya que el código cliente se vuelve dependiente de clases

concretas de los documentos. Si introduces una nueva subclase de documento, necesitarás cambiar el código cliente para que lo soporte.



DESPUÉS: el problema se resuelve tras hacer a la clase de documento de solo lectura la clase base de la jerarquía.

Puedes resolver el problema rediseñando la jerarquía de clases: una subclase debe extender el comportamiento de una superclase, por lo tanto, el documento de solo lectura se convierte en la clase base de la jerarquía. El documento de escritura es ahora una subclase que extiende la clase base y añade el comportamiento de guardar.

www.detodoprogramacion.org

I Principio de segregación de la interfaz
nterface Segregation Principle

> No se debe forzar a los clientes a depender de métodos que no utilizan.

Intenta que tus interfaces sean lo suficientemente estrechas para que las clases del cliente no tengan que implementar comportamientos que no necesitan.

Según el principio de segregación de la interfaz, debes desintegrar las interfaces "gruesas" hasta crear otras más detalladas y específicas. Los clientes deben implementar únicamente aquellos métodos que necesitan de verdad. De lo contrario, un cambio en una interfaz "gruesa" descompondrá incluso clientes que no utilizan los métodos cambiados.

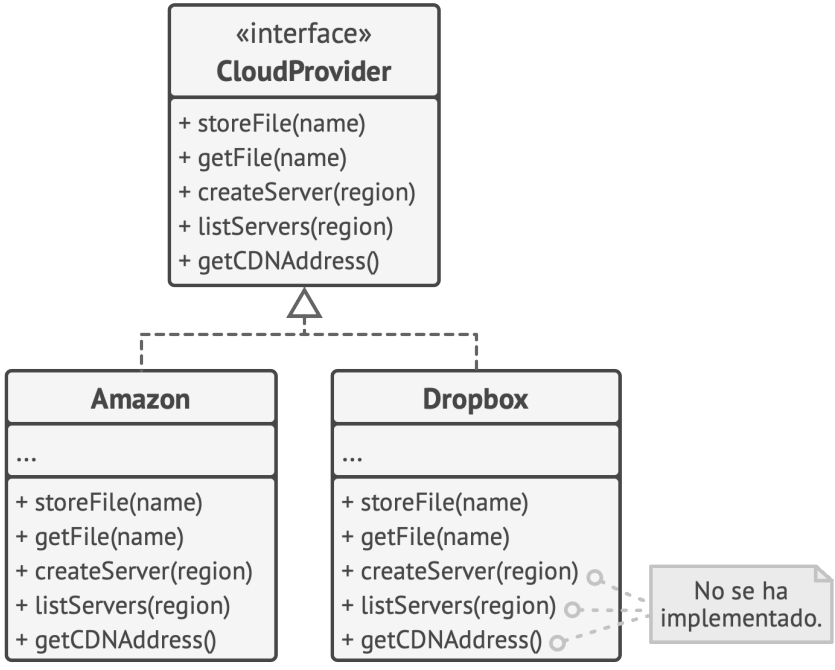
La herencia de clases permite a una clase tener una única superclase, pero no limita el número de interfaces que la clase puede implementar al mismo tiempo. Por lo tanto, no hay necesidad de amontonar toneladas de métodos no relacionados en una única interfaz. Divídela en varias interfaces más refinadas; puedes implementarlas todas en una única clase si es necesario. Sin embargo, a algunas clases no les importa que sólo implementes una de ellas.

www.detodoprogramacion.org

Ejemplo

Imagina que creaste una biblioteca que facilita la integración de aplicaciones con varios proveedores de computación en la nube. Aunque en la versión inicial sólo soportaba Amazon Cloud, cubría todos los servicios y funciones de la nube.

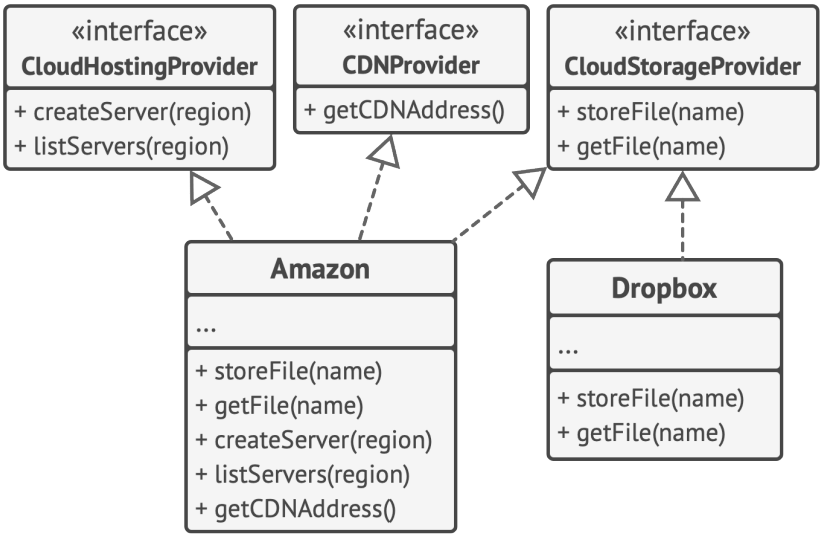
En aquel momento asumiste que todos los proveedores en la nube tienen la misma amplitud de espectro de funciones que Amazon. Pero cuando hubo que implementar soporte para otro proveedor, resultó que la mayoría de las interfaces de la biblioteca eran demasiado amplias. Algunos métodos describen funciones que otros proveedores de la nube no incluyen.



ANTES: no todos los clientes pueden satisfacer los requisitos de la abotargada interfaz.

www.detodoprogramacion.org

Aunque puedes implementar estos métodos y colocar algunas maquetas, no será una solución muy limpia. La mejor solución es dividir la interfaz en partes. Las clases capaces de implementar la interfaz original pueden ahora implementar varias interfaces refinadas. Otras clases pueden implementar únicamente aquellas interfaces que tienen métodos que tienen sentido para ellas.



DESPUÉS: una interfaz abotargada se divide en un grupo de interfaces más detalladas.

Como sucede con otros principios, con éste también puedes ir demasiado lejos. No dividas una interfaz que ya es bastante específica. Recuerda que, cuantas más interfaces crees, más se complicará tu código. Mantén el equilibrio.

www.detodoprogramacion.org

D Principio de inversión de la dependencia
ependency Inversion Principle

> Las clases de alto nivel no deben depender de clases de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.

Al diseñar software, normalmente puedes distinguir entre dos niveles de clases.

- Las **clases de bajo nivel** implementan operaciones básicas, como trabajar con un disco, transferir datos por una red, conectar con una base de datos, etc.
- Las **clases de alto nivel** contienen la lógica de negocio compleja que ordena a las clases de bajo nivel que hagan algo.

Algunas personas diseñan primero las clases de bajo nivel y sólo entonces comienzan a trabajar con las de alto nivel. Esto es muy habitual cuando empiezas a desarrollar un prototipo de un nuevo sistema y no estás seguro de lo que es posible a alto nivel, porque el contenido de bajo nivel aún no está implementado o claro. Con este sistema, las clases de la lógica de negocio tienden a hacerse dependientes de clases primarias de bajo nivel.

www.detodoprogramacion.org

El principio de inversión de la dependencia sugiere cambiar la dirección de esta dependencia.

1. Para empezar, debes describir interfaces para operaciones de bajo nivel en las que se basarán las clases de alto nivel, preferiblemente en términos de negocio. Por ejemplo, la lógica de negocio debe invocar un método `abrirInforme(archivo)` en lugar de una serie de métodos `abrirArchivo(x)`, `leerBytes(n)`, `cerrarArchivo(x)`. Estas interfaces cuentan como de alto nivel.
2. Ahora puedes hacer las clases de alto nivel dependientes de esas interfaces, en lugar de clases concretas de bajo nivel. Esta dependencia será mucho más débil que la original.
3. Una vez que las clases de bajo nivel implementan esas interfaces, se vuelven dependientes del nivel de la lógica de negocio, invirtiendo la dirección de la dependencia original.

El principio de inversión de la dependencia suele ir de la mano del *principio de abierto/cerrado*: puedes extender clases de bajo nivel para utilizarlas con distintas clases de lógica de negocio sin descomponer clases existentes.

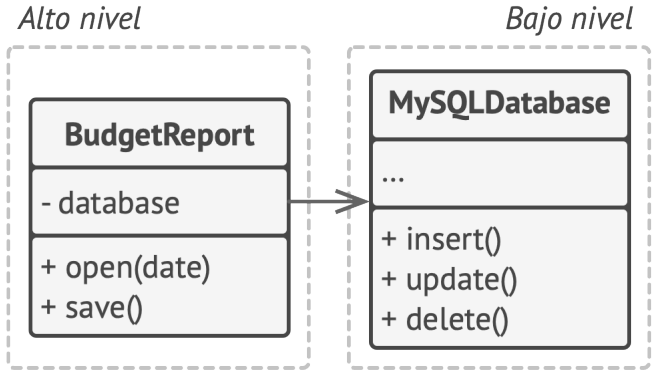
Ejemplo

Descargado de: www.detodoprogramacion.org

En este ejemplo, la clase de alto nivel — que se ocupa de informes presupuestarios — utiliza una clase de base de datos de bajo nivel para leer y almacenar su información. Esto significa que cualquier cambio en la clase de bajo nivel, como en el caso del lanzamiento de una nueva versión del servi-

www.detodoprogramacion.org

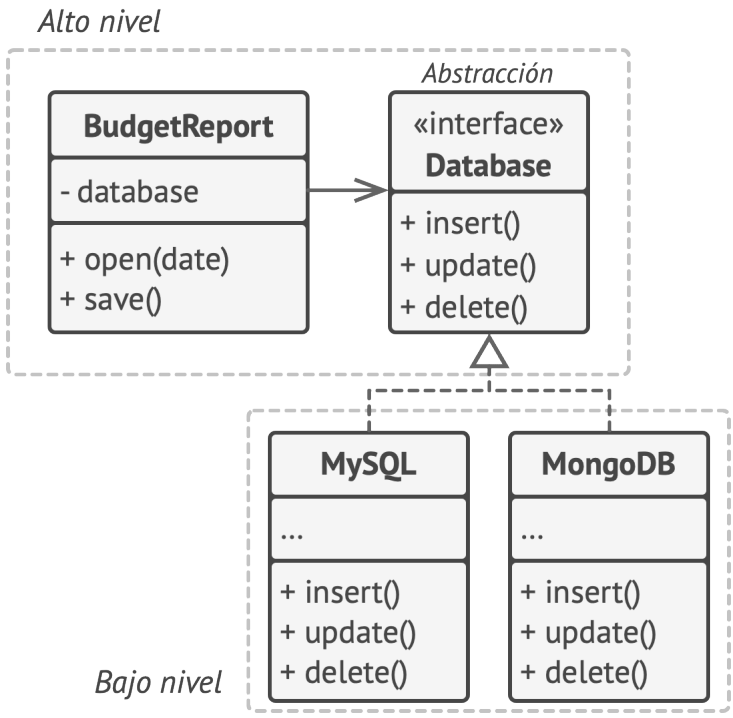
dor de la base de datos, puede afectar a la clase de alto nivel, que no tiene por qué conocer los detalles de almacenamiento de datos.



ANTES: una clase de alto nivel depende de una clase de bajo nivel.

Puedes arreglar este problema creando una interfaz de alto nivel que describa operaciones de leer/escribir y haciendo que la clase de informes utilice esa interfaz en lugar de la clase de bajo nivel. Después puedes cambiar o extender la clase de bajo nivel original para implementar la nueva interfaz de leer/escribir declarada por la lógica de negocio.

www.detodoprogramacion.org



DESPUÉS: las clases de bajo nivel dependen de una abstracción de alto nivel.

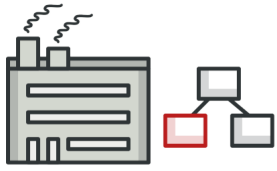
Como resultado, la dirección de la dependencia original se ha invertido: las clases de bajo nivel dependen ahora de abstracciones de alto nivel.

EL CATÁLOGO DE PATRONES DE DISEÑO

www.detodoprogramacion.org

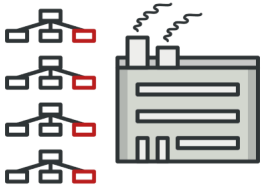
Patrones creacionales

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



Factory Method

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



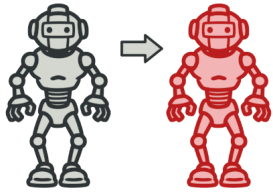
Abstract Factory

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



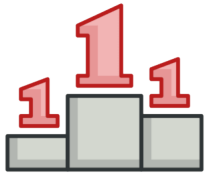
Builder

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



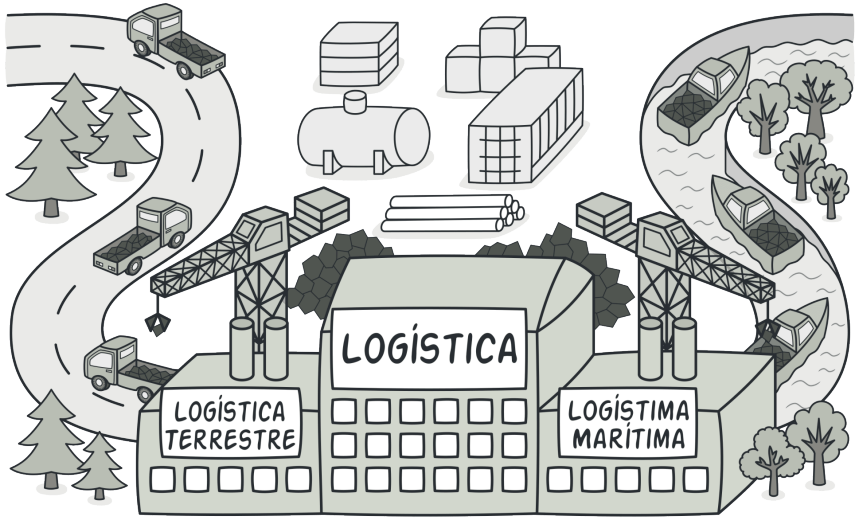
Prototype

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



Singleton

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



FACTORY METHOD

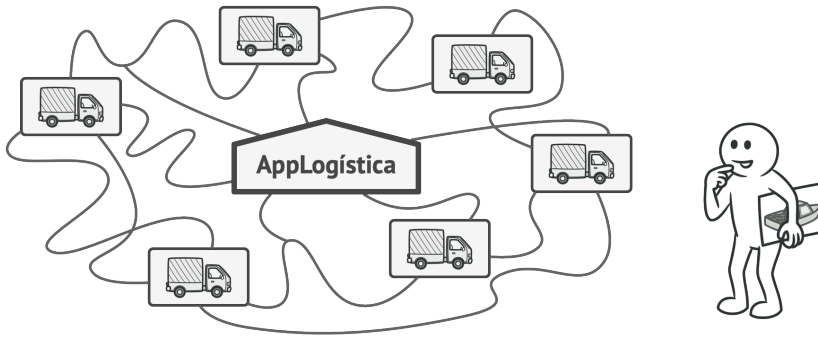
También llamado: Método fábrica, Constructor virtual

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

☹ Problema

Imagina que estás creando una aplicación de gestión logística. La primera versión de tu aplicación sólo es capaz de manejar el transporte en camión, por lo que la mayor parte de tu código se encuentra dentro de la clase `Camión`.

Al cabo de un tiempo, tu aplicación se vuelve bastante popular. Cada día recibes decenas de peticiones de empresas de transporte marítimo para que incorpores la logística por mar a la aplicación.



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

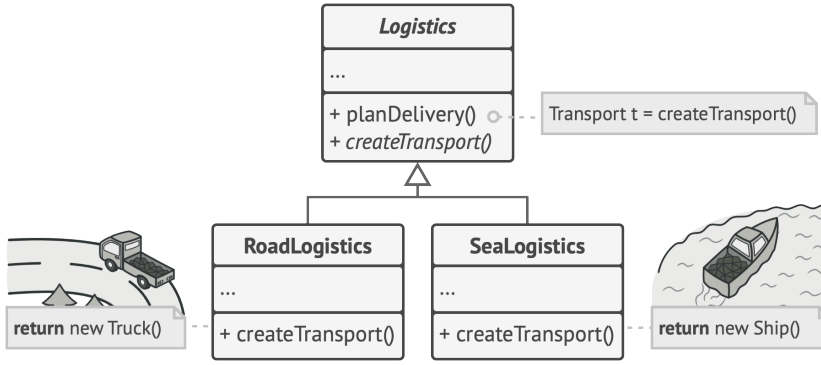
Estupendo, ¿verdad? Pero, ¿qué pasa con el código? En este momento, la mayor parte de tu código está acoplado a la clase `Camión`. Para añadir barcos a la aplicación habría que hacer cambios en toda la base del código. Además, si más tarde decides añadir otro tipo de transporte a la aplicación, probablemente tendrás que volver a hacer todos estos cambios.

www.detodoprogramacion.org

Al final acabarás con un código bastante sucio, plagado de condicionales que cambian el comportamiento de la aplicación dependiendo de la clase de los objetos de transporte.

😊 Solución

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

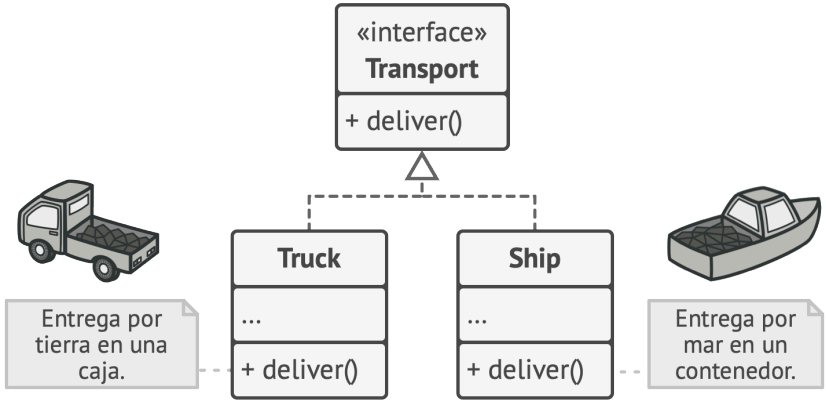


Las subclases pueden alterar la clase de los objetos devueltos por el método fábrica.

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

www.detodoprogramacion.org

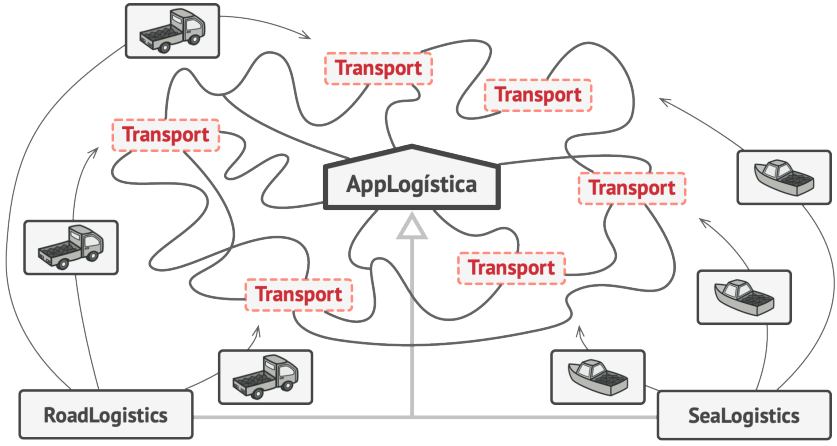
No obstante, hay una pequeña limitación: las subclases sólo pueden devolver productos de distintos tipos si dichos productos tienen una clase base o interfaz común. Además, el método fábrica en la clase base debe tener su tipo de retorno declarado como dicha interfaz.



Todos los productos deben seguir la misma interfaz.

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

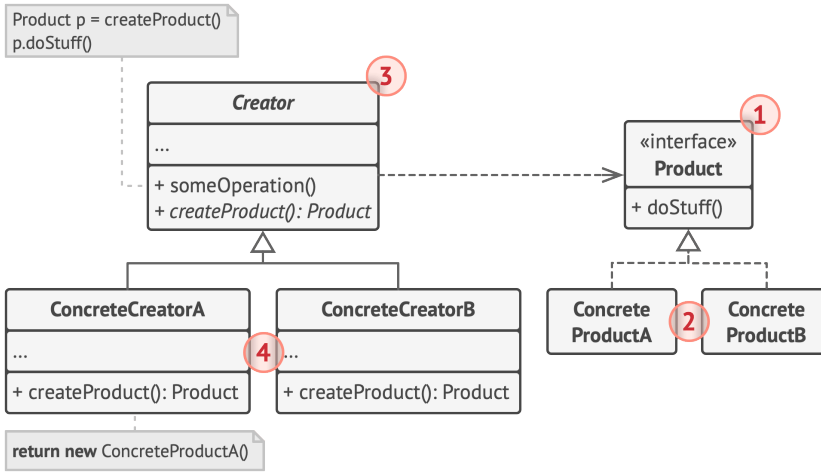
www.detodoprogramacion.org



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

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

Estructura



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

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

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

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

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

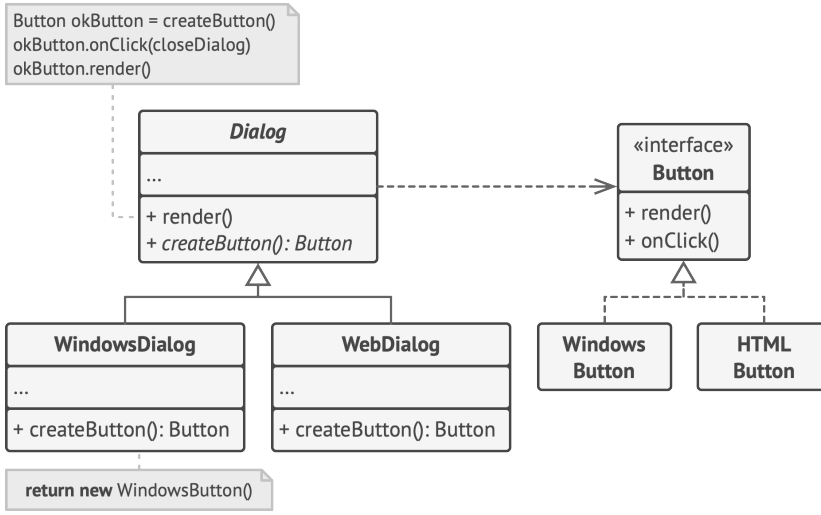
4. Los **Creadores Concretos** sobrescriben el Factory Method base, de modo que devuelva un tipo diferente de producto.

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

Pseudocódigo

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

www.detodoprogramacion.org



Ejemplo del diálogo multiplataforma.

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

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

www.detodoprogramacion.org

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

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

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

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

```
    Button okButton = createButton()
    // Ahora utiliza el producto.
    okButton.onClick(closeDialog)
    okButton.render()


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

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


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

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

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

```

El Mundo de la Programación en tus Manos...!
DETODOPROGRAMACION.ORG

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

class Application is
  field dialog: Dialog

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

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

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

Aplicabilidad

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

El patrón Factory Method separa el código de construcción de producto del código que hace uso del producto. Por ello, es más fácil extender el código de construcción de producto de forma independiente al resto del código.

Por ejemplo, para añadir un nuevo tipo de producto a la aplicación, sólo tendrás que crear una nueva subclase creadora y sobrescribir el Factory Method que contiene.

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

La herencia es probablemente la forma más sencilla de extender el comportamiento por defecto de una biblioteca o un framework. Pero, ¿cómo reconoce el framework si debe utilizar tu subclase en lugar de un componente estándar?

La solución es reducir el código que construye componentes en todo el framework a un único patrón Factory Method y permitir que cualquiera sobrescriba este método además de extender el propio componente.

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

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



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



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

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

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

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

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

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

Cómo implementarlo

1. Haz que todos los productos sigan la misma interfaz. Esta interfaz deberá declarar métodos que tengan sentido en todos los productos.

2. Añade un patrón Factory Method vacío dentro de la clase creadora. El tipo de retorno del método deberá coincidir con la interfaz común de los productos.

3. Encuentra todas las referencias a constructores de producto en el código de la clase creadora. Una a una, sustitúyelas por invocaciones al Factory Method, mientras extraes el código de creación de productos para colocarlo dentro del Factory Method.

   Puede ser que tengas que añadir un parámetro temporal al Factory Method para controlar el tipo de producto devuelto.

   A estas alturas, es posible que el aspecto del código del Factory Method luzca bastante desagradable. Puede ser que tenga un operador `switch` largo que elige qué clase de producto instanciar. Pero, no te preocupes, lo arreglaremos enseguida.

4. Ahora, crea un grupo de subclases creadoras para cada tipo de producto enumerado en el Factory Method. Sobrescribe el Factory Method en las subclases y extrae las partes adecuadas de código constructor del método base.

5. Si hay demasiados tipos de producto y no tiene sentido crear subclases para todos ellos, puedes reutilizar el parámetro de control de la clase base en las subclases.

   Por ejemplo, imagina que tienes la siguiente jerarquía de clases: la clase base `Correo` con las subclases `CorreoAéreo`

www.detodoprogramacion.org

y `CorreoTerrestre` y la clase `Transporte` con `Avión`, `Camión` y `Tren`. La clase `CorreoAéreo` sólo utiliza objetos `Avión`, pero `CorreoTerrestre` puede funcionar tanto con objetos `Camión`, como con objetos `Tren`. Puedes crear una nueva subclase (digamos, `CorreoFerroviario`) que gestione ambos casos, pero hay otra opción. El código cliente puede pasar un argumento al Factory Method de la clase `CorreoTerrestre` para controlar el producto que quiere recibir.

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

⚖ Pros y contras

✓ Evitas un acoplamiento fuerte entre el creador y los productos concretos.

✓ *Principio de responsabilidad única*. Puedes mover el código de creación de producto a un lugar del programa, haciendo que el código sea más fácil de mantener.

✓ *Principio de abierto/cerrado*. Puedes incorporar nuevos tipos de productos en el programa sin descomponer el código cliente existente.

✕ Puede ser que el código se complique, ya que debes incorporar una multitud de nuevas subclases para implementar el patrón. La situación ideal sería introducir el patrón en una jerarquía existente de clases creadoras.

www.detodoprogramacion.org

⇄ Relaciones con otros patrones

- Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).

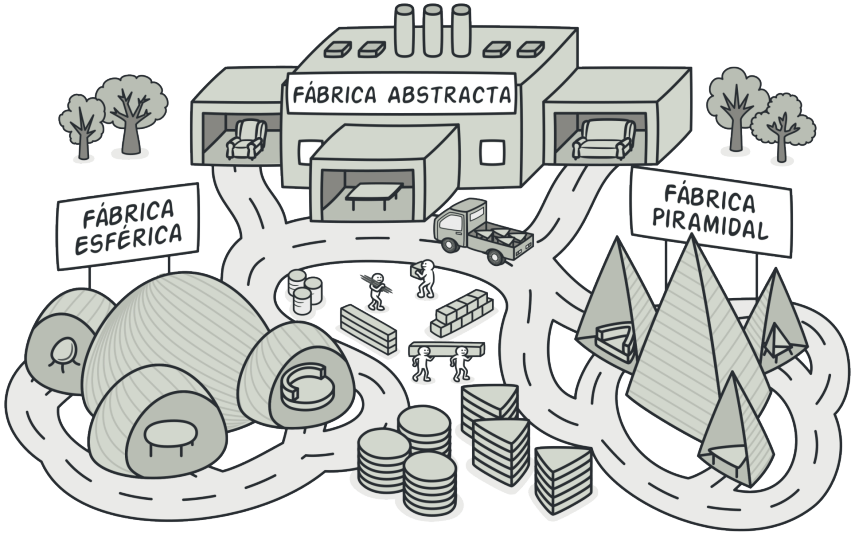
- Las clases del **Abstract Factory** a menudo se basan en un grupo de **métodos de fábrica**, pero también puedes utilizar **Prototype** para escribir los métodos de estas clases.

- Puedes utilizar el patrón **Factory Method** junto con el **Iterator** para permitir que las subclases de la colección devuelvan distintos tipos de iteradores que sean compatibles con las colecciones.

- **Prototype** no se basa en la herencia, por lo que no presenta sus inconvenientes. No obstante, *Prototype* requiere de una inicialización complicada del objeto clonado. **Factory Method** se basa en la herencia, pero no requiere de un paso de inicialización.

- **Factory Method** es una especialización del **Template Method**. Al mismo tiempo, un *Factory Method* puede servir como paso de un gran *Template Method*.

www.detodoprogramacion.org



ABSTRACT FACTORY

También llamado: Fábrica abstracta

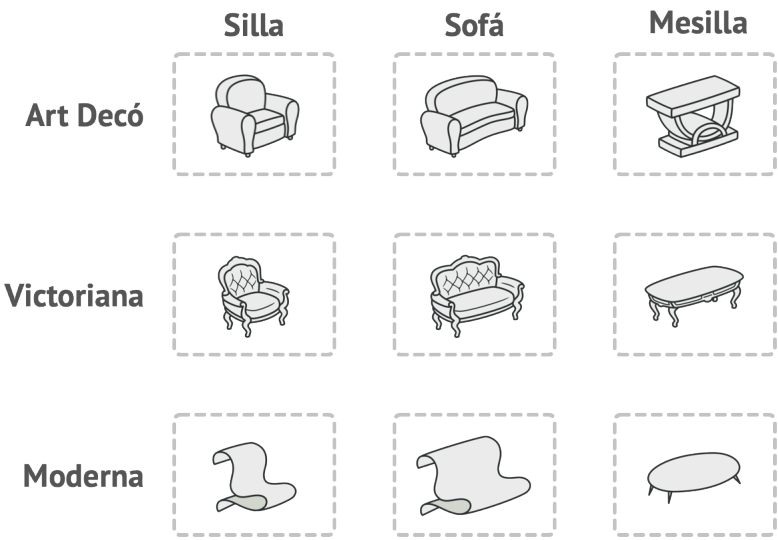
Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

☹ Problema

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

1. Una familia de productos relacionados, digamos: `Silla` + `Sofá` + `Mesilla`.

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



Familias de productos y sus variantes.

Necesitamos una forma de crear objetos individuales de mobiliario para que combinen con otros objetos de la misma fami-

lia. Los clientes se enfadan bastante cuando reciben muebles que no combinan.



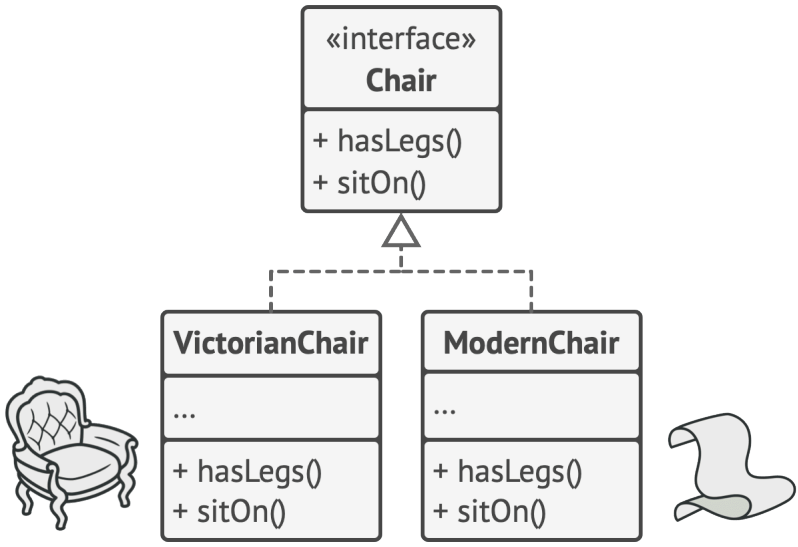
Un sofá de estilo moderno no combina con unas sillas de estilo victoriano.

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

😊 Solución

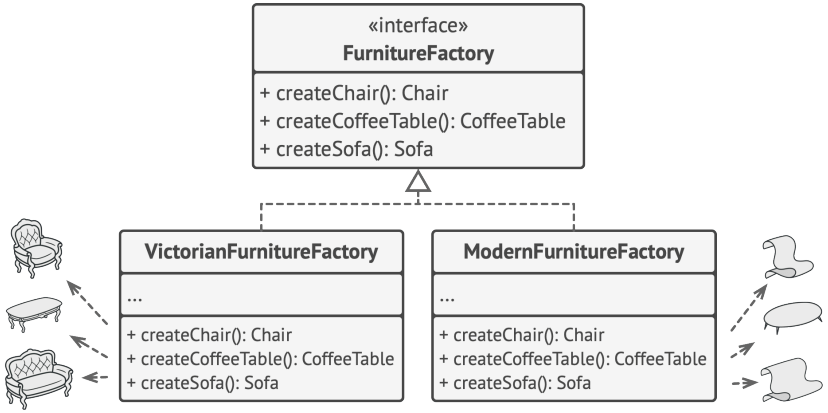
Lo primero que sugiere el patrón Abstract Factory es que declaremos de forma explícita interfaces para cada producto diferente de la familia de productos (por ejemplo, silla, sofá o mesilla). Después podemos hacer que todas las variantes de los productos sigan esas interfaces. Por ejemplo, todas las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

como todas las variantes de mesilla pueden implementar la interfaz `Mesilla`, y así sucesivamente.



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

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



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

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

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



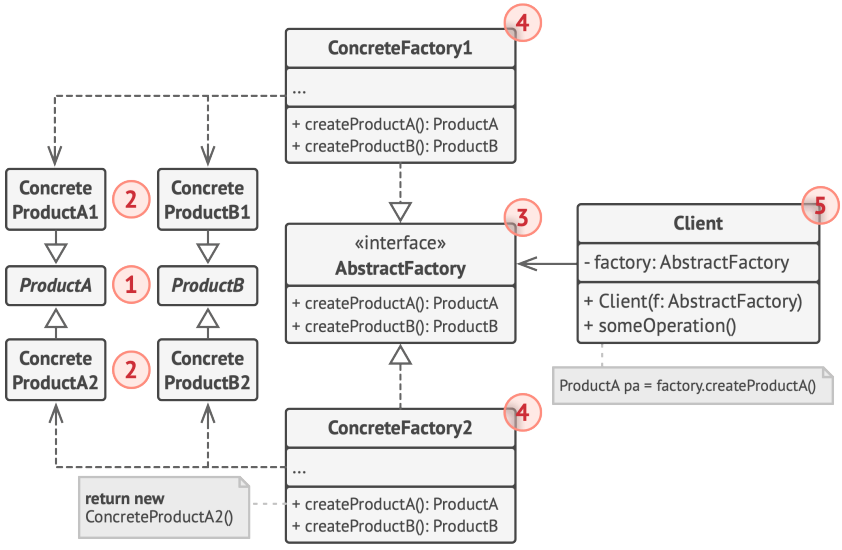
Al cliente no le debe importar la clase concreta de la fábrica con la que funciona.

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

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

www.detodoprogramacion.org

Estructura



1. Los **Productos Abstractos** declaran interfaces para un grupo de productos diferentes pero relacionados que forman una familia de productos.

2. Los **Productos Concretos** son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (silla/sofá) debe implementarse en todas las variantes dadas (victoriano/moderno).

3. La interfaz **Fábrica Abstracta** declara un grupo de métodos para crear cada uno de los productos abstractos.

4. Las **Fábricas Concretas** implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con

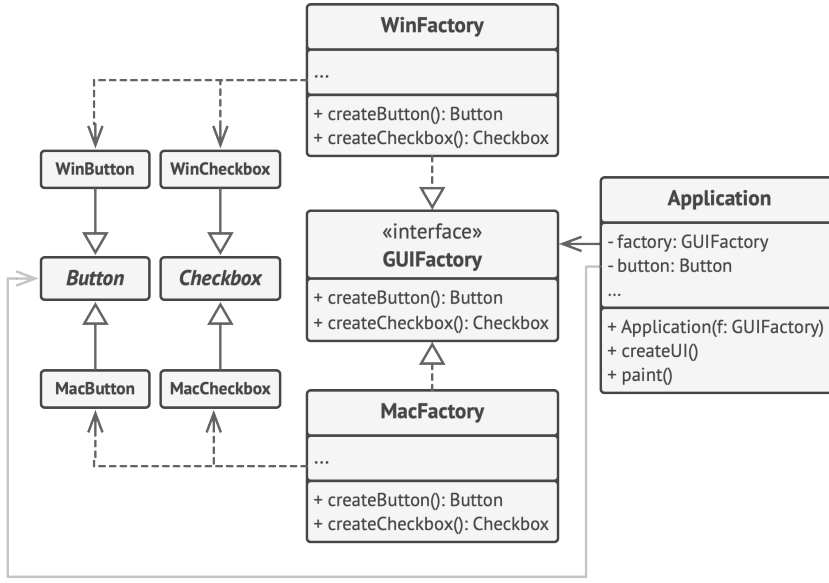
una variante específica de los productos y crea tan solo dichas variantes de los productos.

5. Aunque las fábricas concretas instancian productos concretos, las firmas de sus métodos de creación deben devolver los productos *abstractos* correspondientes. De este modo, el código cliente que utiliza una fábrica no se acopla a la variante específica del producto que obtiene de una fábrica. El **Cliente** puede funcionar con cualquier variante fábrica/producto concreta, siempre y cuando se comunique con sus objetos a través de interfaces abstractas.

Pseudocódigo

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

www.detodoprogramacion.org



Ejemplo de clases UI multiplataforma.

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

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

www.detodoprogramacion.org

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

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

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

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

```
    method createCheckbox():Checkbox


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

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


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

// Los productos concretos son creados por las fábricas
// concretas correspondientes.
```

```
class WinButton implements Button is
  method paint() is
    // Representa un botón en estilo Windows.

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

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

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

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


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

```
    this.factory = factory
  method createUI() is
    this.button = factory.createButton()
  method paint() is
    button.paint()


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

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

    Application app = new Application(factory)
```

Aplicabilidad

Utiliza el patrón Abstract Factory cuando tu código deba funcionar con varias familias de productos relacionados, pero no desees que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad.

El patrón Abstract Factory nos ofrece una interfaz para crear objetos a partir de cada clase de la familia de productos. Mientras tu código cree objetos a través de esta interfaz, no tendrás que preocuparte por crear la variante errónea de un producto que no combine con los productos que ya ha creado tu aplicación.

- Considera la implementación del patrón Abstract Factory cuando tengas una clase con un grupo de **métodos de fábrica** que nublen su responsabilidad principal.

- En un programa bien diseñado *cada clase es responsable tan solo de una cosa*. Cuando una clase lidia con varios tipos de productos, puede ser que valga la pena extraer sus métodos de fábrica para ponerlos en una clase única de fábrica o una implementación completa del patrón Abstract Factory.

Cómo implementarlo

1. Mapea una matriz de distintos tipos de productos frente a variantes de dichos productos.

2. Declara interfaces abstractas de producto para todos los tipos de productos. Después haz que todas las clases concretas de productos implementen esas interfaces.

3. Declara la interfaz de la fábrica abstracta con un grupo de métodos de creación para todos los productos abstractos.

www.detodoprogramacion.org

4. Implementa un grupo de clases concretas de fábrica, una por cada variante de producto.

5. Crea un código de inicialización de la fábrica en algún punto de la aplicación. Deberá instanciar una de las clases concretas de la fábrica, dependiendo de la configuración de la aplicación o del entorno actual. Pasa este objeto de fábrica a todas las clases que construyen productos.

6. Explora el código y encuentra todas las llamadas directas a constructores de producto. Sustitúyelas por llamadas al método de creación adecuado dentro del objeto de fábrica.

Pros y contras

✓ Puedes tener la certeza de que los productos que obtienes de una fábrica son compatibles entre sí.

✓ Evitas un acoplamiento fuerte entre productos concretos y el código cliente.

✓ *Principio de responsabilidad única*. Puedes mover el código de creación de productos a un solo lugar, haciendo que el código sea más fácil de mantener.

✓ *Principio de abierto/cerrado*. Puedes introducir nuevas variantes de productos sin descomponer el código cliente existente.

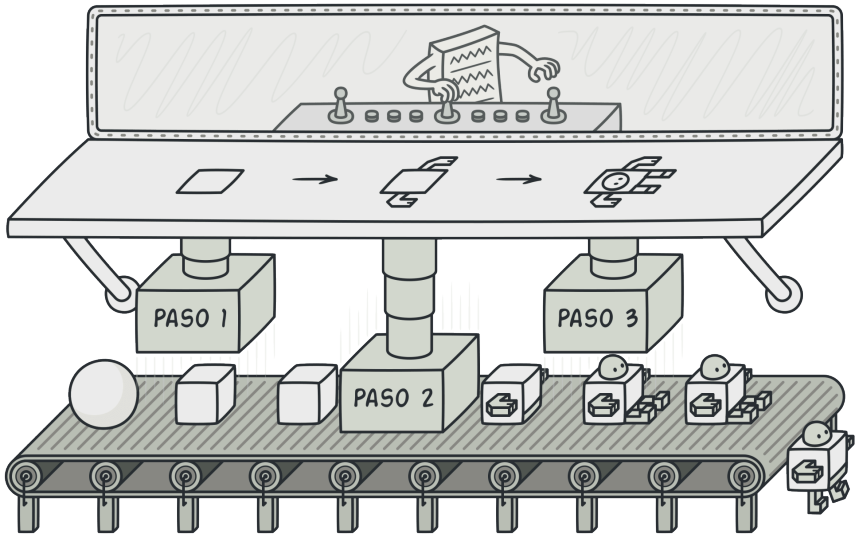
✕ Puede ser que el código se complique más de lo que debería, ya que se introducen muchas nuevas interfaces y clases junto al patrón.

www.detodoprogramacion.org

⇄ Relaciones con otros patrones

- Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).
- **Builder** se enfoca en construir objetos complejos, paso a paso. **Abstract Factory** se especializa en crear familias de objetos relacionados. *Abstract Factory* devuelve el producto inmediatamente, mientras que *Builder* te permite ejecutar algunos pasos adicionales de construcción antes de extraer el producto.
- Las clases del **Abstract Factory** a menudo se basan en un grupo de **métodos de fábrica**, pero también puedes utilizar **Prototype** para escribir los métodos de estas clases.
- **Abstract Factory** puede servir como alternativa a **Facade** cuando tan solo deseas esconder la forma en que se crean los objetos del subsistema a partir del código cliente.
- Puedes utilizar **Abstract Factory** junto a **Bridge**. Este emparejamiento resulta útil cuando algunas abstracciones definidas por *Bridge* sólo pueden funcionar con implementaciones específicas. En este caso, *Abstract Factory* puede encapsular estas relaciones y esconder la complejidad al código cliente.
- Los patrones **Abstract Factory**, **Builder** y **Prototype** pueden todos ellos implementarse como **Singletons**.

www.detodoprogramacion.org



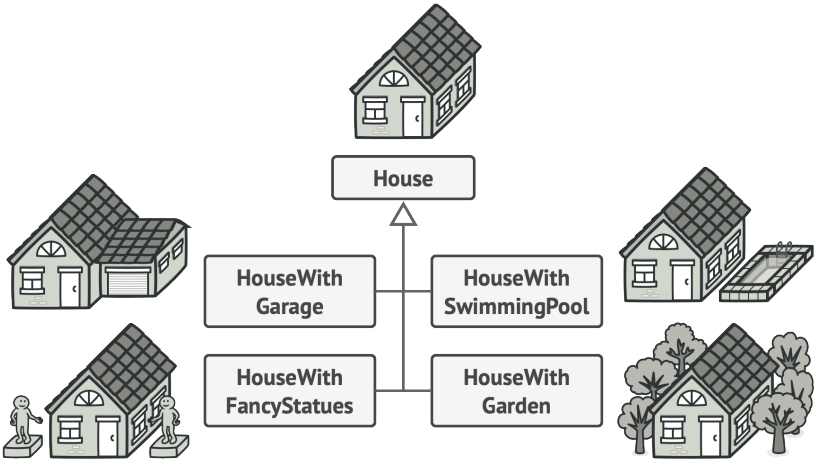
BUILDER

También llamado: Constructor

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

☹ Problema

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

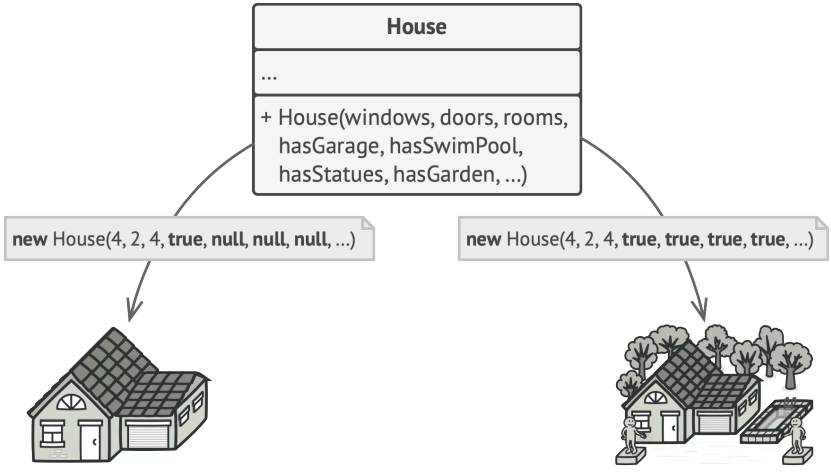


Crear una subclase por cada configuración posible de un objeto puede complicar demasiado el programa.

Por ejemplo, pensemos en cómo crear un objeto `Casa`. Para construir una casa sencilla, debemos construir cuatro paredes y un piso, así como instalar una puerta, colocar un par de ventanas y ponerle un tejado. Pero ¿qué pasa si quieres una casa más grande y luminosa, con un jardín y otros extras (como sistema de calefacción, instalación de fontanería y cableado eléctrico)?

La solución más sencilla es extender la clase base `Casa` y crear un grupo de subclases que cubran todas las combinaciones posibles de los parámetros. Pero, en cualquier caso, acabarás con una cantidad considerable de subclases. Cualquier parámetro nuevo, como el estilo del porche, exigirá que incrementes esta jerarquía aún más.

Existe otra posibilidad que no implica generar subclases. Puedes crear un enorme constructor dentro de la clase base `Casa` con todos los parámetros posibles para controlar el objeto casa. Aunque es cierto que esta solución elimina la necesidad de las subclases, genera otro problema.



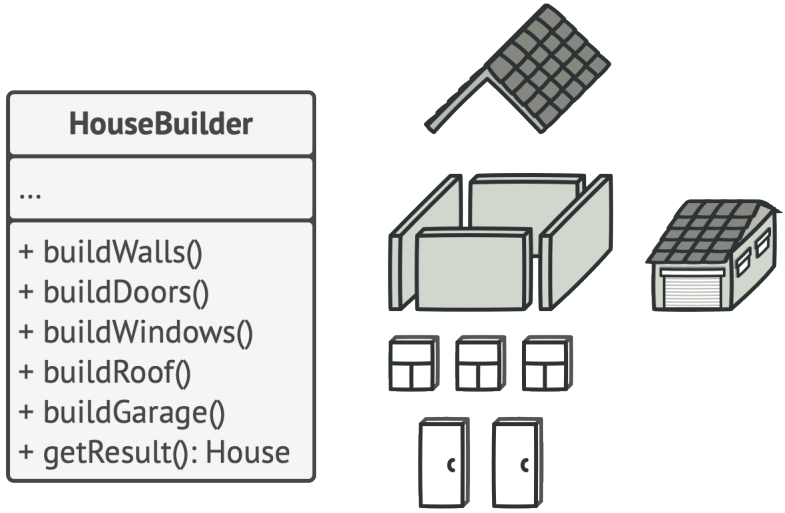
Un constructor con un montón de parámetros tiene su inconveniente: no todos los parámetros son necesarios todo el tiempo.

En la mayoría de los casos, gran parte de los parámetros no se utilizará, lo que provocará que **las llamadas al constructor sean bastante feas**. Por ejemplo, solo una pequeña parte de las

casas tiene piscina, por lo que los parámetros relacionados con piscinas serán inútiles en nueve de cada diez casos.

😊 Solución

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



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

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

pasos. Puedes invocar sólo aquellos que sean necesarios para producir una configuración particular de un objeto.

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

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



Los distintos constructores ejecutan la misma tarea de formas distintas.

Por ejemplo, imagina un constructor que construye todo de madera y vidrio, otro que construye todo con piedra y hierro

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

Clase directora

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



La clase directora sabe qué pasos de construcción ejecutar para lograr un producto que funcione.

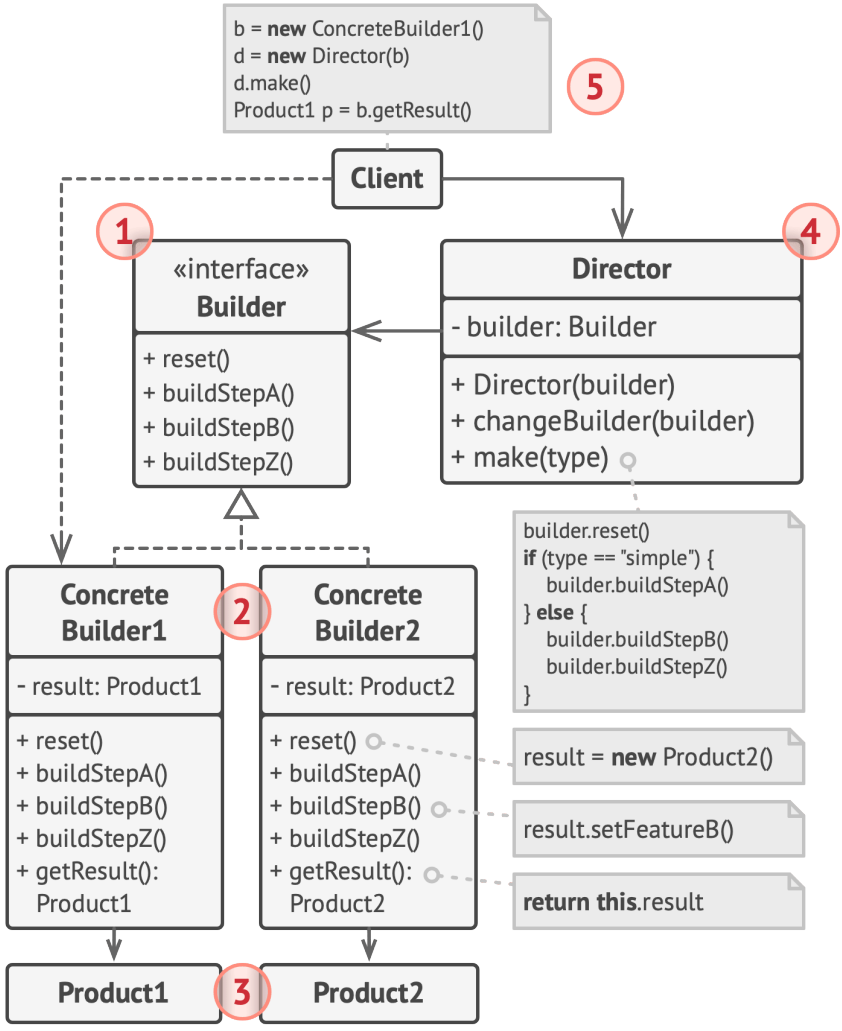
www.detodoprogramacion.org

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

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

www.detodoprogramacion.org

Estructura



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

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

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

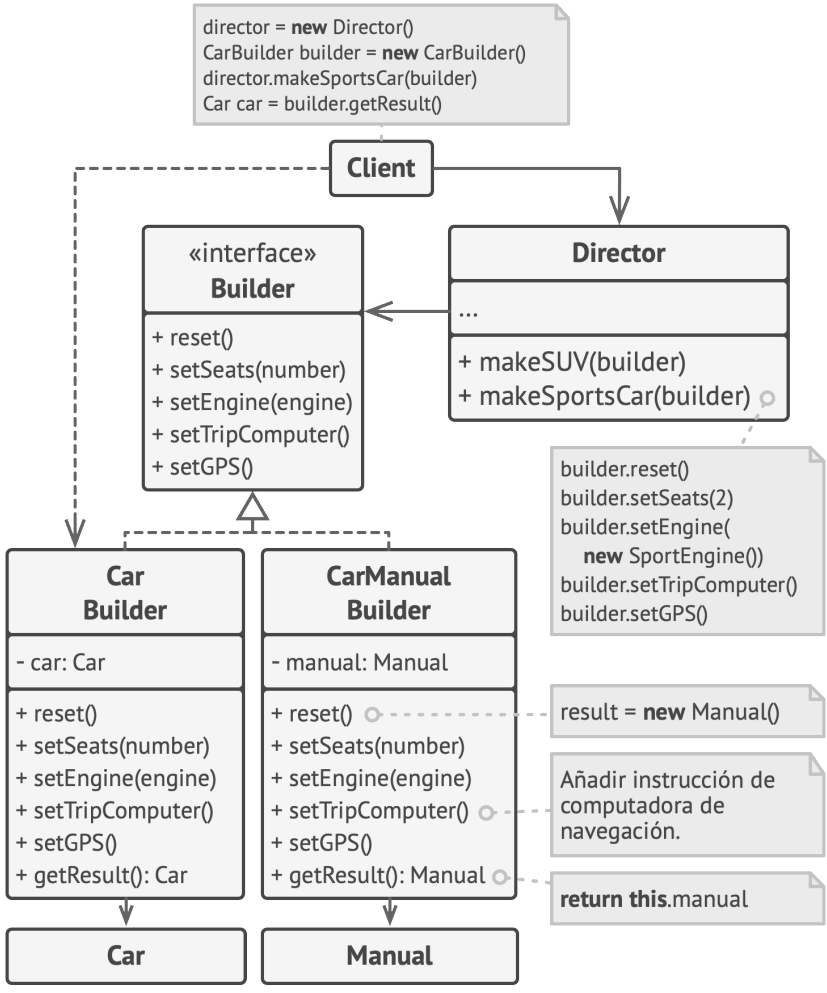
4. La clase **Directora** define el orden en el que se invocarán los pasos de construcción, por lo que puedes crear y reutilizar configuraciones específicas de los productos.

5. El **Cliente** debe asociar uno de los objetos constructores con la clase directora. Normalmente, se hace una sola vez mediante los parámetros del constructor de la clase directora, que utiliza el objeto constructor para el resto de la construcción. No obstante, existe una solución alternativa para cuando el cliente pasa el objeto constructor al método de producción de la clase directora. En este caso, puedes utilizar un constructor diferente cada vez que produzcas algo con la clase directora.

Pseudocódigo

Este ejemplo del patrón **Builder** ilustra cómo se puede reutilizar el mismo código de construcción de objetos a la hora de construir distintos tipos de productos, como automóviles, y crear los correspondientes manuales para esos automóviles.

www.detodoprogramacion.org



Ejemplo de una construcción paso a paso de automóviles y de los manuales de usuario para esos modelos de automóvil.

Un automóvil es un objeto complejo que puede construirse de mil maneras diferentes. En lugar de saturar la clase `Automóvil` con un constructor enorme, extrajimos el código de ensamblaje del automóvil y lo pusimos en una clase constructora de automóviles independiente. Esta clase tiene un

www.detodoprogramacion.org

grupo de métodos para configurar las distintas partes de un automóvil.

Si el código cliente necesita ensamblar un modelo de automóvil con ajustes especiales, puede trabajar directamente con el objeto constructor. Por otro lado, el cliente puede delegar el ensamblaje a la clase directora, que sabe cómo utilizar un objeto constructor para construir varios de los modelos más populares de automóviles.

Puede que te sorprenda, pero todo automóvil necesita un manual (en serio, ¿quién se los lee?). El manual explica cada característica del automóvil, de modo que los detalles del manual varían de un modelo a otro. Por eso tiene lógica reutilizar un proceso de construcción existente para automóviles reales y sus respectivos manuales. Por supuesto, elaborar un manual no es lo mismo que fabricar un automóvil, por lo que debemos incluir otra clase constructora especializada en elaborar manuales. Esta clase implementa los mismos métodos de construcción que su hermana constructora de automóviles , pero, en lugar de fabricar piezas del automóvil, las describe. Al pasar estos constructores al mismo objeto director, podemos construir tanto un automóvil como un manual.

La última parte consiste en buscar el objeto resultante. Un automóvil de metal y un manual de papel, aunque estén relacionados, son objetos muy diferentes. No podemos colocar un método para buscar resultados en la clase directora sin acoplarla a clases de productos concretos. Por lo tanto, obtenemos

www.detodoprogramacion.org

el resultado de la construcción del constructor que realizó el trabajo.

```
// El uso del patrón Builder sólo tiene sentido cuando tus
// productos son bastante complejos y requieren una
// configuración extensiva. Los dos siguientes productos están
// relacionados, aunque no tienen una interfaz común.
class Car is
  // Un coche puede tener un GPS, una computadora de
  // navegación y cierto número de asientos. Los distintos
  // modelos de coches (deportivo, SUV, descapotable) pueden
  // tener distintas características instaladas o habilitadas.

class Manual is
  // Cada coche debe contar con un manual de usuario que se
  // corresponda con la configuración del coche y explique
  // todas sus características.


// La interfaz constructora especifica métodos para crear las
// distintas partes de los objetos del producto.
interface Builder is
  method reset()
  method setSeats(...)
  method setEngine(...)
  method setTripComputer(...)
  method setGPS(...)

// Las clases constructoras concretas siguen la interfaz
// constructora y proporcionan implementaciones específicas de
// los pasos de construcción. Tu programa puede tener multitud
```

```
// de variaciones de objetos constructores, cada una de ellas
// implementada de forma diferente.
class CarBuilder implements Builder is
  private field car:Car

  // Una nueva instancia de la clase constructora debe
  // contener un objeto de producto en blanco que utiliza en
  // el montaje posterior.
  constructor CarBuilder() is
    this.reset()

  // El método reset despeja el objeto en construcción.
  method reset() is
    this.car = new Car()

  // Todos los pasos de producción funcionan con la misma
  // instancia de producto.
  method setSeats(...) is
    // Establece la cantidad de asientos del coche.

  method setEngine(...) is
    // Instala un motor específico.

  method setTripComputer(...) is
    // Instala una computadora de navegación.

  method setGPS(...) is
    // Instala un GPS.

  // Los constructores concretos deben proporcionar sus
  // propios métodos para obtener resultados. Esto se debe a
  // que varios tipos de objetos constructores pueden crear
```

```
    // productos completamente diferentes de los cuales no todos
    // siguen la misma interfaz. Por lo tanto, dichos métodos no
    // pueden declararse en la interfaz constructora (al menos
    // no en un lenguaje de programación de tipado estático).
    //
    // Normalmente, tras devolver el resultado final al cliente,
    // una instancia constructora debe estar lista para empezar
    // a generar otro producto. Ese es el motivo por el que es
    // práctica común invocar el método reset al final del
    // cuerpo del método `getProduct`. Sin embargo, este
    // comportamiento no es obligatorio y puedes hacer que tu
    // objeto constructor espere una llamada reset explícita del
    // código cliente antes de desechar el resultado anterior.
    method getProduct():Car is
      product = this.car
      this.reset()
      return product

// Al contrario que otros patrones creacionales, Builder te
// permite construir productos que no siguen una interfaz común.
class CarManualBuilder implements Builder is
  private field manual:Manual

  constructor CarManualBuilder() is
    this.reset()

  method reset() is
    this.manual = new Manual()

  method setSeats(...) is
    // Documenta las características del asiento del coche.

```

```
    method setEngine(...) is
      // Añade instrucciones del motor.

    method setTripComputer(...) is
      // Añade instrucciones de la computadora de navegación.

    method setGPS(...) is
      // Añade instrucciones del GPS.

    method getProduct():Manual is
      // Devuelve el manual y rearma el constructor.


// El director sólo es responsable de ejecutar los pasos de
// construcción en una secuencia particular. Resulta útil cuando
// se crean productos de acuerdo con un orden o configuración
// específicos. En sentido estricto, la clase directora es
// opcional, ya que el cliente puede controlar directamente los
// objetos constructores.
class Director is
  private field builder:Builder

  // El director funciona con cualquier instancia de
  // constructor que le pase el código cliente. De esta forma,
  // el código cliente puede alterar el tipo final del
  // producto recién montado.
  method setBuilder(builder:Builder)
    this.builder = builder

  // El director puede construir multitud de variaciones de
  // producto utilizando los mismos pasos de construcción.
  method constructSportsCar(builder: Builder) is
```

```
    builder.reset()
    builder.setSeats(2)
    builder.setEngine(new SportEngine())
    builder.setTripComputer(true)
    builder.setGPS(true)

  method constructSUV(builder: Builder) is
    // ...


// El código cliente crea un objeto constructor, lo pasa al
// director y después inicia el proceso de construcción. El
// resultado final se extrae del objeto constructor.
class Application is

  method makeCar() is
    director = new Director()

    CarBuilder builder = new CarBuilder()
    director.constructSportsCar(builder)
    Car car = builder.getProduct()

    CarManualBuilder builder = new CarManualBuilder()
    director.constructSportsCar(builder)

    // El producto final a menudo se extrae de un objeto
    // constructor, ya que el director no conoce y no
    // depende de constructores y productos concretos.
    Manual manual = builder.getProduct()
```

Aplicabilidad

Utiliza el patrón Builder para evitar un "constructor telescópico".

Digamos que tenemos un constructor con diez parámetros opcionales. Invocar a semejante bestia es poco práctico, por lo que sobrecargamos el constructor y creamos varias versiones más cortas con menos parámetros. Estos constructores siguen recurriendo al principal, pasando algunos valores por defecto a cualquier parámetro omitido.

```
class Pizza {
  Pizza(int size) { ... }
  Pizza(int size, boolean cheese) { ... }
  Pizza(int size, boolean cheese, boolean pepperoni) { ... }
  // ...
```

Crear un monstruo semejante sólo es posible en lenguajes que soportan la sobrecarga de métodos, como C# o Java.

El patrón Builder permite construir objetos paso a paso, utilizando tan solo aquellos pasos que realmente necesitamos. Una vez implementado el patrón, ya no hará falta apiñar decenas de parámetros dentro de los constructores.

Utiliza el patrón Builder cuando quieras que el código sea capaz de crear distintas representaciones de ciertos productos (por ejemplo, casas de piedra y madera).

El patrón Builder se puede aplicar cuando la construcción de varias representaciones de un producto requiera de pasos similares que sólo varían en los detalles.

La interfaz constructora base define todos los pasos de construcción posibles, mientras que los constructores concretos implementan estos pasos para construir representaciones particulares del producto. Entre tanto, la clase directora guía el orden de la construcción.

Utiliza el patrón Builder para construir árboles con el patrón Composite u otros objetos complejos.

El patrón Builder te permite construir productos paso a paso. Podrás aplazar la ejecución de ciertos pasos sin descomponer el producto final. Puedes incluso invocar pasos de forma recursiva, lo cual resulta útil cuando necesitas construir un árbol de objetos.

Un constructor no expone el producto incompleto mientras ejecuta los pasos de construcción. Esto evita que el código cliente extraiga un resultado incompleto.

Cómo implementarlo

1. Asegúrate de poder definir claramente los pasos comunes de construcción para todas las representaciones disponibles del producto. De lo contrario, no podrás proceder a implementar el patrón.

www.detodoprogramacion.org

2. Declara estos pasos en la interfaz constructora base.

3. Crea una clase constructora concreta para cada una de las representaciones de producto e implementa sus pasos de construcción.

   No olvides implementar un método para extraer el resultado de la construcción. La razón por la que este método no se puede declarar dentro de la interfaz constructora es que varios constructores pueden construir productos sin una interfaz común. Por lo tanto, no sabemos cuál será el tipo de retorno para un método como éste. No obstante, si trabajas con productos de una única jerarquía, el método de extracción puede añadirse sin problemas a la interfaz base.

4. Piensa en crear una clase directora. Puede encapsular varias formas de construir un producto utilizando el mismo objeto constructor.

5. El código cliente crea tanto el objeto constructor como el director. Antes de que empiece la construcción, el cliente debe pasar un objeto constructor al director. Normalmente, el cliente hace esto sólo una vez, mediante los parámetros del constructor del director. El director utiliza el objeto constructor para el resto de la construcción. Existe una manera alternativa, en la que el objeto constructor se pasa directamente al método de construcción del director.

www.detodoprogramacion.org

6. El resultado de la construcción tan solo se puede obtener directamente del director si todos los productos siguen la misma interfaz. De lo contrario, el cliente deberá extraer el resultado del constructor.

Pros y contras

✓ Puedes construir objetos paso a paso, aplazar pasos de la construcción o ejecutar pasos de forma recursiva.

✓ Puedes reutilizar el mismo código de construcción al construir varias representaciones de productos.

✓ *Principio de responsabilidad única*. Puedes aislar un código de construcción complejo de la lógica de negocio del producto.

✕ La complejidad general del código aumenta, ya que el patrón exige la creación de varias clases nuevas.

Relaciones con otros patrones

- Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).

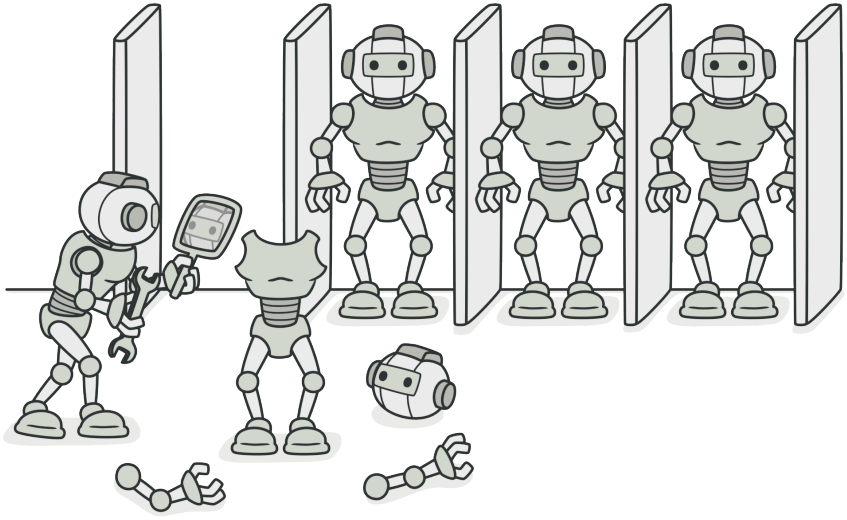
- **Builder** se enfoca en construir objetos complejos, paso a paso. **Abstract Factory** se especializa en crear familias de objetos relacionados. *Abstract Factory* devuelve el producto inmediatamente, mientras que *Builder* te permite ejecutar algunos pasos adicionales de construcción antes de extraer el producto.

www.detodoprogramacion.org

- Puedes utilizar **Builder** al crear árboles **Composite** complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.

- Puedes combinar **Builder** con **Bridge**: la clase *directora* juega el papel de la abstracción, mientras que diferentes *constructoras* actúan como *implementaciones*.

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



PROTOTYPE

También llamado: Prototipo, Clon, Clone

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

☹ Problema

Digamos que tienes un objeto y quieres crear una copia exacta de él. ¿Cómo lo harías? En primer lugar, debes crear un nuevo objeto de la misma clase. Después debes recorrer todos los campos del objeto original y copiar sus valores en el nuevo objeto.

¡Bien! Pero hay una trampa. No todos los objetos se pueden copiar de este modo, porque algunos de los campos del objeto pueden ser privados e invisibles desde fuera del propio objeto.



No siempre *es posible copiar un objeto "desde fuera".*

Hay otro problema con el enfoque directo. Dado que debes conocer la clase del objeto para crear un duplicado, el código se vuelve dependiente de esa clase. Si esta dependencia adicional no te da miedo, todavía hay otra trampa. En ocasiones tan solo conocemos la interfaz que sigue el objeto, pero no su

www.detodoprogramacion.org

clase concreta, cuando, por ejemplo, un parámetro de un método acepta cualquier objeto que siga cierta interfaz.

😊 Solución

El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método `clonar`.

La implementación del método `clonar` es muy parecida en todas las clases. El método crea un objeto a partir de la clase actual y lleva todos los valores de campo del viejo objeto, al nuevo. Se puede incluso copiar campos privados, porque la mayoría de los lenguajes de programación permite a los objetos acceder a campos privados de otros objetos que pertenecen a la misma clase.

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

www.detodoprogramacion.org

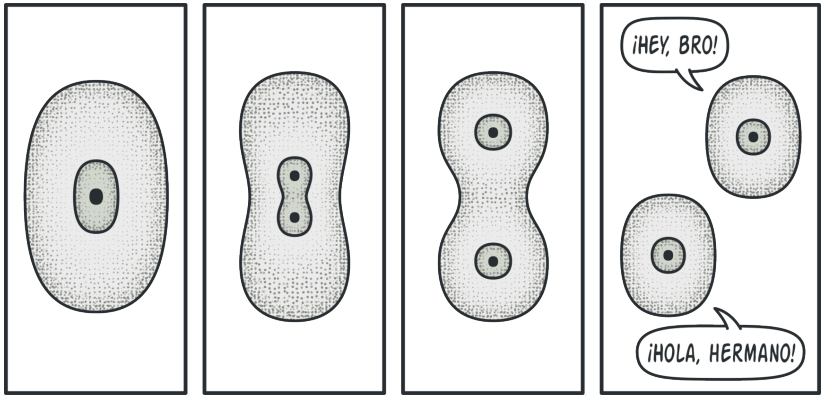


Los prototipos prefabricados pueden ser una alternativa a las subclases.

Funciona así: se crea un grupo de objetos configurados de maneras diferentes. Cuando necesites un objeto como el que has configurado, clonas un prototipo en lugar de construir un nuevo objeto desde cero.

Analogía del mundo real

En la vida real, los prototipos se utilizan para realizar pruebas de todo tipo antes de comenzar con la producción en masa de un producto. Sin embargo, en este caso, los prototipos no forman parte de una producción real, sino que juegan un papel pasivo.

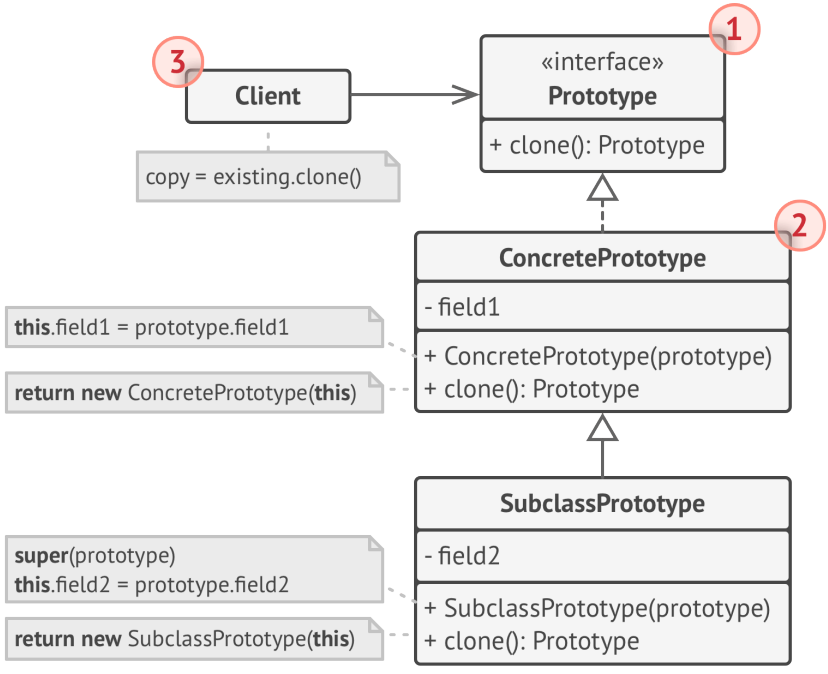


La división de una célula.

Ya que los prototipos industriales en realidad no se copian a sí mismos, una analogía más precisa del patrón es el proceso de la división mitótica de una célula (biología, ¿recuerdas?). Tras la división mitótica, se forma un par de células idénticas. La célula original actúa como prototipo y asume un papel activo en la creación de la copia.

Estructura

Implementación básica



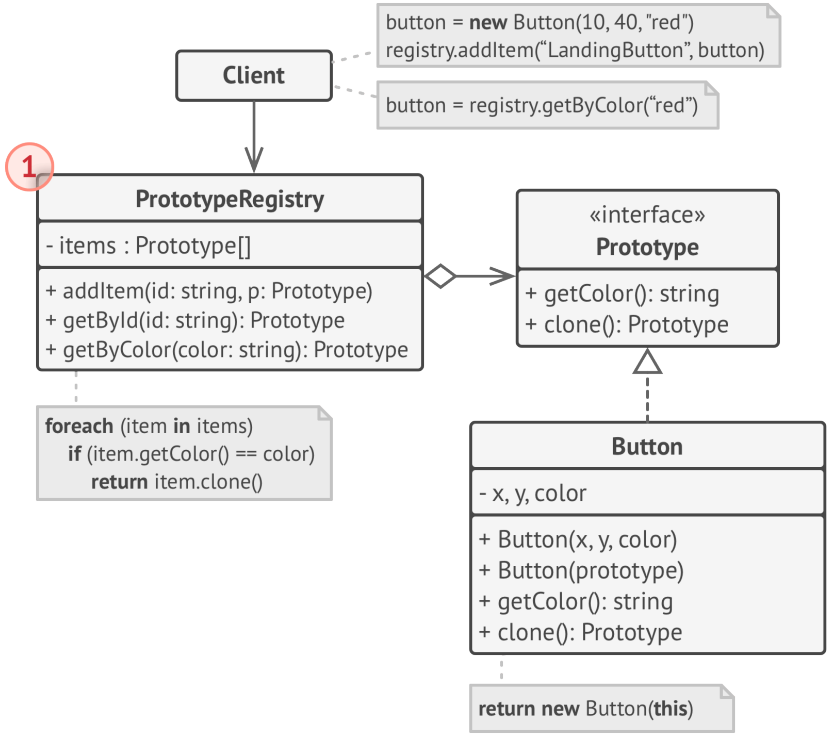
1. La interfaz **Prototipo** declara los métodos de clonación. En la mayoría de los casos, se trata de un único método `clonar`.

2. La clase **Prototipo Concreto** implementa el método de clonación. Además de copiar la información del objeto original al clon, este método también puede gestionar algunos casos extremos del proceso de clonación, como, por ejemplo, clonar objetos vinculados, deshacer dependencias recursivas, etc.

3. El **Cliente** puede producir una copia de cualquier objeto que siga la interfaz del prototipo.

Descargado de: www.detodoprogramacion.org

Implementación del registro de prototipos

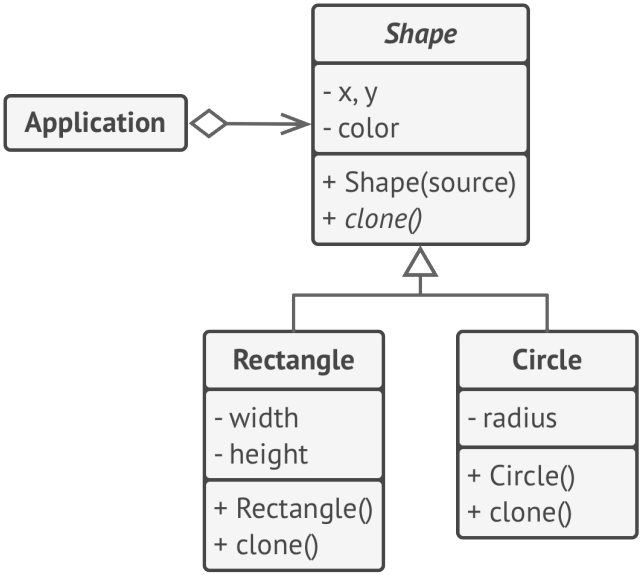


1. El **Registro de Prototipos** ofrece una forma sencilla de acceder a prototipos de uso frecuente. Almacena un grupo de objetos prefabricados listos para ser copiados. El registro de prototipos más sencillo es una tabla *hash* con los pares `name → prototype`. No obstante, si necesitas un criterio de búsqueda más preciso que un simple nombre, puedes crear una versión mucho más robusta del registro.

www.detodoprogramacion.org

Pseudocódigo

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



Clonación de un grupo de objetos que pertenece a una jerarquía de clase.

Todas las clases de forma siguen la misma interfaz, que proporciona un método de clonación. Una subclase puede invocar el método de clonación padre antes de copiar sus propios valores de campo al objeto resultante.

```
1  // Prototipo base.
2  abstract class Shape is
3    field X: int
```

```
    field Y: int
    field color: string

    // Un constructor normal.
    constructor Shape() is
      // ...

    // El constructor prototipo. Un nuevo objeto se inicializa
    // con valores del objeto existente.
    constructor Shape(source: Shape) is
      this()
      this.X = source.X
      this.Y = source.Y
      this.color = source.color

    // La operación clonar devuelve una de las subclases de
    // Shape (Forma).
    abstract method clone():Shape


// Prototipo concreto. El método de clonación crea un nuevo
// objeto y lo pasa al constructor. Hasta que el constructor
// termina, tiene una referencia a un nuevo clon. De este modo
// nadie tiene acceso a un clon a medio terminar. Esto garantiza
// la consistencia del resultado de la clonación.
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
      // Para copiar campos privados definidos en la clase
      // padre es necesaria una llamada a un constructor
```

```
36      // padre.
37      super(source)
38      this.width = source.width
39      this.height = source.height
40
41    method clone():Shape is
42      return new Rectangle(this)
43
44
45  class Circle extends Shape is
46    field radius: int
47
48    constructor Circle(source: Circle) is
49      super(source)
50      this.radius = source.radius
51
52    method clone():Shape is
53      return new Circle(this)
54
55
56  // En alguna parte del código cliente.
57  class Application is
58    field shapes: array of Shape
59
60    constructor Application() is
61      Circle circle = new Circle()
62      circle.X = 10
63      circle.Y = 10
64      circle.radius = 20
65      shapes.add(circle)
66
67      Circle anotherCircle = circle.clone()
```

```
    shapes.add(anotherCircle)
    // La variable `anotherCircle` (otroCírculo) contiene
    // una copia exacta del objeto `circle`.

    Rectangle rectangle = new Rectangle()
    rectangle.width = 10
    rectangle.height = 20
    shapes.add(rectangle)

  method businessLogic() is
    // Prototype es genial porque te permite producir una
    // copia de un objeto sin conocer nada de su tipo.
    Array shapesCopy = new Array of Shapes.

    // Por ejemplo, no conocemos los elementos exactos de la
    // matriz de formas. Lo único que sabemos es que son
    // todas formas. Pero, gracias al polimorfismo, cuando
    // invocamos el método `clonar` en una forma, el
    // programa comprueba su clase real y ejecuta el método
    // de clonación adecuado definido en dicha clase. Por
    // eso obtenemos los clones adecuados en lugar de un
    // grupo de simples objetos Shape.
    foreach (s in shapes) do
      shapesCopy.add(s.clone())

    // La matriz `shapesCopy` contiene copias exactas del
    // hijo de la matriz `shape`.
```

Aplicabilidad

Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.

Esto sucede a menudo cuando tu código funciona con objetos pasados por código de terceras personas a través de una interfaz. Las clases concretas de estos objetos son desconocidas y no podrías depender de ellas aunque quisieras.

El patrón Prototype proporciona al código cliente una interfaz general para trabajar con todos los objetos que soportan la clonación. Esta interfaz hace que el código cliente sea independiente de las clases concretas de los objetos que clona.

Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.

El patrón Prototype te permite utilizar como prototipos un grupo de objetos prefabricados, configurados de maneras diferentes.

En lugar de instanciar una subclase que coincida con una configuración, el cliente puede, sencillamente, buscar el prototipo adecuado y clonarlo.

Cómo implementarlo

1. Crea la interfaz del prototipo y declara el método `clonar` en ella, o, simplemente, añade el método a todas las clases de una jerarquía de clase existente, si la tienes.

2. Una clase de prototipo debe definir el constructor alternativo que acepta un objeto de dicha clase como argumento. El constructor debe copiar los valores de todos los campos definidos en la clase del objeto que se le pasa a la instancia recién creada. Si deseas cambiar una subclase, debes invocar al constructor padre para permitir que la superclase gestione la clonación de sus campos privados.

   Si el lenguaje de programación que utilizas no soporta la sobrecarga de métodos, puedes definir un método especial para copiar la información del objeto. El constructor es el lugar más adecuado para hacerlo, porque entrega el objeto resultante justo después de invocar el operador `new`.

3. Normalmente, el método de clonación consiste en una sola línea que ejecuta un operador `new` con la versión prototípica del constructor. Observa que todas las clases deben sobreescribir explícitamente el método de clonación y utilizar su propio nombre de clase junto al operador `new`. De lo contrario, el método de clonación puede producir un objeto a partir de una clase madre.

www.detodoprogramacion.org

4. Opcionalmente, puedes crear un registro de prototipos centralizado para almacenar un catálogo de prototipos de uso frecuente.

   Puedes implementar el registro como una nueva clase de fábrica o colocarlo en la clase base de prototipo con un método estático para buscar el prototipo. Este método debe buscar un prototipo con base en el criterio de búsqueda que el código cliente pase al método. El criterio puede ser una etiqueta tipo *string* o un grupo complejo de parámetros de búsqueda. Una vez encontrado el prototipo adecuado, el registro deberá clonarlo y devolver la copia al cliente.

   Por último, sustituye las llamadas directas a los constructores de las subclases por llamadas al método de fábrica del registro de prototipos.

⚖ Pros y contras

✓ Puedes clonar objetos sin acoplarlos a sus clases concretas.

✓ Puedes evitar un código de inicialización repetido clonando prototipos prefabricados.

✓ Puedes crear objetos complejos con más facilidad.

✓ Obtienes una alternativa a la herencia al tratar con preajustes de configuración para objetos complejos.

✕ Clonar objetos complejos con referencias circulares puede resultar complicado.

www.detodoprogramacion.org

⇄ Relaciones con otros patrones

- Muchos diseños empiezan utilizando el **Factory Method** (menos complicado y más personalizable mediante las subclases) y evolucionan hacia **Abstract Factory**, **Prototype**, o **Builder** (más flexibles, pero más complicados).

- Las clases del **Abstract Factory** a menudo se basan en un grupo de **métodos de fábrica**, pero también puedes utilizar **Prototype** para escribir los métodos de estas clases.

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

- Los diseños que hacen un uso amplio de **Composite** y **Decorator** a menudo pueden beneficiarse del uso del **Prototype**. Aplicar el patrón te permite clonar estructuras complejas en lugar de reconstruirlas desde cero.

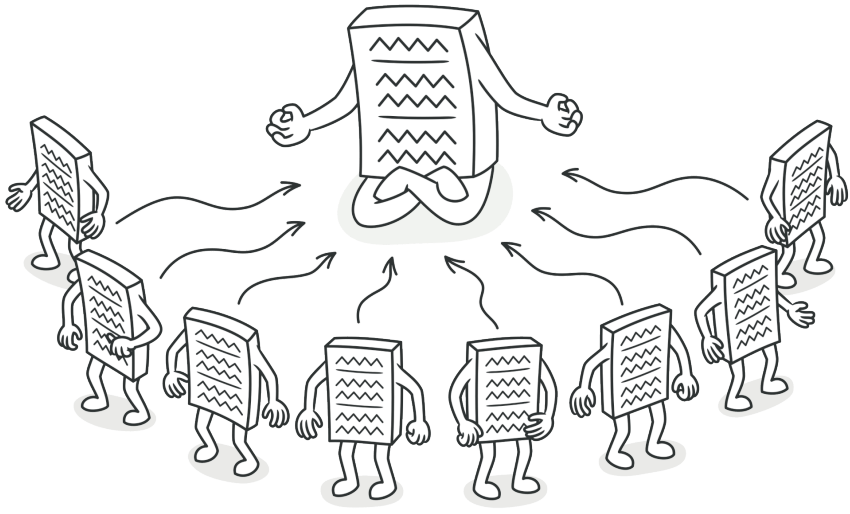
- **Prototype** no se basa en la herencia, por lo que no presenta sus inconvenientes. No obstante, *Prototype* requiere de una inicialización complicada del objeto clonado. **Factory Method** se basa en la herencia, pero no requiere de un paso de inicialización.

- En ocasiones, **Prototype** puede ser una alternativa más simple al patrón **Memento**. Esto funciona si el objeto cuyo estado quieres almacenar en el historial es suficientemente sencillo

www.detodoprogramacion.org

y no tiene enlaces a recursos externos, o estos son fáciles de restablecer.

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



SINGLETON

También llamado: Instancia única

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

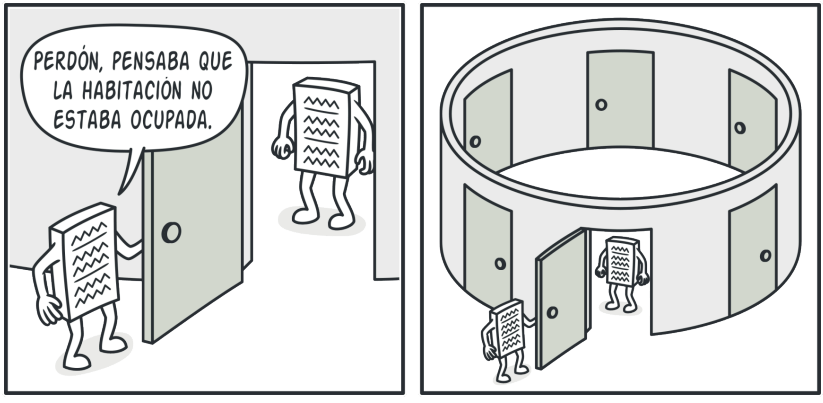
☹ Problema

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

1. **Garantizar que una clase tenga una única instancia**. ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más habitual es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.

   Funciona así: imagina que has creado un objeto y al cabo de un tiempo decides crear otro nuevo. En lugar de recibir un objeto nuevo, obtendrás el que ya habías creado.

   Ten en cuenta que este comportamiento es imposible de implementar con un constructor normal, ya que una llamada al constructor siempre **debe** devolver un nuevo objeto por diseño.



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

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

   Al igual que una variable global, el patrón Singleton nos permite acceder a un objeto desde cualquier parte del programa. No obstante, también evita que otro código sobreescriba esa instancia.

   Este problema tiene otra cara: no queremos que el código que resuelve el primer problema se encuentre disperso por todo el programa. Es mucho más conveniente tenerlo dentro de una clase, sobre todo si el resto del código ya depende de ella.

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

😊 Solución

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

- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador `new` con la clase Singleton.

- Crear un método de creación estático que actúe como constructor. Tras bambalinas, este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

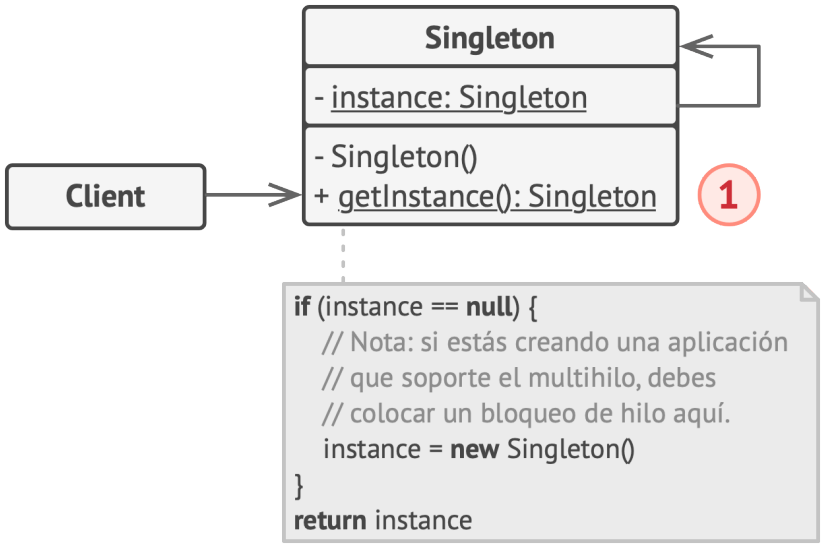
  Si tu código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

Analogía en el mundo real

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

www.detodoprogramacion.org

Estructura



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

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

Pseudocódigo

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

el primer objeto creado y lo devuelve en todas las llamadas siguientes.

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

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

  // El método estático que controla el acceso a la instancia
  // singleton.
  public static method getInstance() is
    if (Database.instance == null) then
      acquireThreadLock() and then
        // Garantiza que la instancia aún no se ha
        // inicializado por otro hilo mientras ésta ha
        // estado esperando el desbloqueo.
        if (Database.instance == null) then
          Database.instance = new Database()
    return Database.instance

```

```
  // Por último, cualquier singleton debe definir cierta
  // lógica de negocio que pueda ejecutarse en su instancia.
  public method query(sql) is
    // Por ejemplo, todas las consultas a la base de datos
    // de una aplicación pasan por este método. Por lo
    // tanto, aquí puedes colocar lógica de regularización
    // (throttling) o de envío a la memoria caché.
    // ...

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

Aplicabilidad

Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.

El patrón Singleton deshabilita el resto de las maneras de crear objetos de una clase, excepto el método especial de creación. Este método crea un nuevo objeto, o bien devuelve uno existente si ya ha sido creado.

Utiliza el patrón Singleton cuando necesites un control más estricto de las variables globales.

Al contrario que las variables globales, el patrón Singleton garantiza que haya una única instancia de una clase. A excepción de la propia clase Singleton, nada puede sustituir la instancia en caché.

Ten en cuenta que siempre podrás ajustar esta limitación y permitir la creación de cierto número de instancias Singleton. La única parte del código que requiere cambios es el cuerpo del método `getInstance`.

Cómo implementarlo

1. Añade un campo estático privado a la clase para almacenar la instancia Singleton.

2. Declara un método de creación estático público para obtener la instancia Singleton.

3. Implementa una inicialización diferida dentro del método estático. Debe crear un nuevo objeto en su primera llamada y colocarlo dentro del campo estático. El método deberá devolver siempre esa instancia en todas las llamadas siguientes.

4. Declara el constructor de clase como privado. El método estático de la clase seguirá siendo capaz de invocar al constructor, pero no a los otros objetos.

www.detodoprogramacion.org

5. Repasa el código cliente y sustituye todas las llamadas directas al constructor de la instancia Singleton por llamadas a su método de creación estático.

Pros y contras

✓ Puedes tener la certeza de que una clase tiene una única instancia.

✓ Obtienes un punto de acceso global a dicha instancia.

✓ El objeto Singleton solo se inicializa cuando se requiere por primera vez.

✗ Vulnera el *Principio de responsabilidad única*. El patrón resuelve dos problemas al mismo tiempo.

✗ El patrón Singleton puede enmascarar un mal diseño, por ejemplo, cuando los componentes del programa saben demasiado los unos sobre los otros.

✗ El patrón requiere de un tratamiento especial en un entorno con múltiples hilos de ejecución, para que varios hilos no creen un objeto Singleton varias veces.

✗ Puede resultar complicado realizar la prueba unitaria del código cliente del Singleton porque muchos *frameworks* de prueba dependen de la herencia a la hora de crear objetos simulados (mock objects). Debido a que la clase Singleton es privada y en la mayoría de los lenguajes resulta imposible sobrescribir métodos estáticos, tendrás que pensar en una manera original de simular el Singleton. O, simplemente, no escribas las pruebas. O no utilices el patrón Singleton.

www.detodoprogramacion.org

⇄ Relaciones con otros patrones

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

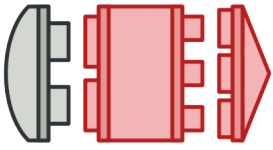
- **Flyweight** podría asemejarse a **Singleton** si de algún modo pudieras reducir todos los estados compartidos de los objetos a un único objeto flyweight. Pero existen dos diferencias fundamentales entre estos patrones:

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

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

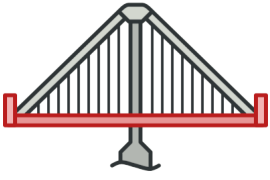
Patrones estructurales

Los patrones estructurales explican cómo ensamblar objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.



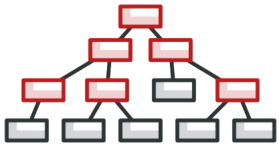
Adapter

Permite la colaboración entre objetos con interfaces incompatibles.



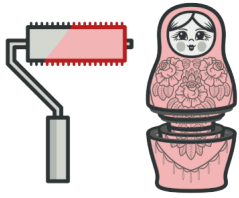
Bridge

Permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



Composite

Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.



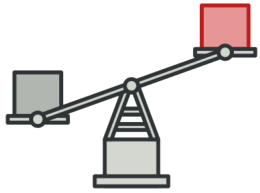
Decorator

Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



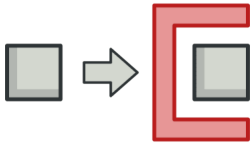
Facade

Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.



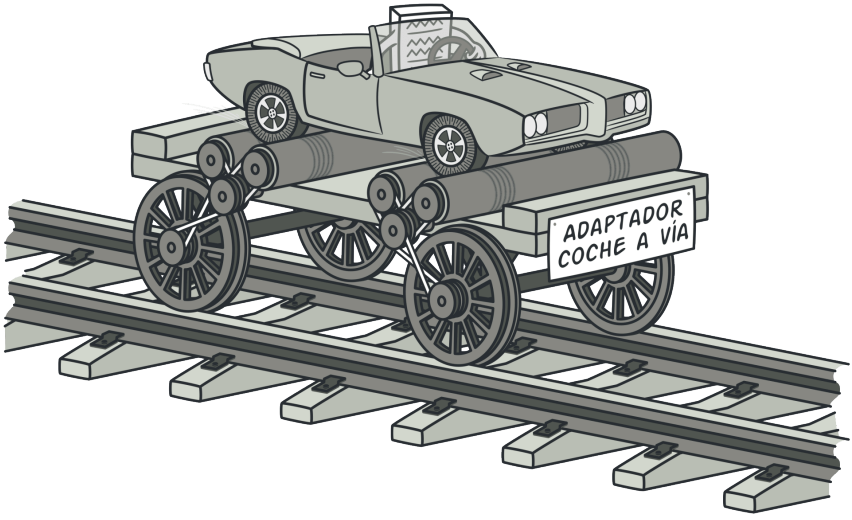
Flyweight

Permite mantener más objetos dentro de la cantidad disponible de memoria RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.



Proxy

Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.



ADAPTER

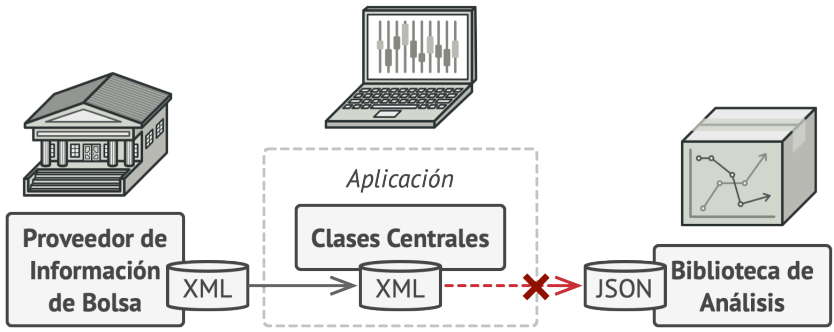
También llamado: Adaptador, Envoltorio, Wrapper

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

☹ Problema

Imagina que estás creando una aplicación de monitoreo del mercado de valores. La aplicación descarga la información de bolsa desde varias fuentes en formato XML para presentarla al usuario con bonitos gráficos y diagramas.

En cierto momento, decides mejorar la aplicación integrando una inteligente biblioteca de análisis de una tercera persona. Pero hay una trampa: la biblioteca de análisis solo funciona con datos en formato JSON.



No puedes utilizar la biblioteca de análisis "tal cual" porque ésta espera los datos en un formato que es incompatible con tu aplicación.

Podrías cambiar la biblioteca para que funcione con XML. Sin embargo, esto podría descomponer parte del código existente que depende de la biblioteca. Y, lo que es peor, podrías no tener siquiera acceso al código fuente de la biblioteca, lo que hace imposible esta solución.

😊 Solución

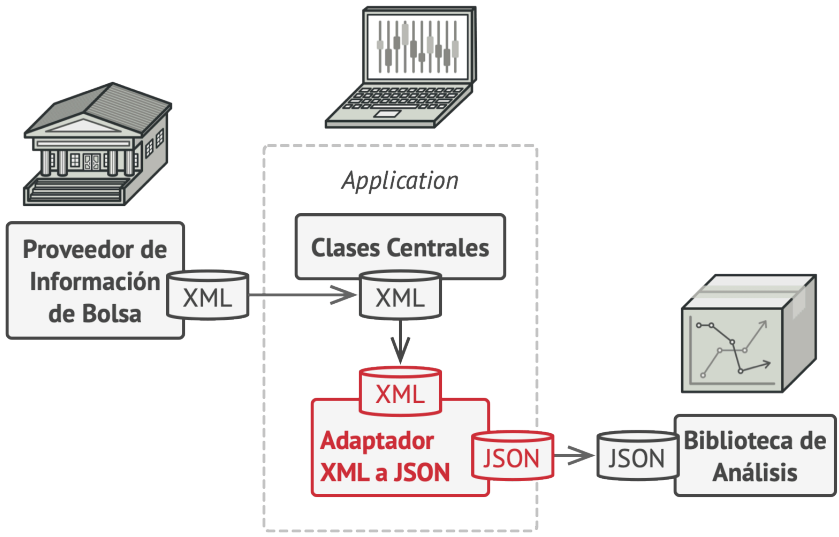
Puedes crear un *adaptador*. Se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

Un adaptador envuelve uno de los objetos para esconder la complejidad de la conversión que tiene lugar tras bambalinas. El objeto envuelto ni siquiera es consciente de la existencia del adaptador. Por ejemplo, puedes envolver un objeto que opera con metros y kilómetros con un adaptador que convierte todos los datos al sistema anglosajón, es decir, pies y millas.

Los adaptadores no solo convierten datos a varios formatos, sino que también ayudan a objetos con distintas interfaces a colaborar. Funciona así:

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
2. Utilizando esta interfaz, el objeto existente puede invocar con seguridad los métodos del adaptador.
3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto espera.

En ocasiones se puede incluso crear un adaptador de dos direcciones que pueda convertir las llamadas en ambos sentidos.



Regresemos a nuestra aplicación del mercado de valores. Para resolver el dilema de los formatos incompatibles, puedes crear adaptadores de XML a JSON para cada clase de la biblioteca de análisis con la que trabaje tu código directamente. Después ajustas tu código para que se comunique con la biblioteca únicamente a través de estos adaptadores. Cuando un adaptador recibe una llamada, traduce los datos XML entrantes a una estructura JSON y pasa la llamada a los métodos adecuados de un objeto de análisis envuelto.

Analogía en el mundo real

Cuando viajas de Europa a Estados Unidos por primera vez, puede ser que te lleves una sorpresa cuanto intentes cargar tu computadora portátil.



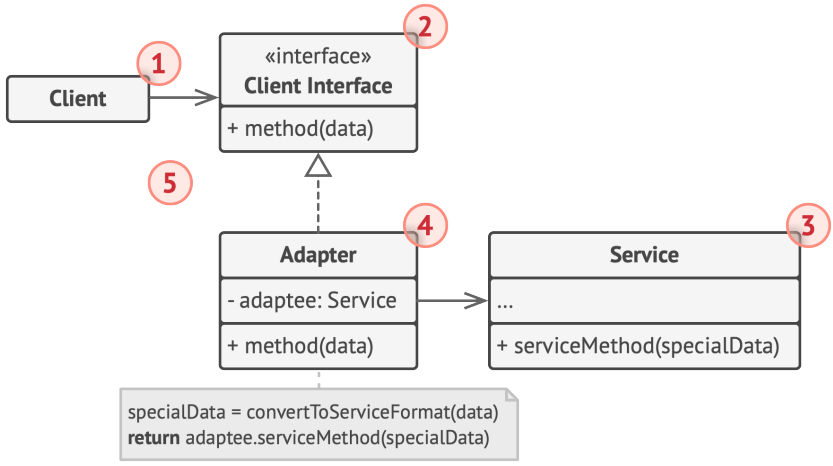
Una maleta antes y después de un viaje al extranjero.

Los tipos de enchufe son diferentes en cada país, por lo que un enchufe español no sirve en Estados Unidos. El problema puede solucionarse utilizando un adaptador que incluya el enchufe americano y el europeo.

Estructura

Adaptador de objetos

Esta implementación utiliza el principio de composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve el otro. Puede implementarse en todos los lenguajes de programación populares.



1. La clase **Cliente** contiene la lógica de negocio existente del programa.

2. La **Interfaz con el Cliente** describe un protocolo que otras clases deben seguir para poder colaborar con el código cliente.

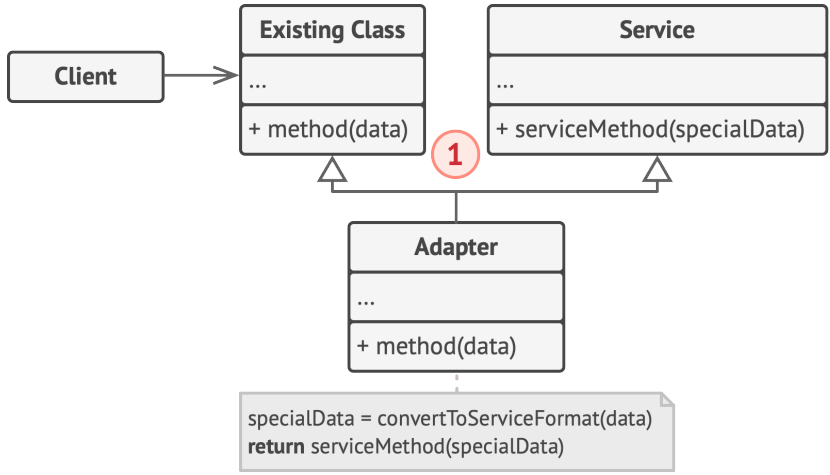
3. **Servicio** es alguna clase útil (normalmente de una tercera parte o heredada). El cliente no puede utilizar directamente esta clase porque tiene una interfaz incompatible.

4. La clase **Adaptadora** es capaz de trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente, mientras envuelve el objeto de la clase de servicio. La clase adaptadora recibe llamadas del cliente a través de la interfaz adaptadora y las traduce en llamadas al objeto envuelto de la clase de servicio, pero en un formato que pueda comprender.

5. El código cliente no se acopla a la clase adaptadora concreta siempre y cuando funcione con la clase adaptadora a través de la interfaz con el cliente. Gracias a esto, puedes introducir nuevos tipos de adaptadores en el programa sin descomponer el código cliente existente. Esto puede resultar útil cuando la interfaz de la clase de servicio se cambia o sustituye, ya que puedes crear una nueva clase adaptadora sin cambiar el código cliente.

Clase adaptadora

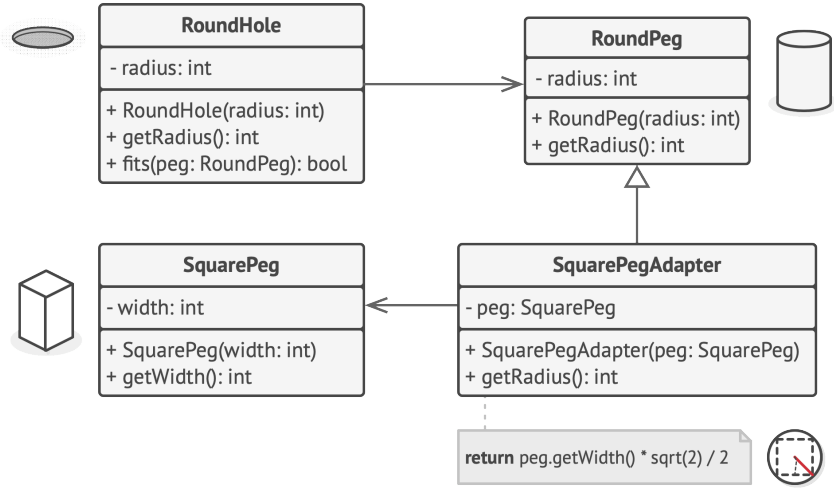
Esta implementación utiliza la herencia, porque la clase adaptadora hereda interfaces de ambos objetos al mismo tiempo. Ten en cuenta que esta opción sólo puede implementarse en lenguajes de programación que soporten la herencia múltiple, como C++.



1. La **Clase adaptadora** no necesita envolver objetos porque hereda comportamientos tanto de la clase cliente como de la clase de servicio. La adaptación tiene lugar dentro de los métodos sobrescritos. La clase adaptadora resultante puede utilizarse en lugar de una clase cliente existente.

Pseudocódigo

Este ejemplo del patrón **Adapter** se basa en el clásico conflicto entre piezas cuadradas y agujeros redondos.



Adaptando piezas cuadradas a agujeros redondos.

El patrón Adapter finge ser una pieza redonda con un radio igual a la mitad del diámetro del cuadrado (en otras palabras, el radio del círculo más pequeño en el que quepa la pieza cuadrada).

```
// Digamos que tienes dos clases con interfaces compatibles:
// RoundHole (HoyoRedondo) y RoundPeg (PiezaRedonda).
class RoundHole is
  constructor RoundHole(radius) { ... }

  method getRadius() is
    // Devuelve el radio del agujero.

  method fits(peg: RoundPeg) is
    return this.getRadius() >= peg.getRadius()

class RoundPeg is
  constructor RoundPeg(radius) { ... }

  method getRadius() is
    // Devuelve el radio de la pieza.


// Pero hay una clase incompatible: SquarePeg (PiezaCuadrada).
class SquarePeg is
  constructor SquarePeg(width) { ... }

  method getWidth() is
    // Devuelve la anchura de la pieza cuadrada.


// Una clase adaptadora te permite encajar piezas cuadradas en
// hoyos redondos. Extiende la clase RoundPeg para permitir a
// los objetos adaptadores actuar como piezas redondas.
class SquarePegAdapter extends RoundPeg is
  // En realidad, el adaptador contiene una instancia de la
  // clase SquarePeg.
```



El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la Programación Java, C/C++/C#,Visual.Net, SQL, Python, Javascript, Oracle, Algoritmos, CSS, Desarrollo Web, Joomla, jquery, Ajax y Mucho Mas…

VISITA

www.detodoprogramacion.org
www.detodopython.com
www.gratiscodigo.com

GRATISCODIGO.COM
Usa, lo que ya se creó...

Detodo Python.com

```
  private field peg: SquarePeg

  constructor SquarePegAdapter(peg: SquarePeg) is
    this.peg = peg

  method getRadius() is
    // El adaptador simula que es una pieza redonda con un
    // radio que pueda albergar la pieza cuadrada que el
    // adaptador envuelve.
    return peg.getWidth() * Math.sqrt(2) / 2


// En algún punto del código cliente.
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // verdadero

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // esto no compila (tipos incompatibles)

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // verdadero
hole.fits(large_sqpeg_adapter) // falso
```



Aplicabilidad



Utiliza la clase adaptadora cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.

El patrón Adapter te permite crear una clase intermedia que sirva como traductora entre tu código y una clase heredada, una clase de un tercero o cualquier otra clase con una interfaz extraña.

Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.

Puedes extender cada subclase y colocar la funcionalidad que falta, dentro de las nuevas clases hijas. No obstante, deberás duplicar el código en todas estas nuevas clases, lo cual **huele muy mal**.

Una solución mucho más elegante sería colocar la funcionalidad que falta dentro de una clase adaptadora. Después puedes envolver objetos a los que les falten funciones, dentro de la clase adaptadora, obteniendo esas funciones necesarias de un modo dinámico. Para que esto funcione, las clases en cuestión deben tener una interfaz común y el campo de la clase adaptadora debe seguir dicha interfaz. Este procedimiento es muy similar al del patrón **Decorator**.

Cómo implementarlo

1. Asegúrate de que tienes al menos dos clases con interfaces incompatibles:

- Una útil clase *servicio* que no puedes cambiar (a menudo de un tercero, heredada o con muchas dependencias existentes).
- Una o varias clases *cliente* que se beneficiarían de contar con una clase de servicio.

2. Declara la interfaz con el cliente y describe el modo en que las clases cliente se comunican con la clase de servicio.

3. Crea la clase adaptadora y haz que siga la interfaz con el cliente. Deja todos los métodos vacíos por ahora.

4. Añade un campo a la clase adaptadora para almacenar una referencia al objeto de servicio. La práctica común es inicializar este campo a través del constructor, pero en ocasiones es adecuado pasarlo al adaptador cuando se invocan sus métodos.

5. Uno por uno, implementa todos los métodos de la interfaz con el cliente en la clase adaptadora. La clase adaptadora deberá delegar la mayor parte del trabajo real al objeto de servicio, gestionando tan solo la interfaz o la conversión de formato de los datos.

6. Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente. Esto te permitirá cambiar o extender las clases adaptadoras sin afectar al código cliente.

www.detodoprogramacion.org

Pros y contras

✓ *Principio de responsabilidad única*. Puedes separar la interfaz o el código de conversión de datos de la lógica de negocio primaria del programa.

✓ *Principio de abierto/cerrado*. Puedes introducir nuevos tipos de adaptadores al programa sin descomponer el código cliente existente, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.

✕ La complejidad general del código aumenta, ya que debes introducir un grupo de nuevas interfaces y clases. En ocasiones resulta más sencillo cambiar la clase de servicio de modo que coincida con el resto de tu código.

Relaciones con otros patrones

- **Bridge** suele diseñarse por anticipado, lo que te permite desarrollar partes de una aplicación de forma independiente entre sí. Por otro lado, **Adapter** se utiliza habitualmente con una aplicación existente para hacer que unas clases que de otro modo serían incompatibles, trabajen juntas sin problemas.

- **Adapter** cambia la interfaz de un objeto existente mientras que **Decorator** mejora un objeto sin cambiar su interfaz. Además, *Decorator* soporta la composición recursiva, lo cual no es posible al utilizar *Adapter*.

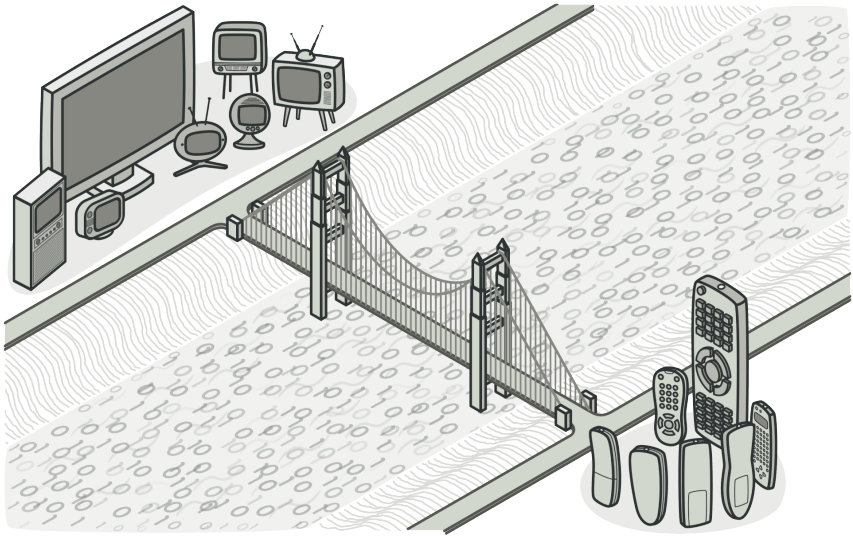
www.detodoprogramacion.org

- **Adapter** proporciona una interfaz diferente al objeto envuelto, **Proxy** le proporciona la misma interfaz y **Decorator** le proporciona una interfaz mejorada.

- **Facade** define una nueva interfaz para objetos existentes, mientras que **Adapter** intenta hacer que la interfaz existente sea utilizable. Normalmente *Adapter* sólo envuelve un objeto, mientras que *Facade* trabaja con todo un subsistema de objetos.

- **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.

www.detodoprogramacion.org



BRIDGE

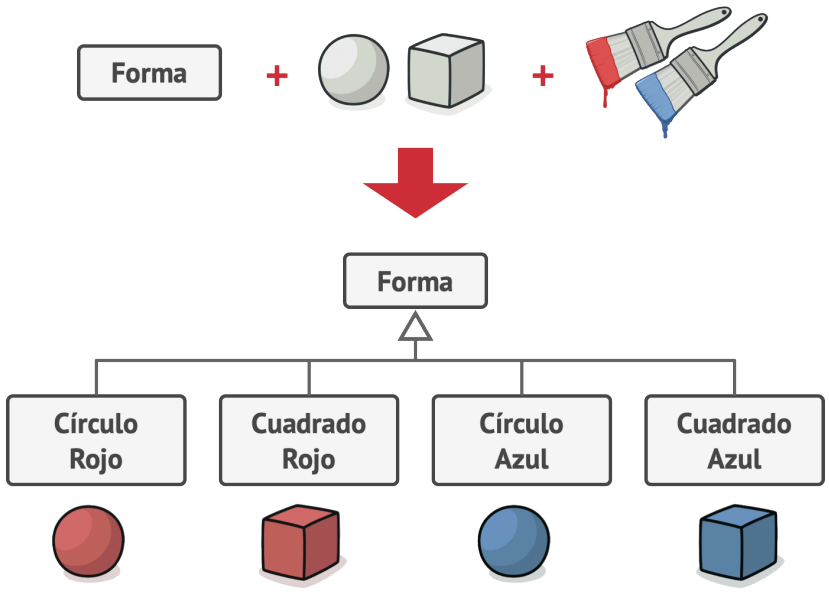
También llamado: Puente

Bridge es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

☹ Problema

¿Abstracción? ¿Implementación? ¿Asusta? Mantengamos la calma y veamos un ejemplo sencillo.

Digamos que tienes una clase geométrica `Forma` con un par de subclases: `Círculo` y `Cuadrado`. Deseas extender esta jerarquía de clase para que incorpore colores, por lo que planeas crear las subclases de forma `Rojo` y `Azul`. Sin embargo, como ya tienes dos subclases, tienes que crear cuatro combinaciones de clase, como `CírculoAzul` y `CuadradoRojo`.



El número de combinaciones de clase crece en progresión geométrica.

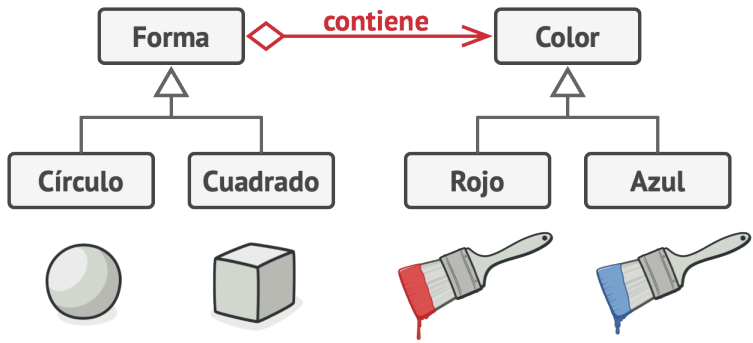
Añadir nuevos tipos de forma y color a la jerarquía hará que ésta crezca exponencialmente. Por ejemplo, para añadir una

forma de triángulo deberás introducir dos subclases, una para cada color. Y, después, para añadir un nuevo color habrá que crear tres subclases, una para cada tipo de forma. Cuanto más avancemos, peor será.

😊 Solución

Este problema se presenta porque intentamos extender las clases de forma en dos dimensiones independientes: por forma y por color. Es un problema muy habitual en la herencia de clases.

El patrón Bridge intenta resolver este problema pasando de la herencia a la composición del objeto. Esto quiere decir que se extrae una de las dimensiones a una jerarquía de clases separada, de modo que las clases originales referencian un objeto de la nueva jerarquía, en lugar de tener todo su estado y sus funcionalidades dentro de una clase.



Puedes evitar la explosión de una jerarquía de clase transformándola en varias jerarquías relacionadas.

Con esta solución, podemos extraer el código relacionado con el color y colocarlo dentro de su propia clase, con dos subclases: `Rojo` y `Azul`. La clase `Forma` obtiene entonces un campo de referencia que apunta a uno de los objetos de color. Ahora la forma puede delegar cualquier trabajo relacionado con el color al objeto de color vinculado. Esa referencia actuará como un puente entre las clases `Forma` y `Color`. En adelante, añadir nuevos colores no exigirá cambiar la jerarquía de forma y viceversa.

Abstracción e implementación

El libro de la GoF[1] introduce los términos *Abstracción* e *Implementación* como parte de la definición del patrón Bridge. En mi opinión, los términos suenan demasiado académicos y provocan que el patrón parezca más complicado de lo que es en realidad. Una vez leído el sencillo ejemplo con las formas y los colores, vamos a descifrar el significado que esconden las temibles palabras del libro de esta banda de cuatro.

La *Abstracción* (también llamada *interfaz*) es una capa de control de alto nivel para una entidad. Esta capa no tiene que hacer ningún trabajo real por su cuenta, sino que debe delegar el trabajo a la capa de *implementación* (también llamada *plataforma*).

1. "Gang of Four" (banda de los cuatro) es el sobrenombre de los cuatro autores del primer libro sobre patrones de diseño: *Patrones de diseño* **https://refactoring.guru/es/gof-book**.

www.detodoprogramacion.org

Ten en cuenta que no estamos hablando de las *interfaces* o las *clases abstractas* de tu lenguaje de programación. Son cosas diferentes.

Cuando hablamos de aplicación reales, la abstracción puede representarse por una interfaz gráfica de usuario (GUI), y la implementación puede ser el código del sistema operativo subyacente (API) a la que la capa GUI llama en respuesta a las interacciones del usuario.

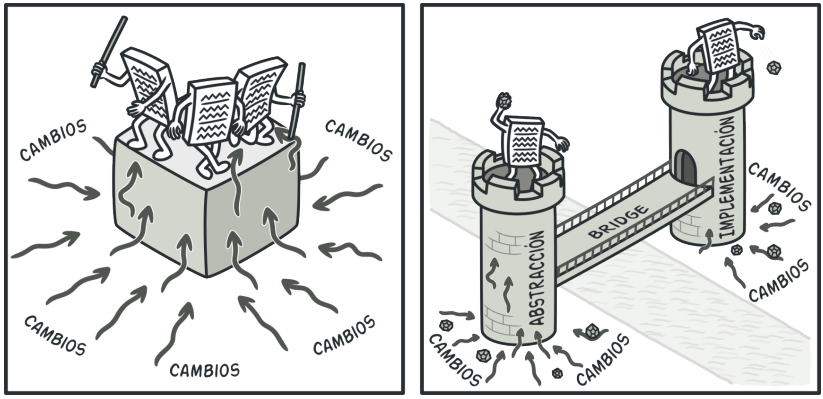
En términos generales, puedes extender esa aplicación en dos direcciones independientes:

- Tener varias GUI diferentes (por ejemplo, personalizadas para clientes regulares o administradores).
- Soportar varias API diferentes (por ejemplo, para poder lanzar la aplicación con Windows, Linux y macOS).

En el peor de los casos, esta aplicación podría asemejarse a un plato gigante de espagueti, en el que cientos de condicionales conectan distintos tipos de GUI con varias API por todo el código.

Puedes poner orden en este caos metiendo el código relacionado con combinaciones específicas interfaz-plataforma dentro de clases independientes. Sin embargo, pronto descubrirás que hay *muchas* de estas clases.

www.detodoprogramacion.org

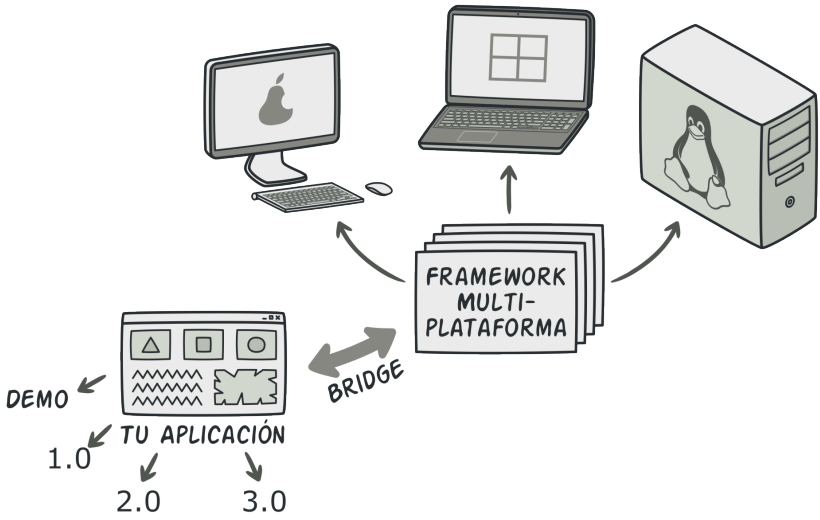


Realizar incluso un cambio sencillo en una base de código monolítica es bastante difícil porque debes comprender todo el asunto muy bien. Es mucho más sencillo realizar cambios en módulos más pequeños y bien definidos.

La jerarquía de clase crecerá exponencialmente porque añadir una nueva GUI o soportar una API diferente exigirá que se creen más y más clases.

Intentemos resolver este problema con el patrón Bridge, que nos sugiere que dividamos las clases en dos jerarquías:

- Abstracción: la capa GUI de la aplicación.
- Implementación: las API de los sistemas operativos.



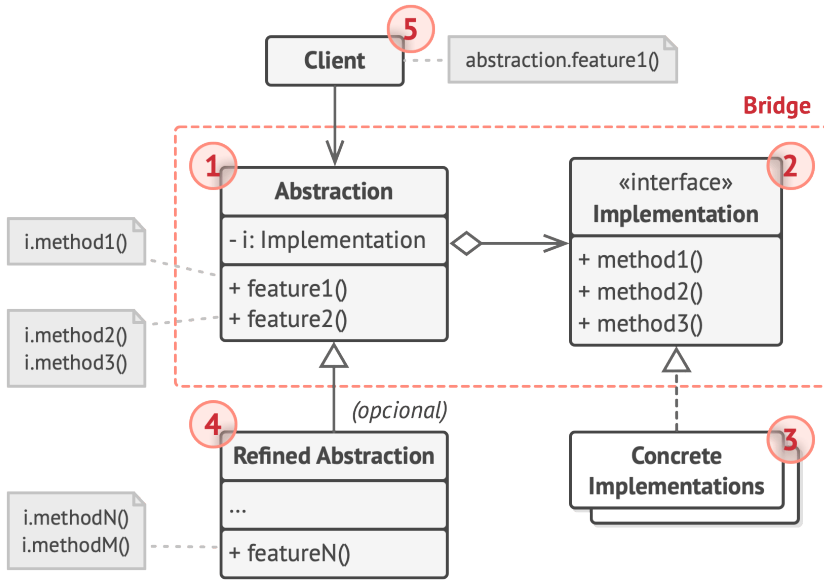
Una de las formas de estructurar una aplicación multiplataforma.

El objeto de la abstracción controla la apariencia de la aplicación, delegando el trabajo real al objeto de la implementación vinculado. Las distintas implementaciones son intercambiables siempre y cuando sigan una interfaz común, permitiendo a la misma GUI funcionar con Windows y Linux.

En consecuencia, puedes cambiar las clases de la GUI sin tocar las clases relacionadas con la API. Además, añadir soporte para otro sistema operativo sólo requiere crear una subclase en la jerarquía de implementación.

www.detodoprogramacion.org

Estructura



1. La **Abstracción** ofrece lógica de control de alto nivel. Depende de que el objeto de la implementación haga el trabajo de bajo nivel.

2. La **Implementación** declara la interfaz común a todas las implementaciones concretas. Una abstracción sólo se puede comunicar con un objeto de implementación a través de los métodos que se declaren aquí.

   La abstracción puede enumerar los mismos métodos que la implementación, pero normalmente la abstracción declara funcionalidades complejas que dependen de una amplia variedad de operaciones primitivas declaradas por la implementación.

3. Las **Implementaciones Concretas** contienen código específico de plataforma.

4. Las **Abstracciones Refinadas** proporcionan variantes de lógica de control. Como sus padres, trabajan con distintas implementaciones a través de la interfaz general de implementación.

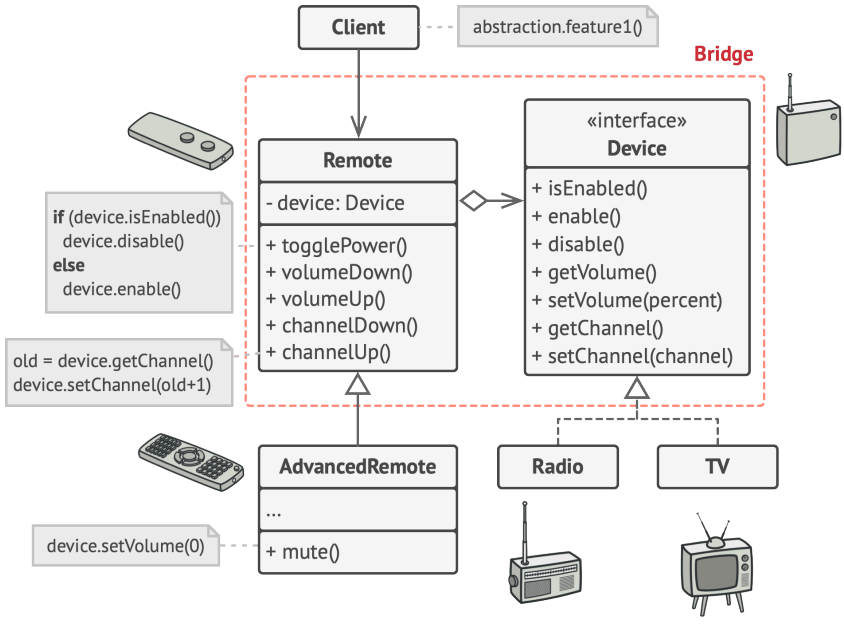
5. Normalmente, el **Cliente** sólo está interesado en trabajar con la abstracción. No obstante, el cliente tiene que vincular el objeto de la abstracción con uno de los objetos de la implementación.

Pseudocódigo

Este ejemplo ilustra cómo puede ayudar el patrón **Bridge** a dividir el código monolítico de una aplicación que gestiona dispositivos y sus controles remotos. Las clases `Dispositivo` actúan como implementación, mientras que las clases `Remoto` actúan como abstracción.

Descargado de: www.detodoprogramacion.org

www.detodoprogramacion.org



La jerarquía de clase original se divide en dos partes: dispositivos y controles remotos.

La clase base de control remoto declara un campo de referencia que la vincula con un objeto de dispositivo. Todos los controles remotos funcionan con los dispositivos a través de la interfaz general de dispositivos, que permite al mismo remoto soportar varios tipos de dispositivos.

Puedes desarrollar las clases de control remoto independientemente de las clases de dispositivo. Lo único necesario es crear una nueva subclase de control remoto. Por ejemplo, puede ser que un control remoto básico cuente tan solo con dos botones, pero puedes extenderlo añadiéndole funciones, como una batería adicional o pantalla táctil.

El código cliente vincula el tipo deseado de control remoto con un objeto específico de dispositivo a través del constructor del control remoto.

```
// La "abstracción" define la interfaz para la parte de
// "control" de las dos jerarquías de clase. Mantiene una
// referencia a un objeto de la jerarquía de "implementación" y
// delega todo el trabajo real a este objeto.
class RemoteControl is
  protected field device: Device
  constructor RemoteControl(device: Device) is
    this.device = device
  method togglePower() is
    if (device.isEnabled()) then
      device.disable()
    else
      device.enable()
  method volumeDown() is
    device.setVolume(device.getVolume() - 10)
  method volumeUp() is
    device.setVolume(device.getVolume() + 10)
  method channelDown() is
    device.setChannel(device.getChannel() - 1)
  method channelUp() is
    device.setChannel(device.getChannel() + 1)


// Puedes extender clases de la jerarquía de abstracción
// independientemente de las clases de dispositivo.
class AdvancedRemoteControl extends RemoteControl is
  method mute() is
```

```
    device.setVolume(0)


// La interfaz de "implementación" declara métodos comunes a
// todas las clases concretas de implementación. No tiene por
// qué coincidir con la interfaz de la abstracción. De hecho,
// las dos interfaces pueden ser completamente diferentes.
// Normalmente, la interfaz de implementación únicamente
// proporciona operaciones primitivas, mientras que la
// abstracción define operaciones de más alto nivel con base en
// las primitivas.
interface Device is
  method isEnabled()
  method enable()
  method disable()
  method getVolume()
  method setVolume(percent)
  method getChannel()
  method setChannel(channel)


// Todos los dispositivos siguen la misma interfaz.
class Tv implements Device is
  // ...

class Radio implements Device is
  // ...


// En algún lugar del código cliente.
tv = new Tv()
remote = new RemoteControl(tv)
```

```
60  remote.togglePower()
61
62  radio = new Radio()
63  remote = new AdvancedRemoteControl(radio)
```

Aplicabilidad

Utiliza el patrón Bridge cuando quieras dividir y organizar una clase monolítica que tenga muchas variantes de una sola funcionalidad (por ejemplo, si la clase puede trabajar con diversos servidores de bases de datos).

Conforme más crece una clase, más difícil resulta entender cómo funciona y más tiempo se tarda en realizar un cambio. Cambiar una de las variaciones de funcionalidad puede exigir realizar muchos cambios a toda la clase, lo que a menudo provoca que se cometan errores o no se aborden algunos de los efectos colaterales críticos.

El patrón Bridge te permite dividir la clase monolítica en varias jerarquías de clase. Después, puedes cambiar las clases de cada jerarquía independientemente de las clases de las otras. Esta solución simplifica el mantenimiento del código y minimiza el riesgo de descomponer el código existente.

Utiliza el patrón cuando necesites extender una clase en varias dimensiones ortogonales (independientes).

El patrón Bridge sugiere que extraigas una jerarquía de clase separada para cada una de las dimensiones. La clase original delega el trabajo relacionado a los objetos pertenecientes a dichas jerarquías, en lugar de hacerlo todo por su cuenta.

Utiliza el patrón Bridge cuando necesites poder cambiar implementaciones durante el tiempo de ejecución.

Aunque es opcional, el patrón Bridge te permite sustituir el objeto de implementación dentro de la abstracción. Es tan sencillo como asignar un nuevo valor a un campo.

*Por cierto, este último punto es la razón principal por la que tanta gente confunde el patrón Bridge con el patrón **Strategy**. Recuerda que un patrón es algo más que un cierto modo de estructurar tus clases. También puede comunicar intención y el tipo de problema que se está abordando.*

Cómo implementarlo

1. Identifica las dimensiones ortogonales de tus clases. Estos conceptos independientes pueden ser: abstracción/plataforma, dominio/infraestructura, *front end*/*back end*, o interfaz/implementación.

2. Comprueba qué operaciones necesita el cliente y defínelas en la clase base de abstracción.

www.detodoprogramacion.org

3. Determina las operaciones disponibles en todas las plataformas. Declara aquellas que necesite la abstracción en la interfaz general de implementación.

4. Crea clases concretas de implementación para todas las plataformas de tu dominio, pero asegúrate de que todas sigan la interfaz de implementación.

5. Dentro de la clase de abstracción añade un campo de referencia para el tipo de implementación. La abstracción delega la mayor parte del trabajo al objeto de la implementación referenciado en ese campo.

6. Si tienes muchas variantes de lógica de alto nivel, crea abstracciones refinadas para cada variante extendiendo la clase base de abstracción.

7. El código cliente debe pasar un objeto de implementación al constructor de la abstracción para asociar el uno con el otro. Después, el cliente puede ignorar la implementación y trabajar solo con el objeto de la abstracción.

Pros y contras

- ✓ Puedes crear clases y aplicaciones independientes de plataforma.
- ✓ El código cliente funciona con abstracciones de alto nivel. No está expuesto a los detalles de la plataforma.

www.detodoprogramacion.org

✓ *Principio de abierto/cerrado*. Puedes introducir nuevas abstracciones e implementaciones independientes entre sí.

✓ *Principio de responsabilidad única*. Puedes centrarte en la lógica de alto nivel en la abstracción y en detalles de la plataforma en la implementación.

✕ Puede ser que el código se complique si aplicas el patrón a una clase muy cohesionada.

⇄ Relaciones con otros patrones

- **Bridge** suele diseñarse por anticipado, lo que te permite desarrollar partes de una aplicación de forma independiente entre sí. Por otro lado, **Adapter** se utiliza habitualmente con una aplicación existente para hacer que unas clases que de otro modo serían incompatibles, trabajen juntas sin problemas.

- **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.

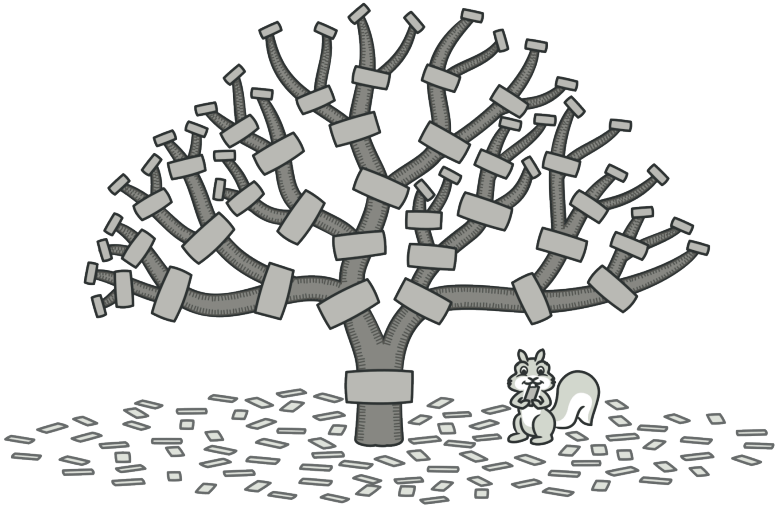
- Puedes utilizar **Abstract Factory** junto a **Bridge**. Este emparejamiento resulta útil cuando algunas abstracciones definidas por *Bridge* sólo pueden funcionar con implementaciones específi-

www.detodoprogramacion.org

cas. En este caso, *Abstract Factory* puede encapsular estas relaciones y esconder la complejidad al código cliente.

- Puedes combinar **Builder** con **Bridge**: la clase *directora* juega el papel de la abstracción, mientras que diferentes *constructoras* actúan como *implementaciones*.





COMPOSITE

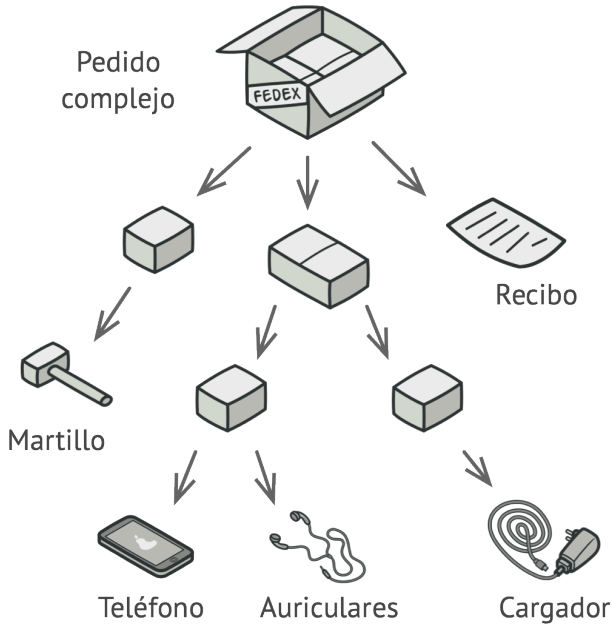
También llamado: Objeto compuesto, Object Tree

Composite es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

☹ Problema

El uso del patrón Composite sólo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.

Por ejemplo, imagina que tienes dos tipos de objetos: `Productos` y `Cajas`. Una `Caja` puede contener varios `Productos` así como cierto número de `Cajas` más pequeñas. Estas `Cajas` pequeñas también pueden contener algunos `Productos` o incluso `Cajas` más pequeñas, y así sucesivamente.



Un pedido puede incluir varios productos empaquetados en cajas, que a su vez están empaquetados en cajas más grandes y así sucesivamente. La estructura se asemeja a un árbol boca abajo.

Digamos que decides crear un sistema de pedidos que utiliza estas clases. Los pedidos pueden contener productos sencillos sin envolver, así como cajas llenas de productos… y otras cajas. ¿Cómo determinarás el precio total de ese pedido?

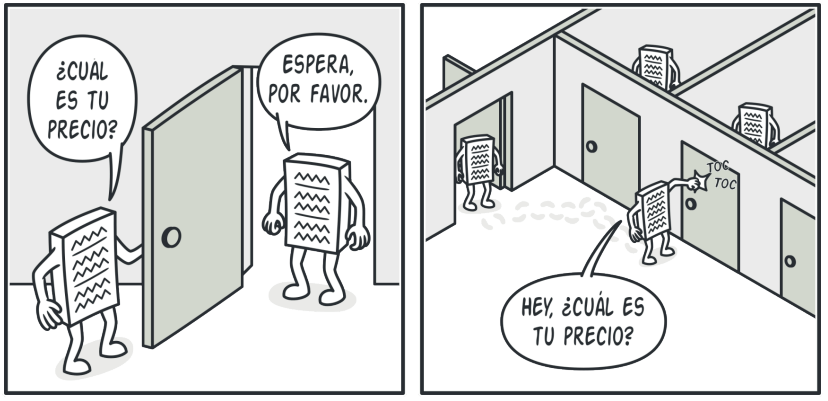
Puedes intentar la solución directa: desenvolver todas las cajas, repasar todos los productos y calcular el total. Esto sería viable en el mundo real; pero en un programa no es tan fácil como ejecutar un bucle. Tienes que conocer de antemano las clases de `Productos` y `Cajas` a iterar, el nivel de anidación de las cajas y otros detalles desagradables. Todo esto provoca que la solución directa sea demasiado complicada, o incluso imposible.

😊 Solución

El patrón Composite sugiere que trabajes con `Productos` y `Cajas` a través de una interfaz común que declara un método para calcular el precio total.

¿Cómo funcionaría este método? Para un producto, sencillamente devuelve el precio del producto. Para una caja, recorre cada artículo que contiene la caja, pregunta su precio y devuelve un total por la caja. Si uno de esos artículos fuera una caja más pequeña, esa caja también comenzaría a repasar su contenido y así sucesivamente, hasta que se calcule el precio de todos los componentes internos. Una caja podría incluso añadir costos adicionales al precio final, como costos de empaquetado.

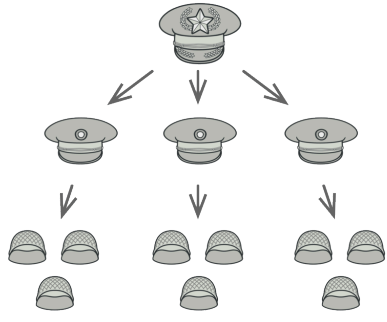
www.detodoprogramacion.org



El patrón Composite te permite ejecutar un comportamiento de forma recursiva sobre todos los componentes de un árbol de objetos.

La gran ventaja de esta solución es que no tienes que preocuparte por las clases concretas de los objetos que componen el árbol. No tienes que saber si un objeto es un producto simple o una sofisticada caja. Puedes tratarlos a todos por igual a través de la interfaz común. Cuando invocas un método, los propios objetos pasan la solicitud a lo largo del árbol.

Analogía en el mundo real

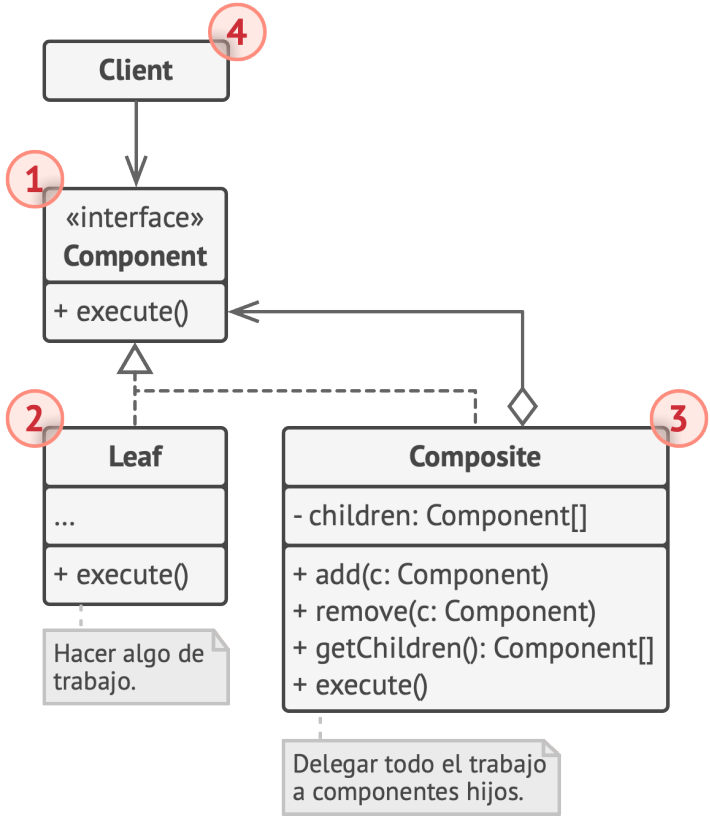


Un ejemplo de estructura militar.

www.detodoprogramacion.org

Los ejércitos de la mayoría de países se estructuran como jerarquías. Un ejército está formado por varias divisiones; una división es un grupo de brigadas y una brigada está formada por pelotones, que pueden dividirse en escuadrones. Por último, un escuadrón es un pequeño grupo de soldados reales. Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo por cada nivel hasta que todos los soldados saben lo que hay que hacer.

Estructura



1. La interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.

2. La **Hoja** es un elemento básico de un árbol que no tiene subelementos.

   Normalmente, los componentes de la hoja acaban realizando la mayoría del trabajo real, ya que no tienen a nadie a quien delegarle el trabajo.

3. El **Contenedor** (también llamado *compuesto*) es un elemento que tiene subelementos: hojas u otros contenedores. Un contenedor no conoce las clases concretas de sus hijos. Funciona con todos los subelementos únicamente a través de la interfaz componente.

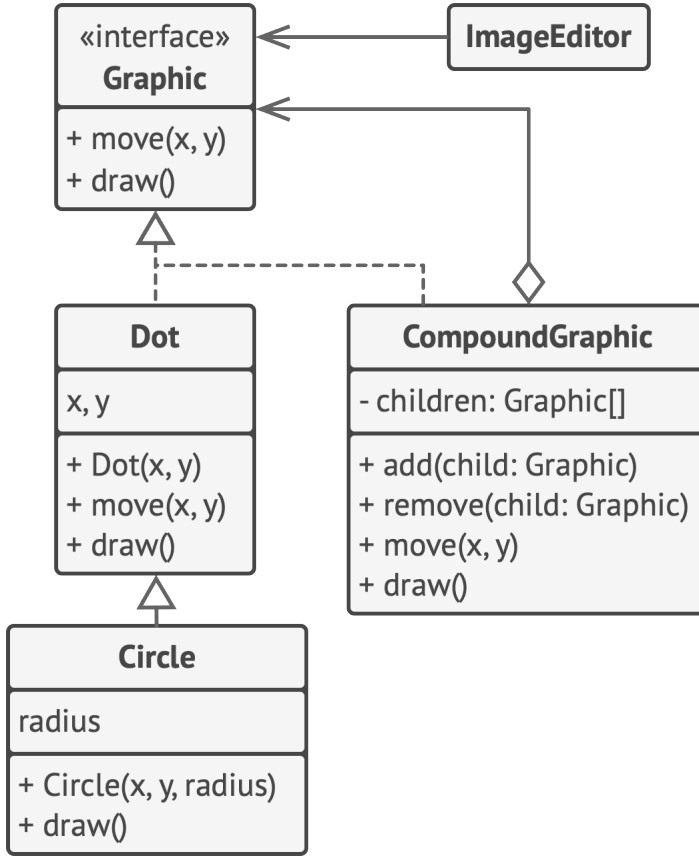
   Al recibir una solicitud, un contenedor delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al cliente.

4. El **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol.

www.detodoprogramacion.org

Pseudocódigo

En este ejemplo, el patrón **Composite** te permite implementar el apilamiento (*stacking*) de formas geométricas en un editor gráfico.



Ejemplo del editor de formas geométricas.

La clase `GráficoCompuesto` es un contenedor que puede incluir cualquier cantidad de subformas, incluyendo otras formas compuestas. Una forma compuesta tiene los mismos métodos

que una forma simple. Sin embargo, en lugar de hacer algo por su cuenta, una forma compuesta pasa la solicitud de forma recursiva a todos sus hijos y "suma" el resultado.

El código cliente trabaja con todas las formas a través de la interfaz común a todas las clases de forma. De este modo, el cliente no sabe si está trabajando con una forma simple o una compuesta. El cliente puede trabajar con estructuras de objetos muy complejas sin acoplarse a las clases concretas que forman esa estructura.

```
// La interfaz componente declara operaciones comunes para
// objetos simples y complejos de una composición.
interface Graphic is
  method move(x, y)
  method draw()

// La clase hoja representa objetos finales de una composición.
// Un objeto hoja no puede tener ningún subobjeto. Normalmente,
// son los objetos hoja los que hacen el trabajo real, mientras
// que los objetos compuestos se limitan a delegar a sus
// subcomponentes.
class Dot implements Graphic is
  field x, y

  constructor Dot(x, y) { ... }

  method move(x, y) is
    this.x += x, this.y += y

```

```
    method draw() is
      // Dibuja un punto en X e Y.

// Todas las clases de componente pueden extender otros
// componentes.
class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
      // Dibuja un círculo en X y Y con radio R.

// La clase compuesta representa componentes complejos que
// pueden tener hijos. Normalmente los objetos compuestos
// delegan el trabajo real a sus hijos y después "recapitulan"
// el resultado.
class CompoundGraphic implements Graphic is
    field children: array of Graphic

    // Un objeto compuesto puede añadir o eliminar otros
    // componentes (tanto simples como complejos) a o desde su
    // lista hija.
    method add(child: Graphic) is
      // Añade un hijo a la matriz de hijos.

    method remove(child: Graphic) is
      // Elimina un hijo de la matriz de hijos.

    method move(x, y) is
      foreach (child in children) do
        child.move(x, y)
```

```
52
53    // Un compuesto ejecuta su lógica primaria de una forma
54    // particular. Recorre recursivamente todos sus hijos,
55    // recopilando y recapitulando sus resultados. Debido a que
56    // los hijos del compuesto pasan esas llamadas a sus propios
57    // hijos y así sucesivamente, se recorre todo el árbol de
58    // objetos como resultado.
59    method draw() is
60      // 1. Para cada componente hijo:
61      //     - Dibuja el componente.
62      //     - Actualiza el rectángulo delimitador.
63      // 2. Dibuja un rectángulo de línea punteada utilizando
64      // las coordenadas de delimitación.
65
66
67  // El código cliente trabaja con todos los componentes a través
68  // de su interfaz base. De esta forma el código cliente puede
69  // soportar componentes de hoja simples así como compuestos
70  // complejos.
71  class ImageEditor is
72    field all: array of Graphic
73
74    method load() is
75      all = new CompoundGraphic()
76      all.add(new Dot(1, 2))
77      all.add(new Circle(5, 3, 10))
78      // ...
79
80    // Combina componentes seleccionados para formar un
81    // componente compuesto complejo.
82    method groupSelected(components: array of Graphic) is
83      group = new CompoundGraphic()
```

```
84      foreach (component in components) do
85        group.add(component)
86        all.remove(component)
87      all.add(group)
88      // Se dibujarán todos los componentes.
89      all.draw()
```

Aplicabilidad

Utiliza el patrón Composite cuando tengas que implementar una estructura de objetos con forma de árbol.

El patrón Composite te proporciona dos tipos de elementos básicos que comparten una interfaz común: hojas simples y contenedores complejos. Un contenedor puede estar compuesto por hojas y por otros contenedores. Esto te permite construir una estructura de objetos recursivos anidados parecida a un árbol.

Utiliza el patrón cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.

Todos los elementos definidos por el patrón Composite comparten una interfaz común. Utilizando esta interfaz, el cliente no tiene que preocuparse por la clase concreta de los objetos con los que funciona.

Cómo implementarlo

1. Asegúrate de que el modelo central de tu aplicación pueda representarse como una estructura de árbol. Intenta dividirlo en elementos simples y contenedores. Recuerda que los contenedores deben ser capaces de contener tanto elementos simples como otros contenedores.

2. Declara la interfaz componente con una lista de métodos que tengan sentido para componentes simples y complejos.

3. Crea una clase hoja para representar elementos simples. Un programa puede tener varias clases hoja diferentes.

4. Crea una clase contenedora para representar elementos complejos. Incluye un campo matriz en esta clase para almacenar referencias a subelementos. La matriz debe poder almacenar hojas y contenedores, así que asegúrate de declararla con el tipo de la interfaz componente.

   Al implementar los métodos de la interfaz componente, recuerda que un contenedor debe delegar la mayor parte del trabajo a los subelementos.

5. Por último, define los métodos para añadir y eliminar elementos hijos dentro del contenedor.

   Ten en cuenta que estas operaciones se pueden declarar en la interfaz componente. Esto violaría el *Principio de segregación*

www.detodoprogramacion.org

de la interfaz porque los métodos de la clase hoja estarían vacíos. No obstante, el cliente podrá tratar a todos los elementos de la misma manera, incluso al componer el árbol.

Pros y contras

✓ Puedes trabajar con estructuras de árbol complejas con mayor comodidad: utiliza el polimorfismo y la recursión en tu favor.

✓ *Principio de abierto/cerrado*. Puedes introducir nuevos tipos de elemento en la aplicación sin descomponer el código existente, que ahora funciona con el árbol de objetos.

✕ Puede resultar difícil proporcionar una interfaz común para clases cuya funcionalidad difiere demasiado. En algunos casos, tendrás que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender.

Relaciones con otros patrones

- Puedes utilizar **Builder** al crear árboles **Composite** complejos porque puedes programar sus pasos de construcción para que funcionen de forma recursiva.

- **Chain of Responsibility** se utiliza a menudo junto a **Composite**. En este caso, cuando un componente hoja recibe una solicitud, puede pasarla a lo largo de la cadena de todos los componentes padre hasta la raíz del árbol de objetos.

- Puedes utilizar **Iteradores** para recorrer árboles **Composite**.

www.detodoprogramacion.org

- Puedes utilizar el patrón **Visitor** para ejecutar una operación sobre un árbol **Composite** entero.

- Puedes implementar nodos de hoja compartidos del árbol **Composite** como **Flyweights** para ahorrar memoria RAM.

- **Composite** y **Decorator** tienen diagramas de estructura similares ya que ambos se basan en la composición recursiva para organizar un número indefinido de objetos.

  Un *Decorator* es como un *Composite* pero sólo tiene un componente hijo. Hay otra diferencia importante: *Decorator* añade responsabilidades adicionales al objeto envuelto, mientras que *Composite* se limita a "recapitular" los resultados de sus hijos.

  No obstante, los patrones también pueden colaborar: puedes utilizar el *Decorator* para extender el comportamiento de un objeto específico del árbol *Composite*.

- Los diseños que hacen un uso amplio de **Composite** y **Decorator** a menudo pueden beneficiarse del uso del **Prototype**. Aplicar el patrón te permite clonar estructuras complejas en lugar de reconstruirlas desde cero.

www.detodoprogramacion.org



DECORATOR

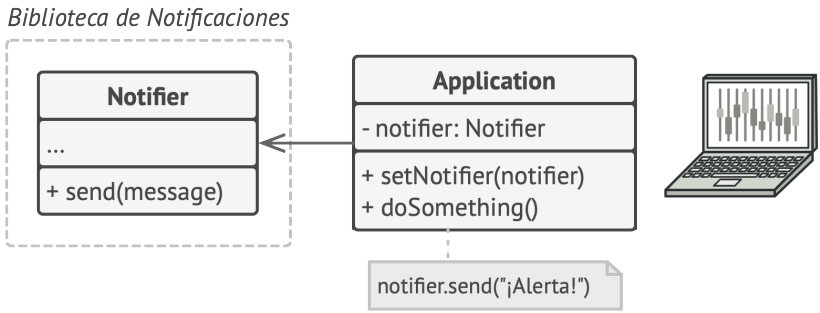
También llamado: Decorador, Envoltorio, Wrapper

Decorator es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

☹ Problema

Imagina que estás trabajando en una biblioteca de notificaciones que permite a otros programas notificar a sus usuarios acerca de eventos importantes.

La versión inicial de la biblioteca se basaba en la clase `Notificador` que solo contaba con unos cuantos campos, un constructor y un único método `send`. El método podía aceptar un argumento de mensaje de un cliente y enviar el mensaje a una lista de correos electrónicos que se pasaban a la clase notificadora a través de su constructor. Una aplicación de un tercero que actuaba como cliente debía crear y configurar el objeto notificador una vez y después utilizarlo cada vez que sucediera algo importante.

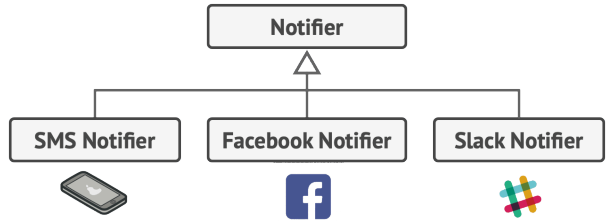


Un programa puede utilizar la clase notificadora para enviar notificaciones sobre eventos importantes a un grupo predefinido de correos electrónicos.

En cierto momento te das cuenta de que los usuarios de la biblioteca esperan algo más que unas simples notificaciones por correo. A muchos de ellos les gustaría recibir mensajes SMS

www.detodoprogramacion.org

sobre asuntos importantes. Otros querrían recibir las notificaciones por Facebook y, por supuesto, a los usuarios corporativos les encantaría recibir notificaciones por Slack.



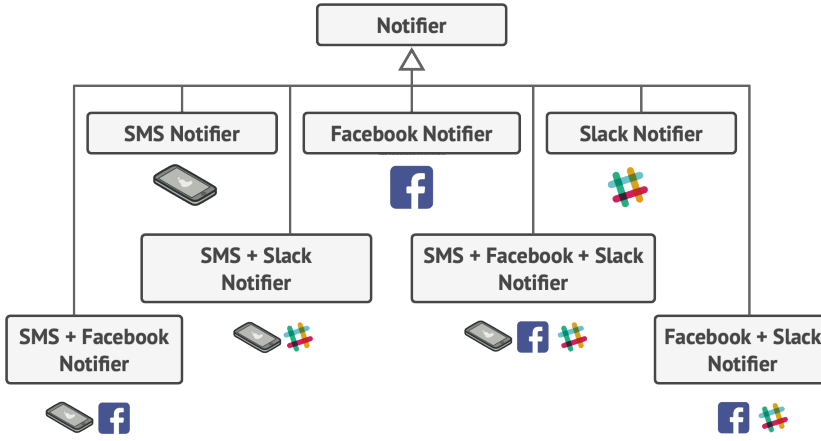
Cada tipo de notificación se implementa como una subclase de la clase notificadora.

No puede ser muy complicado ¿verdad? Extendiste la clase `Notificador` y metiste los métodos adicionales de notificación dentro de nuevas subclases. Ahora el cliente debería instanciar la clase notificadora deseada y utilizarla para el resto de notificaciones.

Pero entonces alguien te hace una pregunta razonable: "¿Por qué no se pueden utilizar varios tipos de notificación al mismo tiempo? Si tu casa está en llamas, probablemente quieras que te informen a través de todos los canales".

Intentaste solucionar ese problema creando subclases especiales que combinaban varios métodos de notificación dentro de una clase. Sin embargo, enseguida resultó evidente que esta solución inflaría el código en gran medida, no sólo el de la biblioteca, sino también el código cliente.

www.detodoprogramacion.org



Explosión combinatoria de subclases.

Debes encontrar alguna otra forma de estructurar las clases de las notificaciones para no alcanzar cifras que rompan accidentalmente un récord Guinness.

😊 Solución

Cuando tenemos que alterar la funcionalidad de un objeto, lo primero que se viene a la mente es extender una clase. No obstante, la herencia tiene varias limitaciones importantes de las que debes ser consciente.

- La herencia es estática. No se puede alterar la funcionalidad de un objeto existente durante el tiempo de ejecución. Sólo se puede sustituir el objeto completo por otro creado a partir de una subclase diferente.

- Las subclases sólo pueden tener una clase padre. En la mayoría de lenguajes, la herencia no permite a una clase heredar comportamientos de varias clases al mismo tiempo.

  Una de las formas de superar estas limitaciones es empleando la *Agregación* o la *Composición*[1] en lugar de la *Herencia*. Ambas alternativas funcionan prácticamente del mismo modo: un objeto *tiene una* referencia a otro y le delega parte del trabajo, mientras que con la herencia, el propio objeto *puede* realizar ese trabajo, heredando el comportamiento de su superclase.

  Con esta nueva solución puedes sustituir fácilmente el objeto "ayudante" vinculado por otro, cambiando el comportamiento del contenedor durante el tiempo de ejecución. Un objeto puede utilizar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas. La agregación/composición es el principio clave que se esconde tras muchos patrones de diseño, incluyendo el Decorator. A propósito, regresemos a la discusión sobre el patrón.



Herencia vs. Agregación

1. *Agregación*: el objeto A contiene objetos B; B puede existir sin A.
   Composición: el objeto A está compuesto de objetos B; A gestiona el ciclo vital de B; B no puede existir sin A.

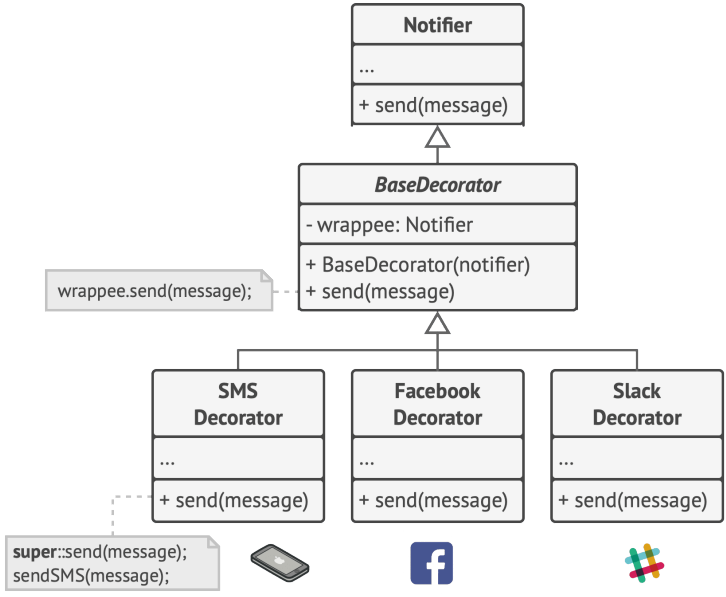
www.detodoprogramacion.org

"Wrapper" (envoltorio, en inglés) es el sobrenombre alternativo del patrón Decorator, que expresa claramente su idea principal. Un *wrapper* es un objeto que puede vincularse con un objeto *objetivo*. El wrapper contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. No obstante, el wrapper puede alterar el resultado haciendo algo antes o después de pasar la solicitud al objetivo.

¿Cuándo se convierte un simple wrapper en el verdadero decorador? Como he mencionado, el wrapper implementa la misma interfaz que el objeto envuelto. Éste es el motivo por el que, desde la perspectiva del cliente, estos objetos son idénticos. Haz que el campo de referencia del wrapper acepte cualquier objeto que siga esa interfaz. Esto te permitirá *envolver* un objeto en varios wrappers, añadiéndole el comportamiento combinado de todos ellos.

En nuestro ejemplo de las notificaciones, dejemos la sencilla funcionalidad de las notificaciones por correo electrónico dentro de la clase base `Notificador`, pero convirtamos el resto de los métodos de notificación en decoradores.

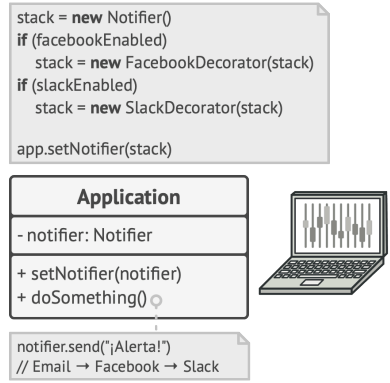
www.detodoprogramacion.org



Varios métodos de notificación se convierten en decoradores.

El código cliente debe envolver un objeto notificador básico dentro de un grupo de decoradores que satisfagan las preferencias del cliente. Los objetos resultantes se estructurarán como una pila.

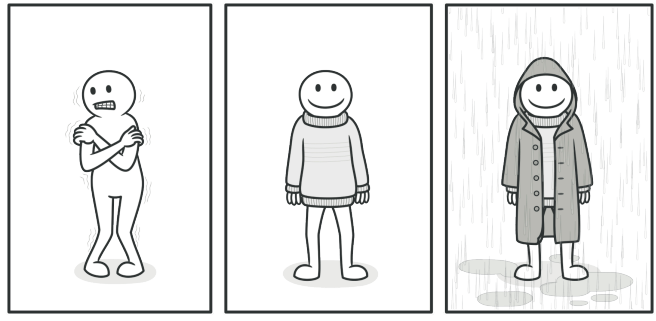
El último decorador de la pila será el objeto con el que el cliente trabaja. Debido a que todos los decoradores implementan la misma interfaz que la notificadora base, al resto del código cliente no le importa si está trabajando con el objeto notificador "puro" o con el decorado.



Las aplicaciones pueden configurar pilas complejas de decoradores de notificación.

Podemos aplicar la misma solución a otras funcionalidades, como el formateo de mensajes o la composición de una lista de destinatarios. El cliente puede decorar el objeto con los decoradores personalizados que desee, siempre y cuando sigan la misma interfaz que los demás.

Analogía en el mundo real



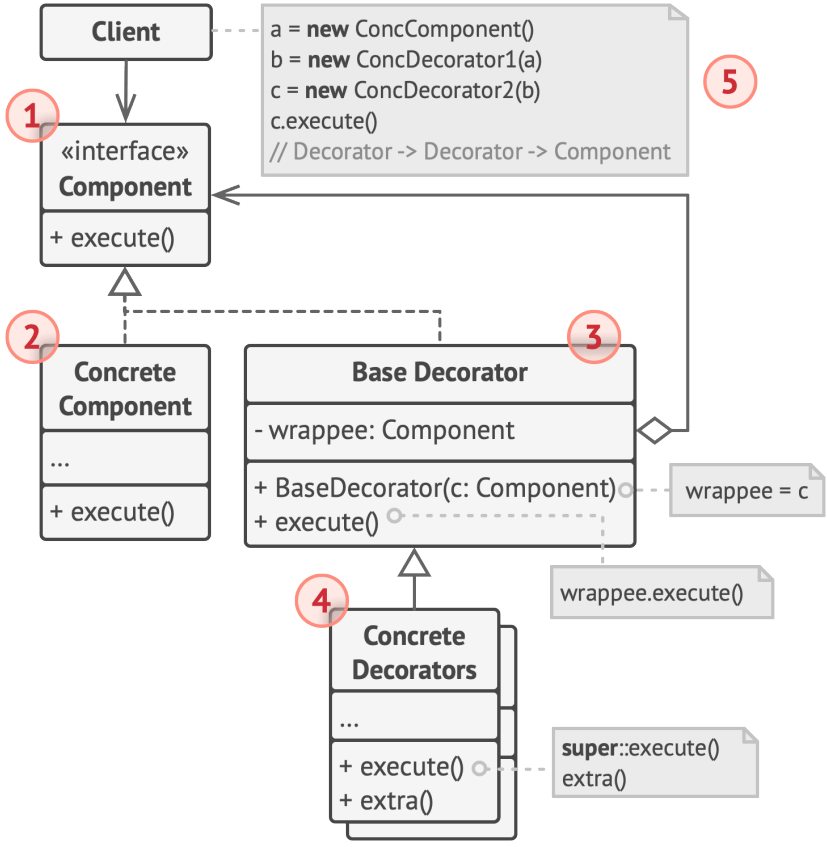
Obtienes un efecto combinado vistiendo varias prendas de ropa.

www.detodoprogramacion.org

Vestir ropa es un ejemplo del uso de decoradores. Cuando tienes frío, te cubres con un suéter. Si sigues teniendo frío a pesar del suéter, puedes ponerte una chaqueta encima. Si está lloviendo, puedes ponerte un impermeable. Todas estas prendas "extienden" tu comportamiento básico pero no son parte de ti, y puedes quitarte fácilmente cualquier prenda cuando lo desees.

Estructura

El Mundo de la Programación en tus Manos...!
DETODOPROGRAMACION.ORG



1. El **Componente** declara la interfaz común tanto para wrappers como para objetos envueltos.

2. **Componente Concreto** es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar.

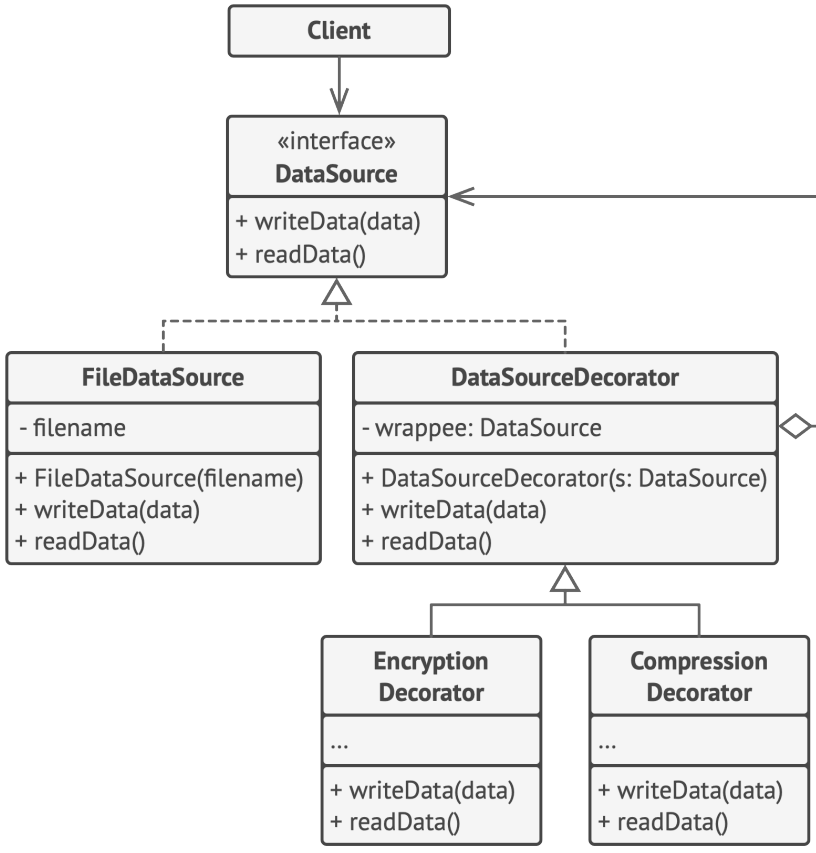
3. La clase **Decoradora Base** tiene un campo para referenciar un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto los componentes concretos como los decoradores. La clase decoradora base delega todas las operaciones al objeto envuelto.

4. Los **Decoradores Concretos** definen funcionalidades adicionales que se pueden añadir dinámicamente a los componentes. Los decoradores concretos sobrescriben métodos de la clase decoradora base y ejecutan su comportamiento, ya sea antes o después de invocar al método padre.

5. El **Cliente** puede envolver componentes en varias capas de decoradores, siempre y cuando trabajen con todos los objetos a través de la interfaz del componente.

Pseudocódigo

En este ejemplo, el patrón **Decorator** te permite comprimir y encriptar información delicada independientemente del código que utiliza esos datos.



Ejemplo de la encriptación y compresión de decoradores.

La aplicación envuelve el objeto de la fuente de datos con un par de decoradores. Ambos wrappers cambian el modo en que los datos se escriben y se leen en el disco:

- Justo antes de que los datos se **escriban en el disco**, los decoradores los encriptan y comprimen. La clase original escribe en el archivo los datos encriptados y protegidos, sin conocer el cambio.

- Después de que los datos son **leídos del disco**, pasan por los mismos decoradores, que los descomprimen y decodifican.

  Los decoradores y la clase fuente de datos implementan la misma interfaz, lo que los hace intercambiables en el código cliente.

```
// La interfaz de componente define operaciones que los
// decoradores pueden alterar.
interface DataSource is
  method writeData(data)
  method readData():data

// Los componentes concretos proporcionan implementaciones por
// defecto para las operaciones. En un programa puede haber
// muchas variaciones de estas clases.
class FileDataSource implements DataSource is
  constructor FileDataSource(filename) { ... }

  method writeData(data) is
    // Escribe datos en el archivo.

  method readData():data is
    // Lee datos del archivo.

// La clase decoradora base sigue la misma interfaz que los
// demás componentes. El principal propósito de esta clase es
// definir la interfaz de encapsulación para todos los
// decoradores concretos. La implementación por defecto del
// código de encapsulación puede incluir un campo para almacenar
// un componente envuelto y los medios para inicializarlo.
```

```
class DataSourceDecorator implements DataSource is
  protected field wrappee: DataSource

  constructor DataSourceDecorator(source: DataSource) is
    wrappee = source

  // La decoradora base simplemente delega todo el trabajo al
  // componente envuelto. En los decoradores concretos se
  // pueden añadir comportamientos adicionales.
  method writeData(data) is
    wrappee.writeData(data)

  // Los decoradores concretos pueden invocar la
  // implementación padre de la operación en lugar de invocar
  // directamente al objeto envuelto. Esta solución simplifica
  // la extensión de las clases decoradoras.
  method readData():data is
    return wrappee.readData()

// Los decoradores concretos deben invocar métodos en el objeto
// envuelto, pero pueden añadir algo de su parte al resultado.
// Los decoradores pueden ejecutar el comportamiento añadido
// antes o después de la llamada a un objeto envuelto.
class EncryptionDecorator extends DataSourceDecorator is
  method writeData(data) is
    // 1. Encripta los datos pasados.
    // 2. Pasa los datos encriptados al método writeData
    // (escribirDatos) del objeto envuelto.

  method readData():data is
    // 1. Obtiene datos del método readData (leerDatos) del
    // objeto envuelto.
```

```
57      // 2. Intenta descifrarlo si está encriptado.
58      // 3. Devuelve el resultado.
59
60  // Puedes envolver objetos en varias capas de decoradores.
61  class CompressionDecorator extends DataSourceDecorator is
62    method writeData(data) is
63      // 1. Comprime los datos pasados.
64      // 2. Pasa los datos comprimidos al método writeData del
65      // objeto envuelto.
66
67    method readData():data is
68      // 1. Obtiene datos del método readData del objeto
69      // envuelto.
70      // 2. Intenta descomprimirlo si está comprimido.
71      // 3. Devuelve el resultado.
72
73
74  // Opción 1. Un ejemplo sencillo del montaje de un decorador.
75  class Application is
76    method dumbUsageExample() is
77      source = new FileDataSource("somefile.dat")
78      source.writeData(salaryRecords)
79      // El archivo objetivo se ha escrito con datos sin
80      // formato.
81
82      source = new CompressionDecorator(source)
83      source.writeData(salaryRecords)
84      // El archivo objetivo se ha escrito con datos
85      // comprimidos.
86
87      source = new EncryptionDecorator(source)
88      // La variable fuente ahora contiene esto:
```

```
    // Cifrado > Compresión > FileDataSource
    source.writeData(salaryRecords)
    // El archivo se ha escrito con datos comprimidos y
    // encriptados.


// Opción 2. El código cliente que utiliza una fuente externa de
// datos. Los objetos SalaryManager no conocen ni se preocupan
// por las especificaciones del almacenamiento de datos.
// Trabajan con una fuente de datos preconfigurada recibida del
// configurador de la aplicación.
class SalaryManager is
  field source: DataSource

  constructor SalaryManager(source: DataSource) { ... }

  method load() is
    return source.readData()

  method save() is
    source.writeData(salaryRecords)
  // ...Otros métodos útiles...


// La aplicación puede montar distintas pilas de decoradores
// durante el tiempo de ejecución, dependiendo de la
// configuración o el entorno.
class ApplicationConfigurator is
  method configurationExample() is
    source = new FileDataSource("salary.dat")
    if (enabledEncryption)
      source = new EncryptionDecorator(source)
```

```
121       if (enabledCompression)
122         source = new CompressionDecorator(source)
123
124       logger = new SalaryManager(source)
125       salary = logger.load()
126     // ...
```

Aplicabilidad

Utiliza el patrón Decorator cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos.

El patrón Decorator te permite estructurar tu lógica de negocio en capas, crear un decorador para cada capa y componer objetos con varias combinaciones de esta lógica, durante el tiempo de ejecución. El código cliente puede tratar a todos estos objetos de la misma forma, ya que todos siguen una interfaz común.

Utiliza el patrón cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.

Muchos lenguajes de programación cuentan con la palabra clave `final` que puede utilizarse para evitar que una clase siga extendiéndose. Para una clase final, la única forma de reutilizar el comportamiento existente será envolver la clase con tu propio wrapper, utilizando el patrón Decorator.

Cómo implementarlo

1. Asegúrate de que tu dominio de negocio puede representarse como un componente primario con varias capas opcionales encima.

2. Decide qué métodos son comunes al componente primario y las capas opcionales. Crea una interfaz de componente y declara esos métodos en ella.

3. Crea una clase concreta de componente y define en ella el comportamiento base.

4. Crea una clase base decoradora. Debe tener un campo para almacenar una referencia a un objeto envuelto. El campo debe declararse con el tipo de interfaz de componente para permitir la vinculación a componentes concretos, así como a decoradores. La clase decoradora base debe delegar todas las operaciones al objeto envuelto.

5. Asegúrate de que todas las clases implementan la interfaz de componente.

6. Crea decoradores concretos extendiéndolos a partir de la decoradora base. Un decorador concreto debe ejecutar su comportamiento antes o después de la llamada al método padre (que siempre delega al objeto envuelto).

www.detodoprogramacion.org

7. El código cliente debe ser responsable de crear decoradores y componerlos del modo que el cliente necesite.

Pros y contras

✓ Puedes extender el comportamiento de un objeto sin crear una nueva subclase.

✓ Puedes añadir o eliminar responsabilidades de un objeto durante el tiempo de ejecución.

✓ Puedes combinar varios comportamientos envolviendo un objeto con varios decoradores.

✓ *Principio de responsabilidad única*. Puedes dividir una clase monolítica que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.

✕ Resulta difícil eliminar un wrapper específico de la pila de wrappers.

✕ Es difícil implementar un decorador de tal forma que su comportamiento no dependa del orden en la pila de decoradores.

✕ El código de configuración inicial de las capas pueden tener un aspecto desagradable.

Relaciones con otros patrones

- **Adapter** cambia la interfaz de un objeto existente mientras que **Decorator** mejora un objeto sin cambiar su interfaz. Además, *Decorator* soporta la composición recursiva, lo cual no es posible al utilizar *Adapter*.

www.detodoprogramacion.org

- **Adapter** proporciona una interfaz diferente al objeto envuelto, **Proxy** le proporciona la misma interfaz y **Decorator** le proporciona una interfaz mejorada.

- **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.

- **Composite** y **Decorator** tienen diagramas de estructura similares ya que ambos se basan en la composición recursiva para organizar un número indefinido de objetos.

  Un *Decorator* es como un *Composite* pero sólo tiene un componente hijo. Hay otra diferencia importante: *Decorator* añade responsabilidades adicionales al objeto envuelto, mientras que *Composite* se limita a "recapitular" los resultados de sus hijos.

  No obstante, los patrones también pueden colaborar: puedes utilizar el *Decorator* para extender el comportamiento de un objeto específico del árbol *Composite*.

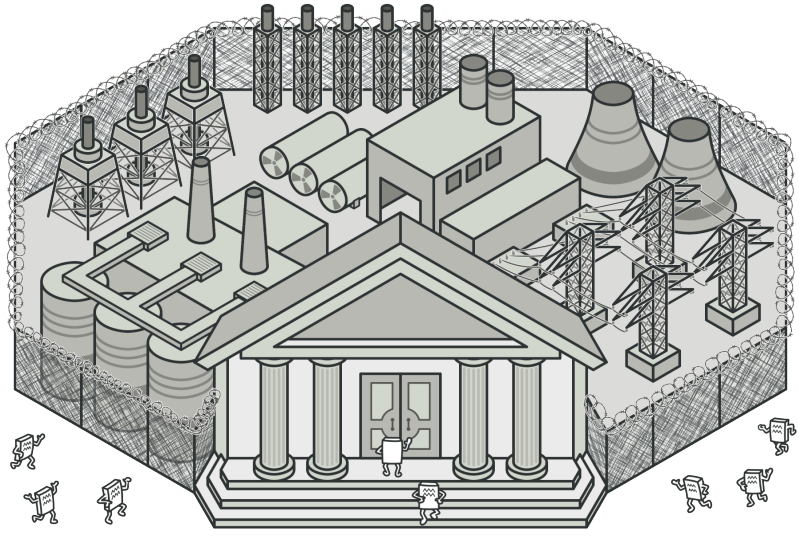
www.detodoprogramacion.org

- Los diseños que hacen un uso amplio de **Composite** y **Decorator** a menudo pueden beneficiarse del uso del **Prototype**. Aplicar el patrón te permite clonar estructuras complejas en lugar de reconstruirlas desde cero.

- **Decorator** te permite cambiar la piel de un objeto, mientras que **Strategy** te permite cambiar sus entrañas.

- **Decorator** y **Proxy** tienen estructuras similares, pero propósitos muy distintos. Ambos patrones se basan en el principio de composición, por el que un objeto debe delegar parte del trabajo a otro. La diferencia es que, normalmente, un *Proxy* gestiona el ciclo de vida de su objeto de servicio por su cuenta, mientras que la composición de los *Decoradores* siempre está controlada por el cliente.

www.detodoprogramacion.org



FACADE

También llamado: Fachada

Facade es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.

☹ Problema

Imagina que debes lograr que tu código trabaje con un amplio grupo de objetos que pertenecen a una sofisticada biblioteca o *framework*. Normalmente, debes inicializar todos esos objetos, llevar un registro de las dependencias, ejecutar los métodos en el orden correcto y así sucesivamente.

Como resultado, la lógica de negocio de tus clases se vería estrechamente acoplada a los detalles de implementación de las clases de terceros, haciéndola difícil de comprender y mantener.

😊 Solución

Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles. Una fachada puede proporcionar una funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente importantes para los clientes.

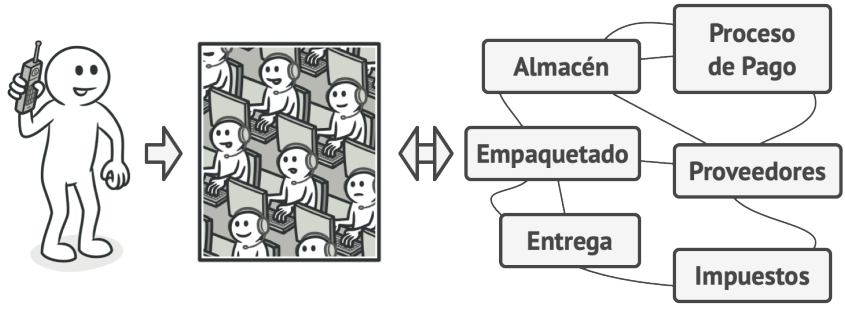
Tener una fachada resulta útil cuando tienes que integrar tu aplicación con una biblioteca sofisticada con decenas de funciones, de la cual sólo necesitas una pequeña parte.

Por ejemplo, una aplicación que sube breves vídeos divertidos de gatos a las redes sociales, podría potencialmente utilizar una biblioteca de conversión de vídeo profesional. Sin embar-

www.detodoprogramacion.org

go, lo único que necesita en realidad es una clase con el método simple `codificar(nombreDelArchivo, formato)`. Una vez que crees dicha clase y la conectes con la biblioteca de conversión de vídeo, tendrás tu primera fachada.

Analogía en el mundo real

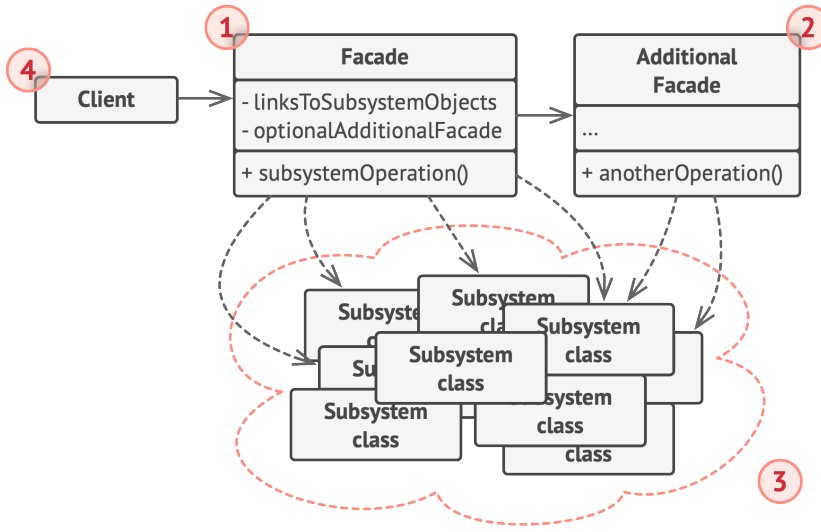


Haciendo pedidos por teléfono.

Cuando llamas a una tienda para hacer un pedido por teléfono, un operador es tu fachada a todos los servicios y departamentos de la tienda. El operador te proporciona una sencilla interfaz de voz al sistema de pedidos, pasarelas de pago y varios servicios de entrega.

www.detodoprogramacion.org

Estructura



1. El patrón **Facade** proporciona un práctico acceso a una parte específica de la funcionalidad del subsistema. Sabe a dónde dirigir la petición del cliente y cómo operar todas las partes móviles.

2. Puede crearse una clase **Fachada Adicional** para evitar contaminar una única fachada con funciones no relacionadas que podrían convertirla en otra estructura compleja. Las fachadas adicionales pueden utilizarse por clientes y por otras fachadas.

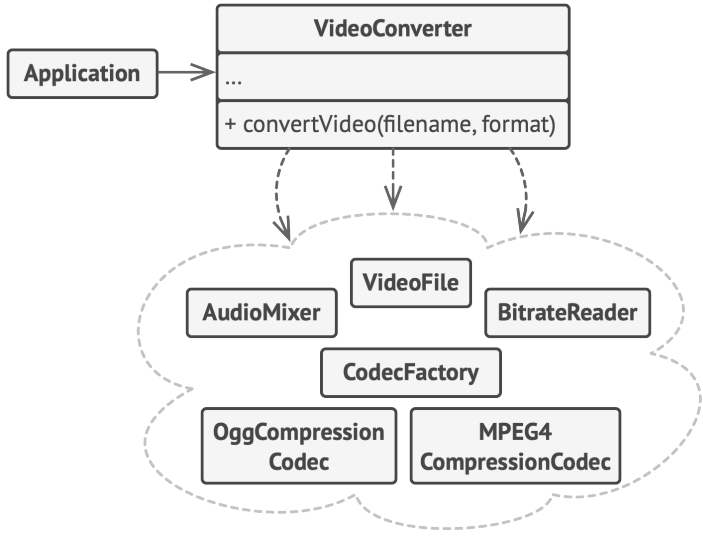
3. El **Subsistema Complejo** consiste en decenas de objetos diversos. Para lograr que todos hagan algo significativo, debes profundizar en los detalles de implementación del subsistema, que pueden incluir inicializar objetos en el orden correcto y suministrarles datos en el formato adecuado.

Las clases del subsistema no conocen la existencia de la fachada. Operan dentro del sistema y trabajan entre sí directamente.

4. El **Cliente** utiliza la fachada en lugar de invocar directamente los objetos del subsistema.

Pseudocódigo

En este ejemplo, el patrón **Facade** simplifica la interacción con un framework complejo de conversión de vídeo.



Un ejemplo de aislamiento de múltiples dependencias dentro de una única clase fachada.

En lugar de hacer que tu código trabaje con decenas de las clases del framework directamente, creas una clase fachada que encapsula esa funcionalidad y la esconde del resto del códi-

www.detodoprogramacion.org

go. Esta estructura también te ayuda a minimizar el esfuerzo de actualizar a futuras versiones del framework o de sustituirlo por otro. Lo único que tendrías que cambiar en la aplicación es la implementación de los métodos de la fachada.

```
// Estas son algunas de las clases de un framework de conversión
// de video de un tercero. No controlamos ese código, por lo que
// no podemos simplificarlo.

class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...


// Creamos una clase fachada para esconder la complejidad del
// framework tras una interfaz simple. Es una solución de
// equilibrio entre funcionalidad y simplicidad.
```

```
class VideoConverter is
  method convert(filename, format):File is
    file = new VideoFile(filename)
    sourceCodec = new CodecFactory.extract(file)
    if (format == "mp4")
      destinationCodec = new MPEG4CompressionCodec()
    else
      destinationCodec = new OggCompressionCodec()
    buffer = BitrateReader.read(filename, sourceCodec)
    result = BitrateReader.convert(buffer, destinationCodec)
    result = (new AudioMixer()).fix(result)
    return new File(result)

// Las clases Application no dependen de un millón de clases
// proporcionadas por el complejo framework. Además, si decides
// cambiar los frameworks, sólo tendrás de volver a escribir la
// clase fachada.
class Application is
  method main() is
    convertor = new VideoConverter()
    mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
    mp4.save()
```

Aplicabilidad

Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.

A menudo los subsistemas se vuelven más complejos con el tiempo. Incluso la aplicación de patrones de diseño suele conducir a la creación de un mayor número de clases. Un subsi-

stema puede hacerse más flexible y más fácil de reutilizar en varios contextos, pero la cantidad de código de configuración que exige de un cliente, crece aún más. El patrón Facade intenta solucionar este problema proporcionando un atajo a las funciones más utilizadas del subsistema que mejor encajan con los requisitos del cliente.

Utiliza el patrón Facade cuando quieras estructurar un subsistema en capas.

Crea fachadas para definir puntos de entrada a cada nivel de un subsistema. Puedes reducir el acoplamiento entre varios subsistemas exigiéndoles que se comuniquen únicamente mediante fachadas.

Por ejemplo, regresemos a nuestro framework de conversión de vídeo. Puede dividirse en dos capas: la relacionada con el vídeo y la relacionada con el audio. Puedes crear una fachada para cada capa y hacer que las clases de cada una de ellas se comuniquen entre sí a través de esas fachadas. Este procedimiento es bastante similar al patrón **Mediator**.

Cómo implementarlo

1. Comprueba si es posible proporcionar una interfaz más simple que la que está proporcionando un subsistema existente. Estás bien encaminado si esta interfaz hace que el código cliente sea independiente de muchas de las clases del subsistema.

www.detodoprogramacion.org

2. Declara e implementa esta interfaz en una nueva clase fachada. La fachada deberá redireccionar las llamadas desde el código cliente a los objetos adecuados del subsistema. La fachada deberá ser responsable de inicializar el subsistema y gestionar su ciclo de vida, a no ser que el código cliente ya lo haga.

3. Para aprovechar el patrón al máximo, haz que todo el código cliente se comunique con el subsistema únicamente a través de la fachada. Ahora el código cliente está protegido de cualquier cambio en el código del subsistema. Por ejemplo, cuando se actualice un subsistema a una nueva versión, sólo tendrás que modificar el código de la fachada.

4. Si la fachada se vuelve **demasiado grande**, piensa en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.

Pros y contras

✓ Puedes aislar tu código de la complejidad de un subsistema.

✕ Una fachada puede convertirse en **un objeto todopoderoso** acoplado a todas las clases de una aplicación.

Relaciones con otros patrones

- **Facade** define una nueva interfaz para objetos existentes, mientras que **Adapter** intenta hacer que la interfaz existente sea

www.detodoprogramacion.org

utilizable. Normalmente *Adapter* sólo envuelve un objeto, mientras que *Facade* trabaja con todo un subsistema de objetos.

- **Abstract Factory** puede servir como alternativa a **Facade** cuando tan solo deseas esconder la forma en que se crean los objetos del subsistema a partir del código cliente.

- **Flyweight** muestra cómo crear muchos pequeños objetos, mientras que **Facade** muestra cómo crear un único objeto que represente un subsistema completo.

- **Facade** y **Mediator** tienen trabajos similares: ambos intentan organizar la colaboración entre muchas clases estrechamente acopladas.

    - *Facade* define una interfaz simplificada a un subsistema de objetos, pero no introduce ninguna nueva funcionalidad. El propio subsistema no conoce la fachada. Los objetos del subsistema pueden comunicarse directamente.
    - *Mediator* centraliza la comunicación entre componentes del sistema. Los componentes conocen únicamente el objeto mediador y no se comunican directamente.

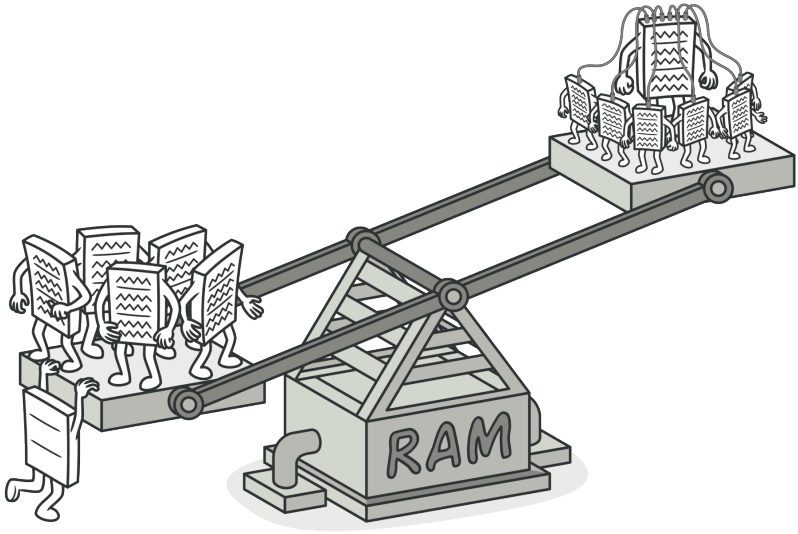
- Una clase **fachada** a menudo puede transformarse en una **Singleton**, ya que un único objeto fachada es suficiente en la mayoría de los casos.

- **Facade** es similar a **Proxy** en el sentido de que ambos pueden almacenar temporalmente una entidad compleja e inicializar-

www.detodoprogramacion.org

la por su cuenta. Al contrario que *Facade*, *Proxy* tiene la misma interfaz que su objeto de servicio, lo que hace que sean intercambiables.





FLYWEIGHT

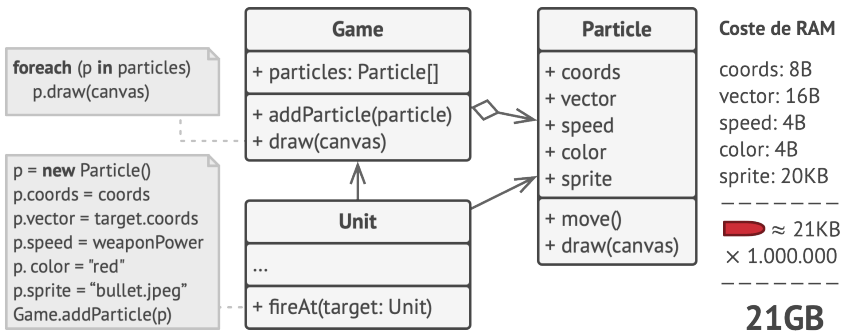
También llamado: Peso mosca, Peso ligero, Cache

Flyweight es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

☹ Problema

Para divertirte un poco después de largas horas de trabajo, decides crear un sencillo videojuego en el que los jugadores se tienen que mover por un mapa disparándose entre sí. Decides implementar un sistema de partículas realistas que lo distinga de otros juegos. Grandes cantidades de balas, misiles y metralla de las explosiones volarán por todo el mapa, ofreciendo una apasionante experiencia al jugador.

Al terminarlo, subes el último cambio, compilas el juego y se lo envias a un amigo para una partida de prueba. Aunque el juego funcionaba sin problemas en tu máquina, tu amigo no logró jugar durante mucho tiempo. En su computadora el juego se paraba a los pocos minutos de empezar. Tras dedicar varias horas a revisar los registros de depuración, descubres que el juego se paraba debido a una cantidad insuficiente de RAM. Resulta que el equipo de tu amigo es mucho menos potente que tu computadora, y esa es la razón por la que el problema surgió tan rápido en su máquina.

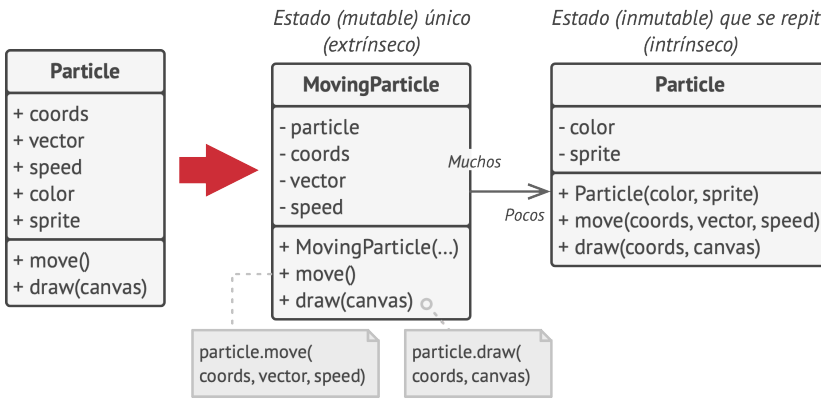


El problema estaba relacionado con tu sistema de partículas. Cada partícula, como una bala, un misil o un trozo de metralla, estaba representada por un objeto separado que contenía gran cantidad de datos. En cierto momento, cuando la masacre alcanzaba su punto culminante en la pantalla del jugador, las partículas recién creadas ya no cabían en el resto de RAM, provocando que el programa fallara.

😊 Solución

Descargado de: www.detodoprogramacion.org

Observando más atentamente la clase `Partícula`, puede ser que te hayas dado cuenta de que los campos de color y *sprite* consumen mucha más memoria que otros campos. Lo que es peor, esos dos campos almacenan información casi idéntica de todas las partículas. Por ejemplo, todas las balas tienen el mismo color y sprite.

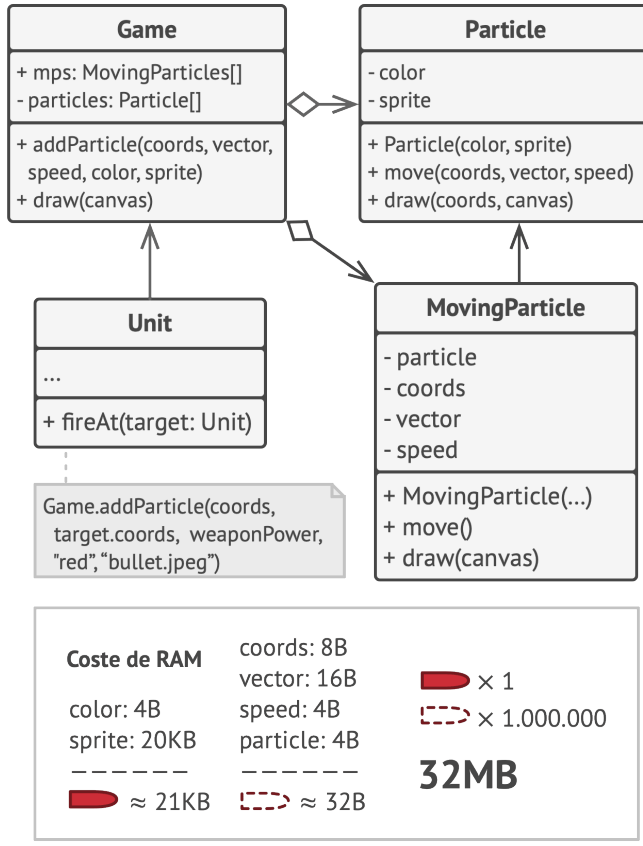


Otras partes del estado de una partícula, como las coordenadas, vector de movimiento y velocidad, son únicas en cada partícula. Después de todo, los valores de estos campos cambian

www.detodoprogramacion.org

a lo largo del tiempo. Estos datos representan el contexto siempre cambiante en el que existe la partícula, mientras que el color y el sprite se mantienen constantes.

Esta información constante de un objeto suele denominarse su *estado intrínseco*. Existe dentro del objeto y otros objetos únicamente pueden leerla, no cambiarla. El resto del estado del objeto, a menudo alterado "desde el exterior" por otros objetos, se denomina el *estado extrínseco*.



El patrón Flyweight sugiere que dejemos de almacenar el estado extrínseco dentro del objeto. En lugar de eso, debes pasar este estado a métodos específicos que dependen de él. Tan solo el estado intrínseco se mantiene dentro del objeto, permitiendo que lo reutilices en distintos contextos. Como resultado, necesitarás menos de estos objetos, ya que sólo se diferencian en el estado intrínseco, que cuenta con muchas menos variaciones que el extrínseco.

Regresemos a nuestro juego. Dando por hecho que hemos extraído el estado extrínseco de la clase de nuestra partícula, únicamente tres objetos diferentes serán suficientes para representar todas las partículas del juego: una bala, un misil y un trozo de metralla. Como probablemente habrás adivinado, un objeto que sólo almacena el estado intrínseco se denomina *Flyweight* (peso mosca).

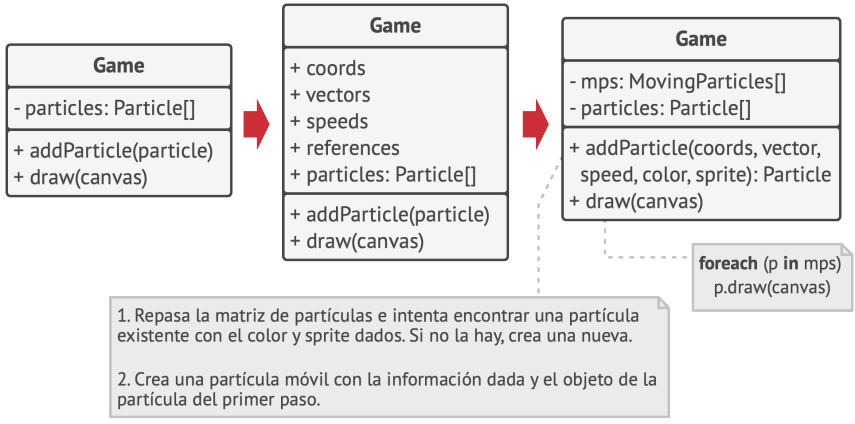
Almacenamiento del estado extrínseco

¿A dónde se mueve el estado extrínseco? Alguna clase tendrá que almacenarlo, ¿verdad? En la mayoría de los casos, se mueve al objeto contenedor, que reúne objetos antes de que apliquemos el patrón.

En nuestro caso, se trata del objeto principal `Juego`, que almacena todas las partículas en su campo `partículas`. Para mover el estado extrínseco a esta clase, debes crear varios campos matriz para almacenar coordenadas, vectores y velocidades de cada partícula individual. Pero eso no es todo. Necesitas

www.detodoprogramacion.org

otra matriz para almacenar referencias a un objeto *flyweight* específico que represente una partícula. Estas matrices deben estar sincronizadas para que puedas acceder a toda la información de una partícula utilizando el mismo índice.



Una solución más elegante sería crear una clase de contexto separada que almacene el estado extrínseco junto con la referencia al objeto flyweight. Esta solución únicamente exigiría tener una matriz en la clase contenedora.

¡Espera un momento! ¿No deberíamos tener tantos de estos objetos contextuales como teníamos al principio? Técnicamente, sí. Pero el caso es que estos objetos son mucho más pequeños que antes. Los campos que consumen más memoria se han movido a unos pocos objetos flyweight. Ahora, cientos de pequeños objetos contextuales pueden reutilizar un único objeto flyweight pesado, en lugar de almacenar cientos de copias de sus datos.

www.detodoprogramacion.org

Flyweight y la inmutabilidad

Debido a que el mismo objeto flyweight puede utilizarse en distintos contextos, debes asegurarte de que su estado no se pueda modificar. Un objeto flyweight debe inicializar su estado una sola vez a través de parámetros del constructor. No debe exponer ningún método *set* (modificador) o campo público a otros objetos.

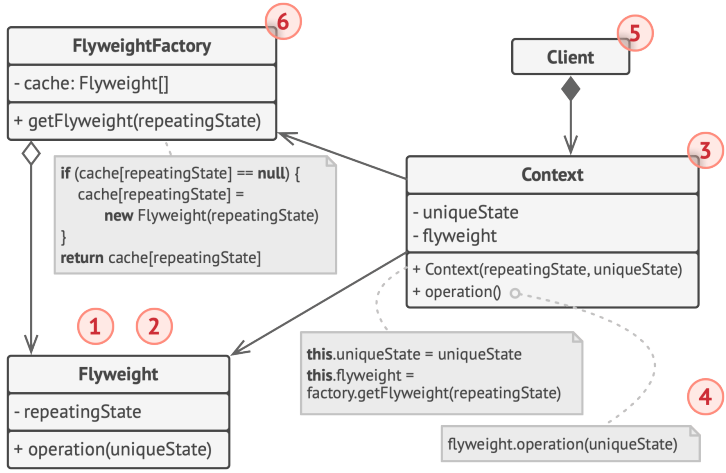
Fábrica flyweight

Para un acceso más cómodo a varios objetos flyweight, puedes crear un método fábrica que gestione un grupo de objetos flyweight existentes. El método acepta el estado intrínseco del flyweight deseado por un cliente, busca un objeto flyweight existente que coincida con este estado y lo devuelve si lo encuentra. Si no, crea un nuevo objeto flyweight y lo añade al grupo.

Existen muchas opciones para colocar este método. El lugar más obvio es un contenedor flyweight. Alternativamente, podrías crea un nueva clase fábrica y hacer estático el método fábrica para colocarlo dentro de una clase flyweight real.

www.detodoprogramacion.org

Estructura



1. El patrón Flyweight es simplemente una optimización. Antes de aplicarlo, asegúrate de que tu programa tenga un problema de consumo de RAM provocado por tener una gran cantidad de objetos similares en la memoria al mismo tiempo. Asegúrate de que este problema no se pueda solucionar de otra forma sensata.

2. La clase **Flyweight** contiene la parte del estado del objeto original que pueden compartir varios objetos. El mismo objeto flyweight puede utilizarse en muchos contextos diferentes. El estado almacenado dentro de un objeto flyweight se denomina *intrínseco*, mientras que al que se pasa a sus métodos se le llama *extrínseco*.

3. La clase **Contexto** contiene el estado extrínseco, único en todos los objetos originales. Cuando un contexto se empareja

con uno de los objetos flyweight, representa el estado completo del objeto original.

4. Normalmente, el comportamiento del objeto original permanece en la clase flyweight. En este caso, quien invoque un método del objeto flyweight debe también pasar las partes adecuadas del estado extrínseco dentro de los parámetros del método. Por otra parte, el comportamiento se puede mover a la clase de contexto, que utilizará el objeto flyweight vinculado como mero objeto de datos.

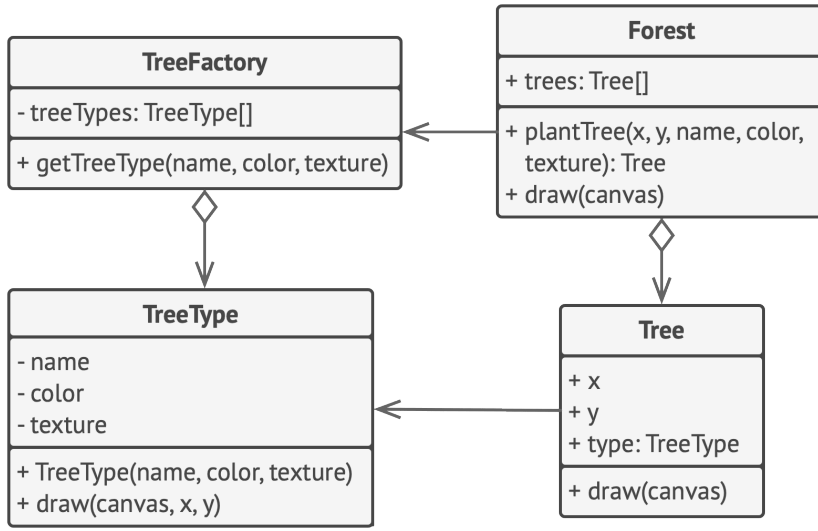
5. El **Cliente** calcula o almacena el estado extrínseco de los objetos flyweight. Desde la perspectiva del cliente, un flyweight es un objeto plantilla que puede configurarse durante el tiempo de ejecución pasando información contextual dentro de los parámetros de sus métodos.

6. La **Fábrica flyweight** gestiona un grupo de objetos flyweight existentes. Con la fábrica, los clientes no crean objetos flyweight directamente. En lugar de eso, invocan a la fábrica, pasándole partes del estado intrínseco del objeto flyweight deseado. La fábrica revisa objetos flyweight creados previamente y devuelve uno existente que coincida con los criterios de búsqueda, o bien crea uno nuevo si no encuentra nada.

www.detodoprogramacion.org

Pseudocódigo

En este ejemplo, el patrón **Flyweight** ayuda a reducir el uso de memoria a la hora de representar millones de objetos de árbol en un lienzo.



El patrón extrae el estado intrínseco repetido de una clase principal `Árbol` y la mueve dentro de la clase flyweight `TipodeÁrbol`.

Ahora, en lugar de almacenar la misma información en varios objetos, se mantiene en unos pocos objetos flyweight vinculados a los objetos de `Árbol` adecuados que actúan como contexto. El código cliente crea nuevos objetos árbol utilizando la fábrica flyweight, que encapsula la complejidad de buscar el objeto adecuado y reutilizarlo si es necesario.

```
// La clase flyweight contiene una parte del estado de un árbol.
// Estos campos almacenan valores que son únicos para cada árbol
// en particular. Por ejemplo, aquí no encontrarás las
// coordenadas del árbol. Pero la textura y los colores que
// comparten muchos árboles sí están aquí. Ya que esta cantidad
// de datos suele ser GRANDE, dedicarás mucha memoria a
// mantenerla en cada objeto árbol. En lugar de eso, podemos
// extraer la textura, el color y otros datos repetidos y
// colocarlos en un objeto independiente que muchos objetos
// individuales del árbol pueden referenciar.
class TreeType is
  field name
  field color
  field texture
  constructor TreeType(name, color, texture) { ... }
  method draw(canvas, x, y) is
    // 1. Crea un mapa de bits de un tipo, color y textura
    // concretos.
    // 2. Dibuja el mapa de bits en el lienzo con las
    // coordenadas X y Y.


// La fábrica flyweight decide si reutiliza el flyweight
// existente o si crea un nuevo objeto.
class TreeFactory is
  static field treeTypes: collection of tree types
  static method getTreeType(name, color, texture) is
    type = treeTypes.find(name, color, texture)
    if (type == null)
      type = new TreeType(name, color, texture)
      treeTypes.add(type)
    return type
```

```
33
34  // El objeto contextual contiene la parte extrínseca del estado
35  // del árbol. Una aplicación puede crear millones de ellas, ya
36  // que son muy pequeñas: dos coordenadas en números enteros y un
37  // campo de referencia.
38  class Tree is
39    field x,y
40    field type: TreeType
41    constructor Tree(x, y, type) { ... }
42    method draw(canvas) is
43      type.draw(canvas, this.x, this.y)
44
45  // Las clases Tree y Forest son los clientes de flyweight.
46  // Puedes fusionarlas si no tienes la intención de desarrollar
47  // más la clase Tree.
48  class Forest is
49    field trees: collection of Trees
50
51    method plantTree(x, y, name, color, texture) is
52      type = TreeFactory.getTreeType(name, color, texture)
53      tree = new Tree(x, y, type)
54      trees.add(tree)
55
56    method draw(canvas) is
57      foreach (tree in trees) do
58        tree.draw(canvas)
```

Aplicabilidad

Utiliza el patrón Flyweight únicamente cuando tu programa deba soportar una enorme cantidad de objetos que apenas quepan en la RAM disponible.

La ventaja de aplicar el patrón depende en gran medida de cómo y dónde se utiliza. Resulta más útil cuando:

- la aplicación necesita generar una cantidad enorme de objetos similares
- esto consume toda la RAM disponible de un dispositivo objetivo
- los objetos contienen estados duplicados que se pueden extraer y compartir entre varios objetos

Cómo implementarlo

1. Divide los campos de una clase que se convertirá en flyweight en dos partes:
    - el estado intrínseco: los campos que contienen información invariable duplicada a través de varios objetos
    - el estado extrínseco: los campos que contienen información contextual única de cada objeto

2. Deja los campos que representan el estado intrínseco en la clase, pero asegúrate de que sean inmutables. Deben llevar sus valores iniciales únicamente dentro del constructor.

3. Repasa los métodos que utilizan campos del estado extrínseco. Para cada campo utilizado en el método, introduce un nuevo parámetro y utilízalo en lugar del campo.

4. Opcionalmente, crea una clase fábrica para gestionar el grupo de objetos flyweight, buscando uno existente antes de crear uno nuevo. Una vez que la fábrica esté en su sitio, los clientes sólo deberán solicitar objetos flyweight a través de ella. Deberán describir el flyweight deseado pasando su estado intrínseco a la fábrica.

5. El cliente deberá almacenar o calcular valores del estado extrínseco (contexto) para poder invocar métodos de objetos flyweight. Por comodidad, el estado extrínseco puede moverse a una clase contexto separada junto con el campo referenciador del flyweight.

Pros y contras

✓ Puedes ahorrar mucha RAM, siempre que tu programa tenga toneladas de objetos similares.

✕ Puede que estés cambiando RAM por ciclos CPU cuando deba calcularse de nuevo parte de la información de contexto cada vez que alguien invoque un método flyweight.

www.detodoprogramacion.org

✕ El código se complica mucho. Los nuevos miembros del equipo siempre estarán preguntándose por qué el estado de una entidad se separó de tal manera.

⇄ Relaciones con otros patrones

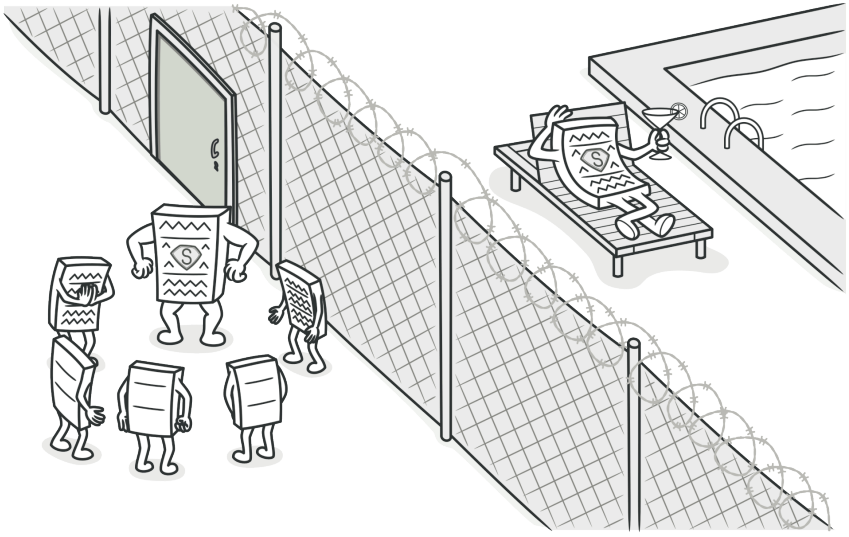
- Puedes implementar nodos de hoja compartidos del árbol **Composite** como **Flyweights** para ahorrar memoria RAM.

- **Flyweight** muestra cómo crear muchos pequeños objetos, mientras que **Facade** muestra cómo crear un único objeto que represente un subsistema completo.

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

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

www.detodoprogramacion.org

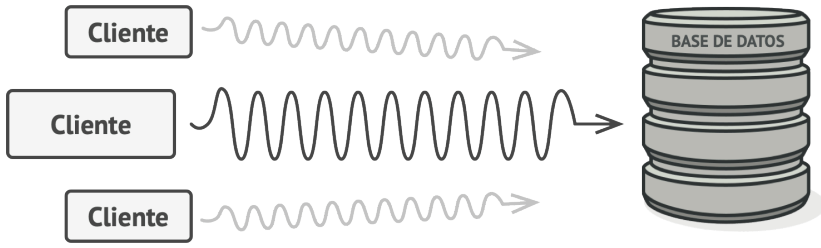


PROXY

Proxy es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original.

☹ Problema

¿Por qué querrías controlar el acceso a un objeto? Imagina que tienes un objeto enorme que consume una gran cantidad de recursos del sistema. Lo necesitas de vez en cuando, pero no siempre.



Las consultas a las bases de datos pueden ser muy lentas.

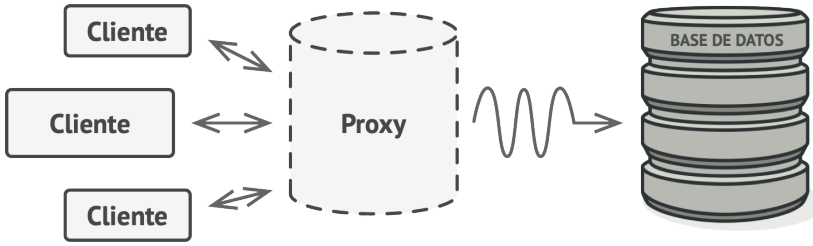
Puedes llevar a cabo una implementación diferida, es decir, crear este objeto sólo cuando sea realmente necesario. Todos los clientes del objeto tendrán que ejecutar algún código de inicialización diferida. Lamentablemente, esto seguramente generará una gran cantidad de código duplicado.

En un mundo ideal, querríamos meter este código directamente dentro de la clase de nuestro objeto, pero eso no siempre es posible. Por ejemplo, la clase puede ser parte de una biblioteca cerrada de un tercero.

www.detodoprogramacion.org

😊 Solución

El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

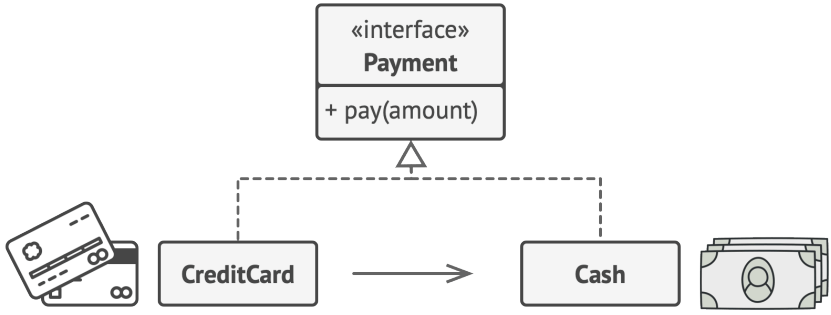


El proxy se camufla como objeto de la base de datos. Puede gestionar la inicialización diferida y el caché de resultados sin que el cliente o el objeto real de la base de datos lo sepan.

Pero, ¿cuál es la ventaja? Si necesitas ejecutar algo antes o después de la lógica primaria de la clase, el proxy te permite hacerlo sin cambiar esa clase. Ya que el proxy implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de servicio real.

www.detodoprogramacion.org

Analogía en el mundo real

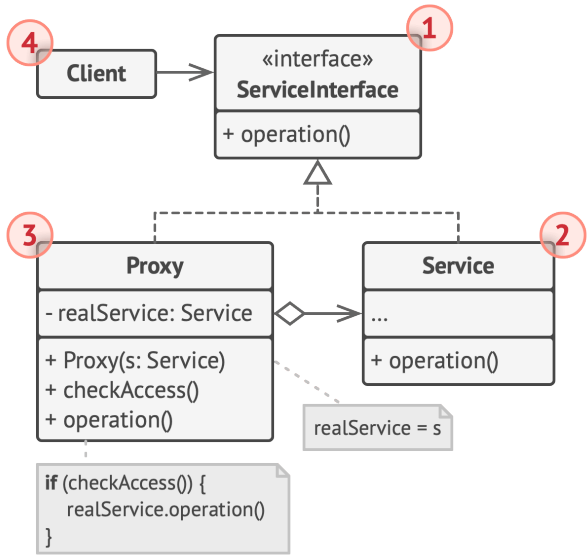


Las tarjetas de crédito pueden utilizarse para realizar pagos tanto como el efectivo.

Una tarjeta de crédito es un proxy de una cuenta bancaria, que, a su vez, es un proxy de un manojo de billetes. Ambos implementan la misma interfaz, por lo que pueden utilizarse para realizar un pago. El consumidor se siente genial porque no necesita llevar un montón de efectivo encima. El dueño de la tienda también está contento porque los ingresos de la transacción se añaden electrónicamente a la cuenta bancaria de la tienda sin el riesgo de perder el depósito o sufrir un robo de camino al banco.

www.detodoprogramacion.org

Estructura



1. La **Interfaz de Servicio** declara la interfaz del Servicio. El proxy debe seguir esta interfaz para poder camuflarse como objeto de servicio.

2. **Servicio** es una clase que proporciona una lógica de negocio útil.

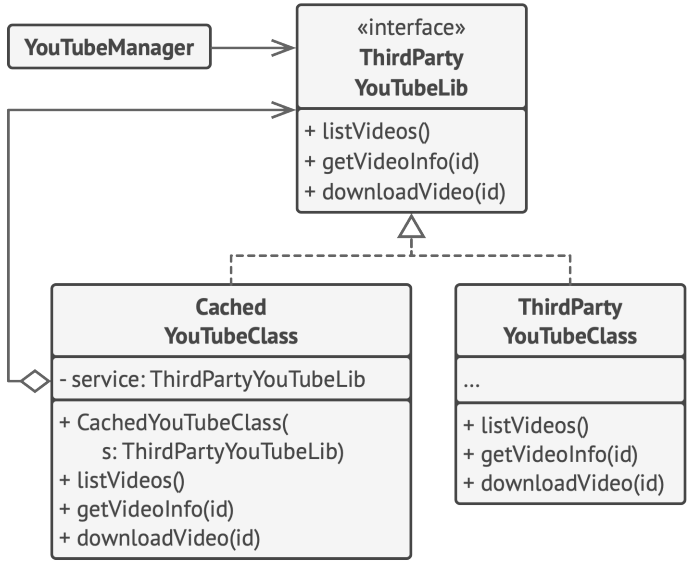
3. La clase **Proxy** tiene un campo de referencia que apunta a un objeto de servicio. Cuando el proxy finaliza su procesamiento (por ejemplo, inicialización diferida, registro, control de acceso, almacenamiento en caché, etc.), pasa la solicitud al objeto de servicio.

Normalmente los proxies gestionan el ciclo de vida completo de sus objetos de servicio.

4. El **Cliente** debe funcionar con servicios y proxies a través de la misma interfaz. De este modo puedes pasar un proxy a cualquier código que espere un objeto de servicio.

Pseudocódigo

Este ejemplo ilustra cómo el patrón **Proxy** puede ayudar a introducir la inicialización diferida y el almacenamiento en caché a una biblioteca de integración de YouTube de un tercero.



Resultados del almacenamiento en caché de un servicio con un proxy.

La biblioteca nos proporciona la clase de descarga de videos. Sin embargo, es muy ineficiente. Si la aplicación cliente solici-

www.detodoprogramacion.org

ta el mismo video muchas veces, la biblioteca lo descarga una y otra vez, en lugar de guardarlo en caché y reutilizar el primer archivo descargado.

La clase proxy implementa la misma interfaz que el descargador original y le delega todo el trabajo. No obstante, mantiene un seguimiento de los archivos descargados y devuelve los resultados en caché cuando la aplicación solicita el mismo video varias veces.

```
// La interfaz de un servicio remoto.
interface ThirdPartyYouTubeLib is
  method listVideos()
  method getVideoInfo(id)
  method downloadVideo(id)

// La implementación concreta de un conector de servicio. Los
// métodos de esta clase pueden solicitar información a YouTube.
// La velocidad de la solicitud depende de la conexión a
// internet del usuario y de YouTube. La aplicación se
// ralentizará si se lanzan muchas solicitudes al mismo tiempo,
// incluso aunque todas soliciten la misma información.
class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
  method listVideos() is
    // Envía una solicitud API a YouTube.

  method getVideoInfo(id) is
    // Obtiene metadatos de algún video.

  method downloadVideo(id) is
```

```
    // Descarga un archivo de video de YouTube.

// Para ahorrar ancho de banda, podemos guardar en caché
// resultados de la solicitud durante algún tiempo, pero se
// puede colocar este código directamente dentro de la clase de
// servicio. Por ejemplo, puede haberse proporcionado como parte
// de la biblioteca de un tercero y/o definido como `final`. Por
// eso colocamos el código de almacenamiento en caché dentro de
// una nueva clase proxy que implementa la misma interfaz que la
// clase servicio. Delega al objeto de servicio únicamente
// cuando deben enviarse las solicitudes reales.
class CachedYouTubeClass implements ThirdPartyYouTubeLib is
  private field service: ThirdPartyYouTubeLib
  private field listCache, videoCache
  field needReset

  constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is
    this.service = service

  method listVideos() is
    if (listCache == null || needReset)
      listCache = service.listVideos()
    return listCache

  method getVideoInfo(id) is
    if (videoCache == null || needReset)
      videoCache = service.getVideoInfo(id)
    return videoCache

  method downloadVideo(id) is
    if (!downloadExists(id) || needReset)
      service.downloadVideo(id)
```

```
53
54  // La clase GUI, que solía trabajar directamente con un objeto
55  // de servicio, permanece sin cambios siempre y cuando trabaje
56  // con el objeto de servicio a través de una interfaz. Podemos
57  // pasar sin riesgo un objeto proxy en lugar de un objeto de
58  // servicio real, ya que ambos implementan la misma interfaz.
59  class YouTubeManager is
60    protected field service: ThirdPartyYouTubeLib
61
62    constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
63      this.service = service
64
65    method renderVideoPage(id) is
66      info = service.getVideoInfo(id)
67      // Representa la página del video.
68
69    method renderListPanel() is
70      list = service.listVideos()
71      // Representa la lista de miniaturas de los videos.
72
73    method reactOnUserInput() is
74      renderVideoPage()
75      renderListPanel()
76
77  // La aplicación puede configurar proxies sobre la marcha.
78  class Application is
79    method init() is
80      aYouTubeService = new ThirdPartyYouTubeClass()
81      aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
82      manager = new YouTubeManager(aYouTubeProxy)
83      manager.reactOnUserInput()
```

Aplicabilidad

Hay decenas de formas de utilizar el patrón Proxy. Repasemos los usos más populares.

Inicialización diferida (proxy virtual). Es cuando tienes un objeto de servicio muy pesado que utiliza muchos recursos del sistema al estar siempre funcionando, aunque solo lo necesites de vez en cuando.

En lugar de crear el objeto cuando se lanza la aplicación, puedes retrasar la inicialización del objeto a un momento en que sea realmente necesario.

Control de acceso (proxy de protección). Es cuando quieres que únicamente clientes específicos sean capaces de utilizar el objeto de servicio, por ejemplo, cuando tus objetos son partes fundamentales de un sistema operativo y los clientes son varias aplicaciones lanzadas (incluyendo maliciosas).

El proxy puede pasar la solicitud al objeto de servicio tan sólo si las credenciales del cliente cumplen ciertos criterios.

Ejecución local de un servicio remoto (proxy remoto). Es cuando el objeto de servicio se ubica en un servidor remoto.

En este caso, el proxy pasa la solicitud del cliente por la red, gestionando todos los detalles desagradables de trabajar con la red.

Solicitudes de registro (proxy de registro). Es cuando quieres mantener un historial de solicitudes al objeto de servicio.

El proxy puede registrar cada solicitud antes de pasarla al servicio.

Resultados de solicitudes en caché (proxy de caché). Es cuando necesitas guardar en caché resultados de solicitudes de clientes y gestionar el ciclo de vida de ese caché, especialmente si los resultados son muchos.

El proxy puede implementar el caché para solicitudes recurrentes que siempre dan los mismos resultados. El proxy puede utilizar los parámetros de las solicitudes como claves de caché.

Referencia inteligente. Es cuando debes ser capaz de desechar un objeto pesado una vez que no haya clientes que lo utilicen.

El proxy puede rastrear los clientes que obtuvieron una referencia del objeto de servicio o sus resultados. De vez en cuando, el proxy puede recorrer los clientes y comprobar si siguen activos. Si la lista del cliente se vacía, el proxy puede desechar el objeto de servicio y liberar los recursos subyacentes del sistema.

El proxy también puede rastrear si el cliente ha modificado el objeto de servicio. Después, los objetos sin cambios pueden ser reutilizados por otros clientes.

www.detodoprogramacion.org

Cómo implementarlo

1. Si no hay una interfaz de servicio preexistente, crea una para que los objetos de proxy y de servicio sean intercambiables. No siempre resulta posible extraer la interfaz de la clase servicio, porque tienes que cambiar todos los clientes del servicio para utilizar esa interfaz. El plan B consiste en convertir el proxy en una subclase de la clase servicio, de forma que herede la interfaz del servicio.

2. Crea la clase proxy. Debe tener un campo para almacenar una referencia al servicio. Normalmente los proxies crean y gestionan el ciclo de vida completo de sus servicios. En raras ocasiones, el cliente pasa un servicio al proxy a través de un constructor.

3. Implementa los métodos del proxy según sus propósitos. En la mayoría de los casos, después de hacer cierta labor, el proxy debería delegar el trabajo a un objeto de servicio.

4. Considera introducir un método de creación que decida si el cliente obtiene un proxy o un servicio real. Puede tratarse de un simple método estático en la clase proxy o de todo un método de fábrica.

5. Considera implementar la inicialización diferida para el objeto de servicio.

www.detodoprogramacion.org

Pros y contras

✓ Puedes controlar el objeto de servicio sin que los clientes lo sepan.

✓ Puedes gestionar el ciclo de vida del objeto de servicio cuando a los clientes no les importa.

✓ El proxy funciona incluso si el objeto de servicio no está listo o no está disponible.

✓ *Principio de abierto/cerrado*. Puedes introducir nuevos proxies sin cambiar el servicio o los clientes.

✕ El código puede complicarse ya que debes introducir gran cantidad de clases nuevas.

✕ La respuesta del servicio puede retrasarse.

Relaciones con otros patrones

- **Adapter** proporciona una interfaz diferente al objeto envuelto, **Proxy** le proporciona la misma interfaz y **Decorator** le proporciona una interfaz mejorada.

- **Facade** es similar a **Proxy** en el sentido de que ambos pueden almacenar temporalmente una entidad compleja e inicializarla por su cuenta. Al contrario que *Facade*, *Proxy* tiene la misma interfaz que su objeto de servicio, lo que hace que sean intercambiables.

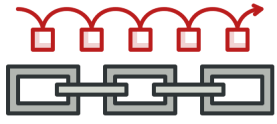
www.detodoprogramacion.org

- **Decorator** y **Proxy** tienen estructuras similares, pero propósitos muy distintos. Ambos patrones se basan en el principio de composición, por el que un objeto debe delegar parte del trabajo a otro. La diferencia es que, normalmente, un *Proxy* gestiona el ciclo de vida de su objeto de servicio por su cuenta, mientras que la composición de los *Decoradores* siempre está controlada por el cliente.

www.detodoprogramacion.org

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.



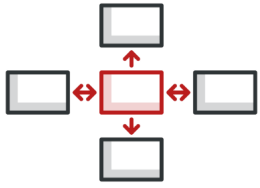
Command

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.



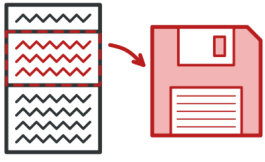
Iterator

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



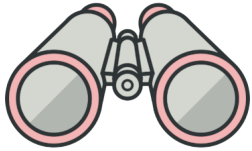
Mediator

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.



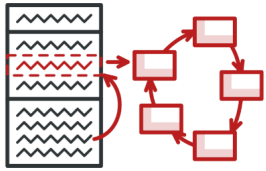
Memento

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



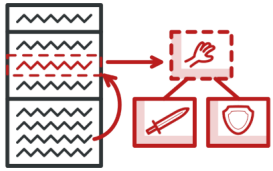
Observer

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.



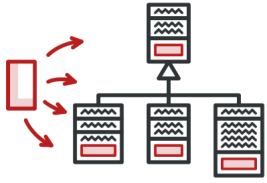
Strategy

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



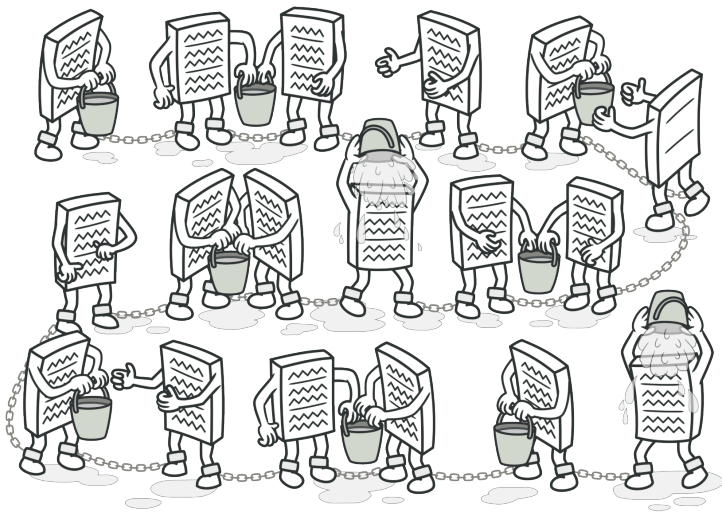
Template Method

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

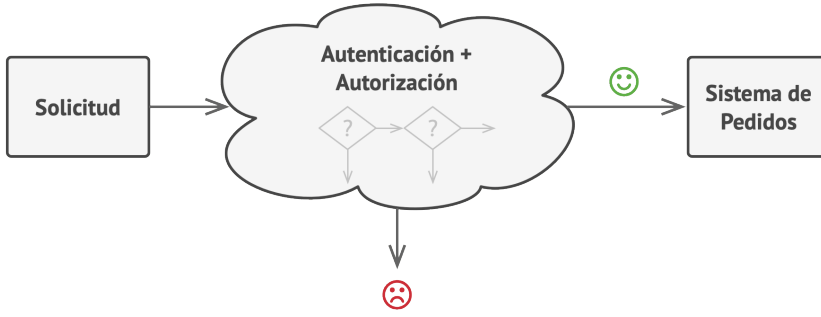
También llamado: Cadena de responsabilidad, CoR, Chain of Command

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.



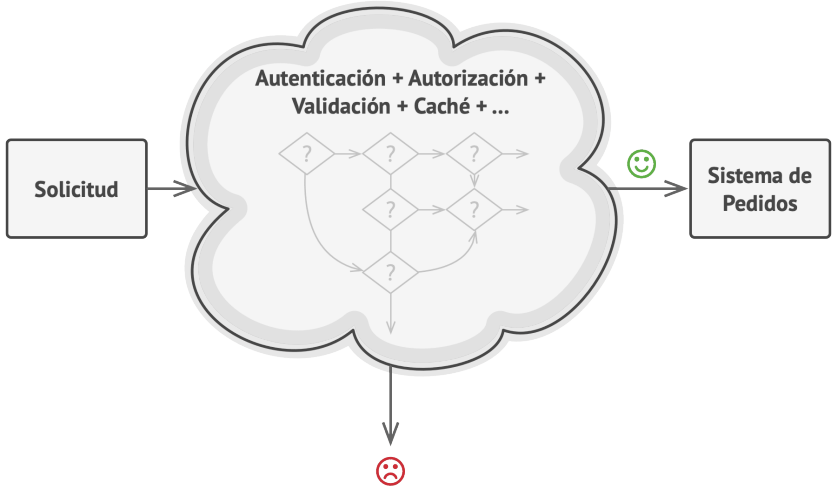
La solicitud debe pasar una serie de comprobaciones antes de que el propio sistema de pedidos pueda gestionarla.

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

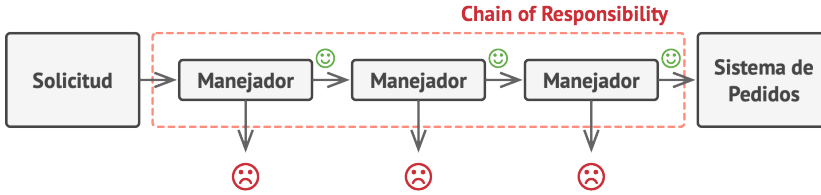
Al igual que muchos otros patrones de diseño de comportamiento, el **Chain of Responsibility** se basa en transformar comportamientos particulares en objetos autónomos llamados *manejadores*. En nuestro caso, cada comprobación debe ponerse dentro de su propia clase con un único método que realice la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

www.detodoprogramacion.org

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.



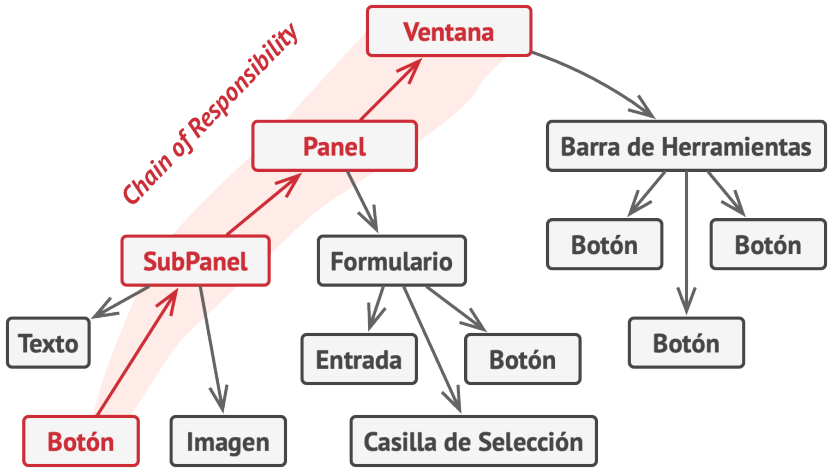
Los manejadores se alinean uno tras otro, formando una cadena.

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-

www.detodoprogramacion.org

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.

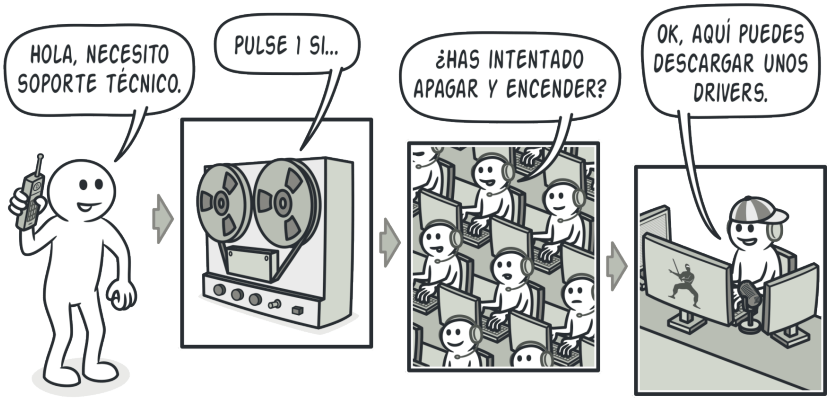
Es fundamental que todas las clases manejadoras implementen la misma interfaz. Cada manejadora concreta solo debe preocuparse por la siguiente que cuente con el método `ejecutar`. De esta forma puedes componer cadenas durante el tiempo de ejecución, utilizando varios manejadores sin acoplar tu código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



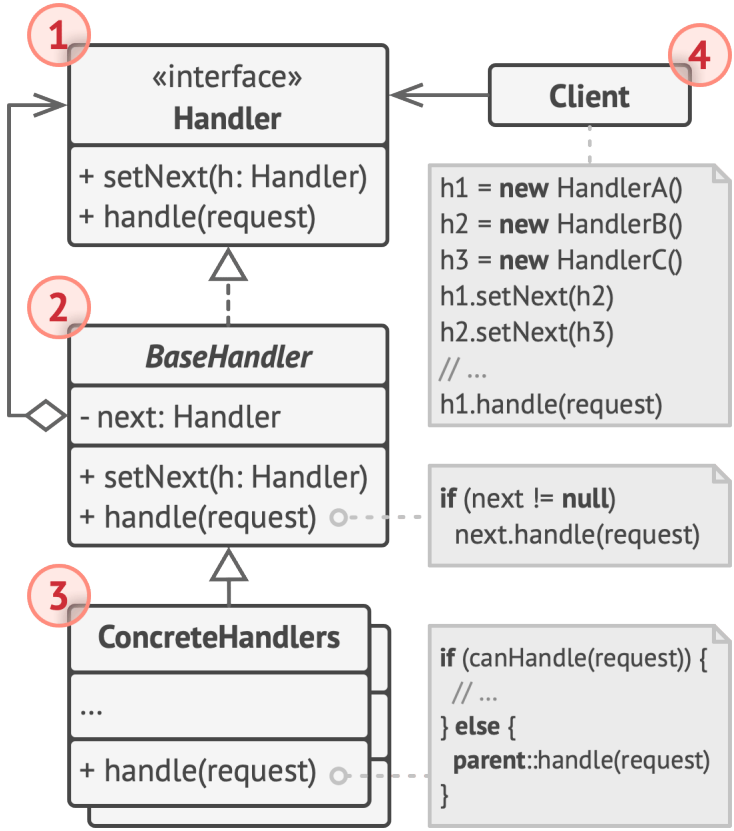
Una llamada al soporte técnico puede pasar por muchos operadores.

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.

www.detodoprogramacion.org

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.

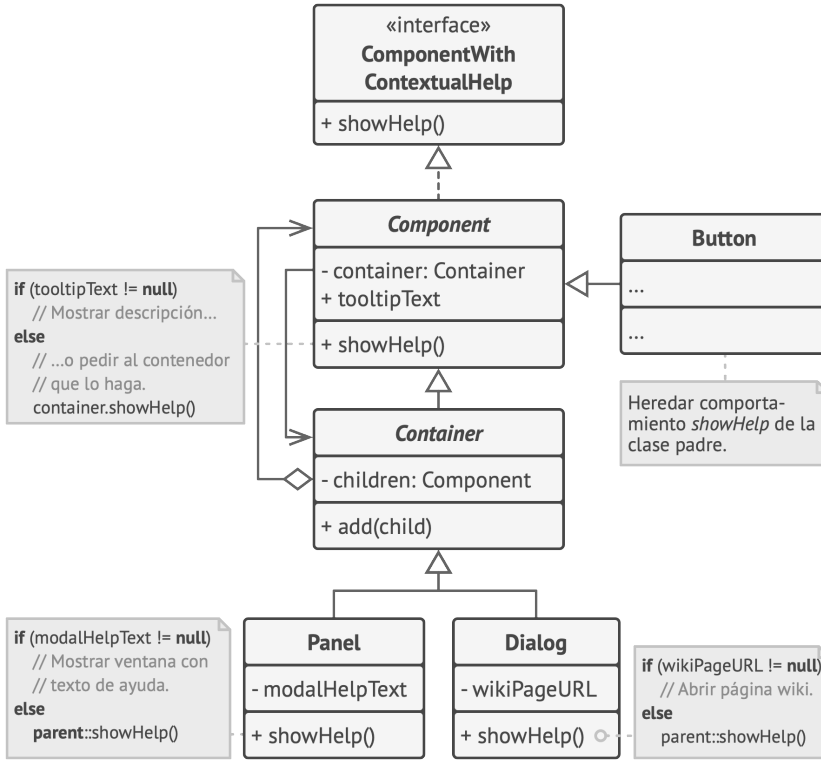
4. El **Cliente** puede componer cadenas una sola vez o componerlas dinámicamente, dependiendo de la lógica de la aplicación.

www.detodoprogramacion.org

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.

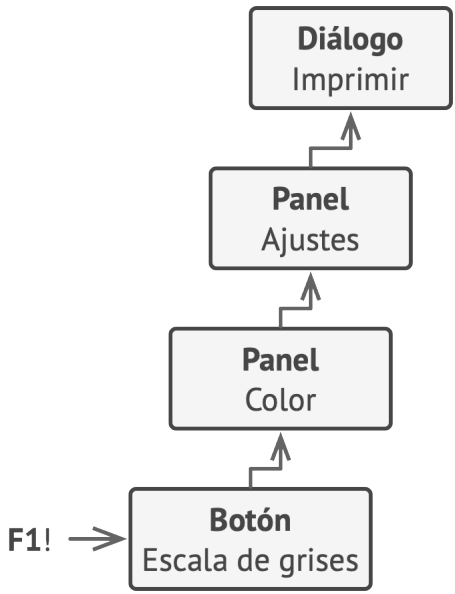


Las clases GUI se crean con el patrón Composite. Cada elemento se vincula a su elemento contenedor. En cualquier momento puedes crear una cadena de elementos que comience con el propio elemento y recorra todos los elementos contenedores.

www.detodoprogramacion.org

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.

www.detodoprogramacion.org

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.

```
// La interfaz manejadora declara un método para construir una
// cadena de manejadores. También declara un método para
// ejecutar una solicitud.
interface ComponentWithContextualHelp is
  method showHelp()


// La clase base para componentes simples.
abstract class Component implements ComponentWithContextualHelp is
  field tooltipText: string

  // El contenedor del componente actúa como el siguiente
  // eslabón de la cadena de manejadores.
  protected field container: Container

  // El componente muestra una pista si tiene un texto de
  // ayuda asignado. De lo contrario, reenvía la llamada al
  // contenedor, si es que existe.
  method showHelp() is
    if (tooltipText != null)
      // Muestra la pista.
    else
      container.showHelp()


```

```
// Los contenedores pueden contener componentes simples y otros
// contenedores como hijos. Las relaciones de la cadena se
// establecen aquí. La clase hereda el comportamiento showHelp
// (mostrarAyuda) de su padre.
abstract class Container extends Component is
  protected field children: array of Component

  method add(child) is
    children.add(child)
    child.container = this


// Los componentes primitivos pueden estar bien con la
// implementación de la ayuda por defecto...
class Button extends Component is
  // ...

// Pero los componentes complejos pueden sobrescribir la
// implementación por defecto. Si no puede proporcionarse el
// texto de ayuda de una nueva forma, el componente siempre
// puede invocar la implementación base (véase la clase
// Componente).
class Panel extends Container is
  field modalHelpText: string

  method showHelp() is
    if (modalHelpText != null)
      // Muestra una ventana modal con el texto de ayuda.
    else
      super.showHelp()

// ...igual que arriba...
```

```
class Dialog extends Container is
  field wikiPageURL: string

  method showHelp() is
    if (wikiPageURL != null)
      // Abre la página de ayuda wiki.
    else
      super.showHelp()


// Código cliente.
class Application is
  // Cada aplicación configura la cadena de forma diferente.
  method createUI() is
    dialog = new Dialog("Budget Reports")
    dialog.wikiPageURL = "http://..."
    panel = new Panel(0, 0, 400, 800)
    panel.modalHelpText = "This panel does..."
    ok = new Button(250, 760, 50, 20, "OK")
    ok.tooltipText = "This is an OK button that..."
    cancel = new Button(320, 760, 50, 20, "Cancel")
    // ...
    panel.add(ok)
    panel.add(cancel)
    dialog.add(panel)

  // Imagina lo que pasa aquí.
  method onF1KeyPress() is
    component = this.getComponentAtMouseCoords()
    component.showHelp()
```

El Mundo de la Programación en tus Manos...!
DETODOPROGRAMACION.ORG

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 conozcan 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.

www.detodoprogramacion.org

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.

Descargado de: www.detodoprogramacion.org

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:

www.detodoprogramacion.org

- 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.

www.detodoprogramacion.org

✓ *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 pasarla a lo largo de la cadena de todos los componentes padre hasta la raíz del árbol de objetos.

www.detodoprogramacion.org

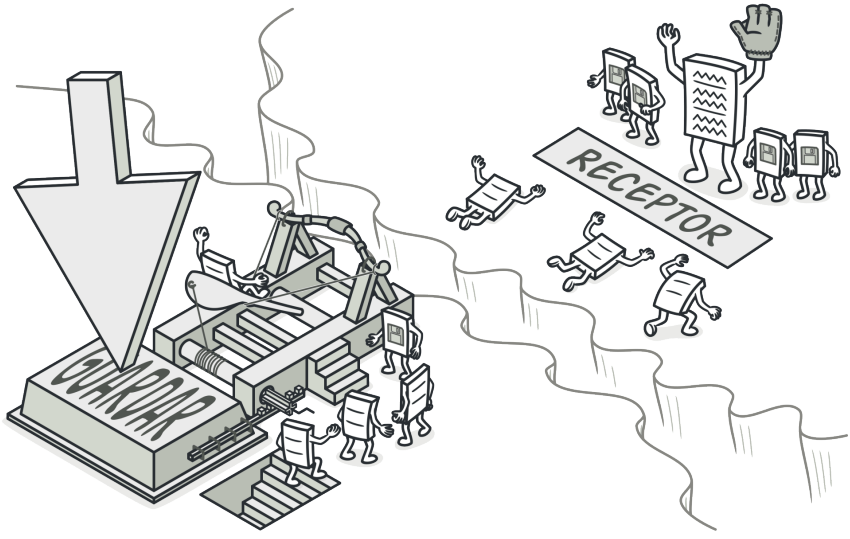
- Los manejadores del **Chain of Responsibility** se pueden implementar como **Comandos**. En este caso, puedes ejecutar muchas operaciones diferentes sobre el mismo objeto de contexto, representado por una solicitud.

  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.

www.detodoprogramacion.org



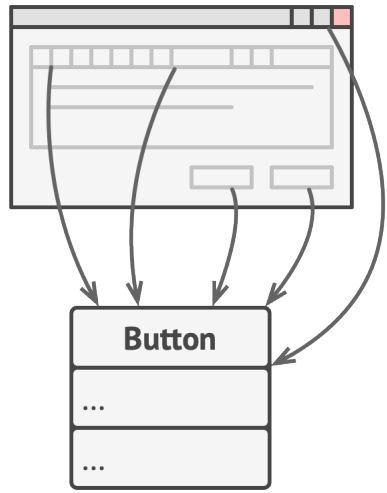
COMMAND

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 soportar operaciones que no se pueden realizar.

☹ Problema

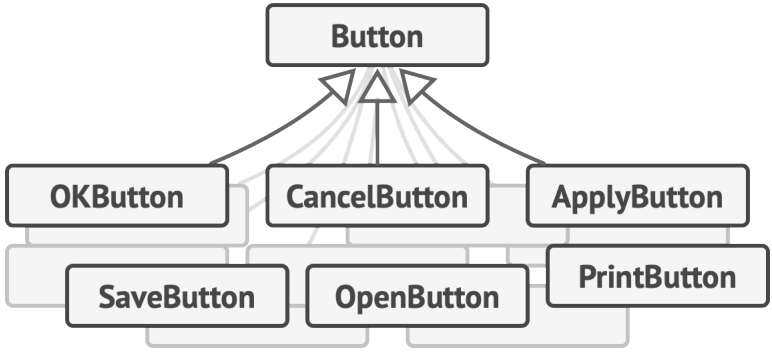
Imagina que estás trabajando en una nueva aplicación de edición de texto. Tu tarea actual consiste en crear una barra de herramientas con unos cuantos botones para varias operaciones del editor. Creaste una clase `Botón` muy limpia que puede utilizarse para los botones de la barra de herramientas y también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

www.detodoprogramacion.org



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.

www.detodoprogramacion.org

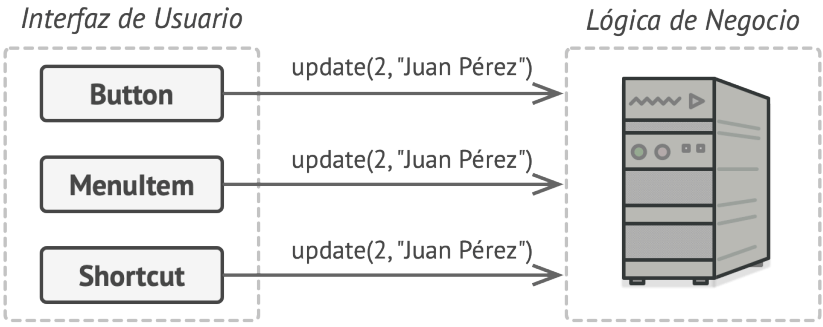
Inicialmente, cuando tu aplicación solo tenía la barra de herramientas, no había problema en colocar la implementación de varias operaciones dentro de las subclases de botón. En otras palabras, tener el código para copiar texto dentro de la subclase `BotónCopiar` estaba bien. Sin embargo, cuando implementas 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*.

www.detodoprogramacion.org

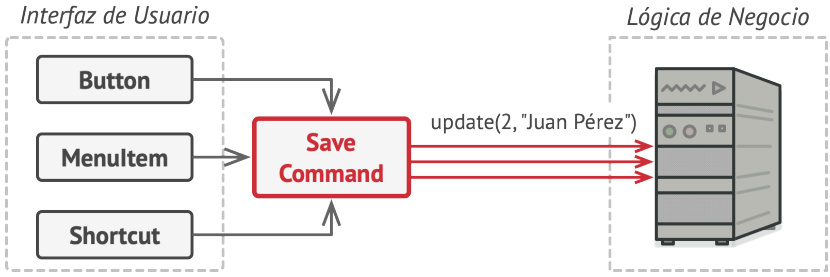


Los objetos GUI pueden acceder directamente a los objetos de la lógica de negocio.

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.

www.detodoprogramacion.org

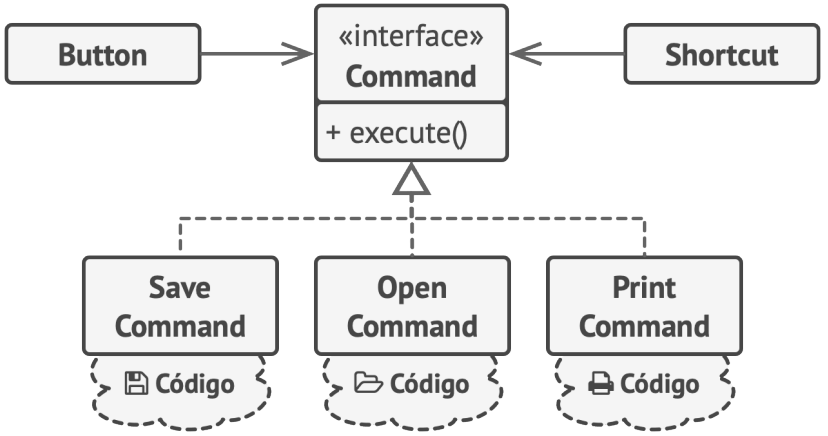


Acceso a la capa de lógica de negocio a través de un comando.

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.

www.detodoprogramacion.org



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.

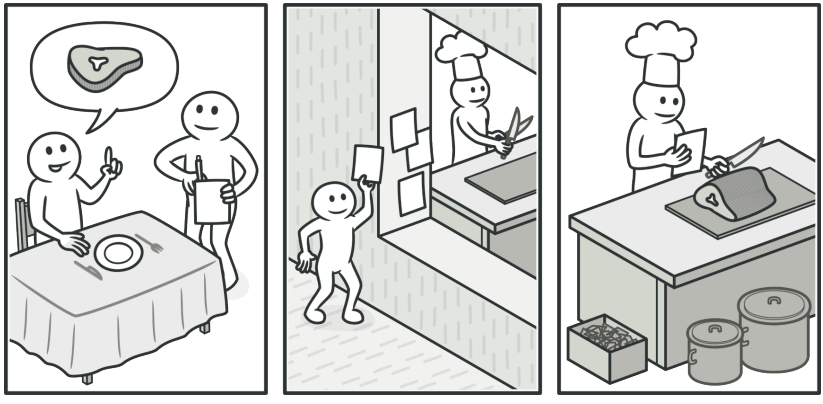
Otros elementos GUI, como menús, atajos o diálogos enteros, se pueden implementar del mismo modo. Se vincularán a un comando que se ejecuta cuando un usuario interactúa con el elemento GUI. Como probablemente ya habrás adivinado, los elementos relacionados con las mismas operaciones se vincu-

www.detodoprogramacion.org

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.

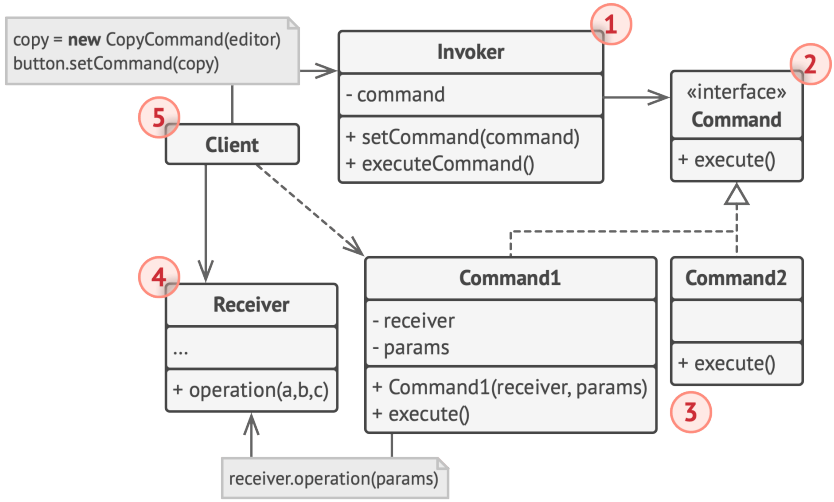
Tras un largo paseo por la ciudad, entras en un buen restaurante y te sientas a una mesa junto a la ventana. Un amable camarero se acerca y toma tu pedido rápidamente, apuntándolo en un papel. El camarero se va a la cocina y pega el pedido a la pared. Al cabo de un rato, el pedido llega al chef, que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

www.detodoprogramacion.org

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 precreado 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.

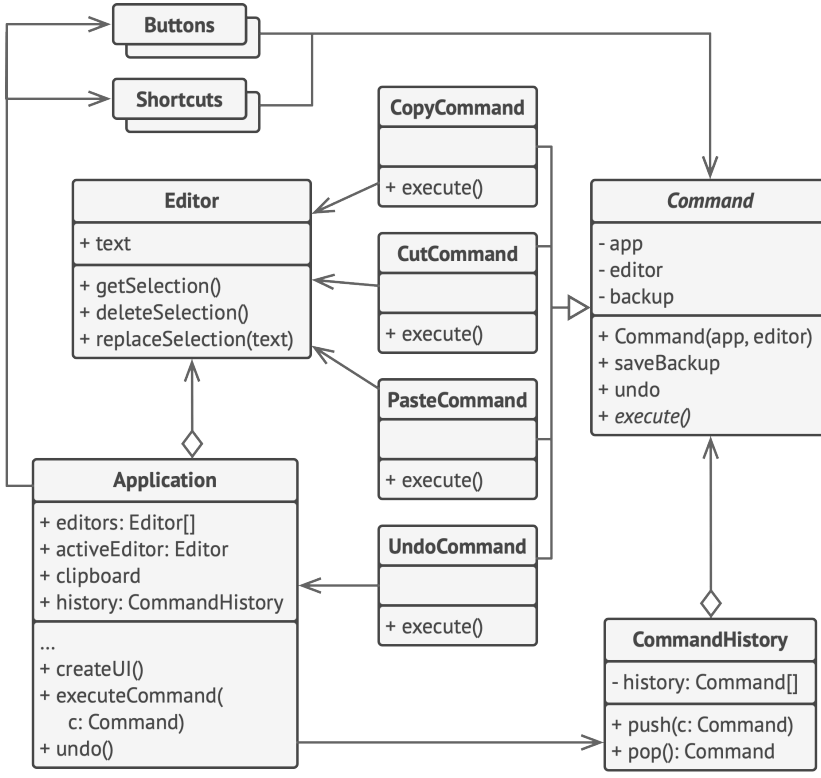
4. La clase **Receptora** contiene cierta lógica de negocio. Casi cualquier objeto puede actuar como receptor. La mayoría de los comandos solo gestiona los detalles sobre cómo se pasa una solicitud al receptor, mientras que el propio receptor hace el trabajo real.

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.

www.detodoprogramacion.org

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.

```
// La clase base comando define la interfaz común a todos los
// comandos concretos.
abstract class Command is
  protected field app: Application
  protected field editor: Editor
  protected field backup: text

  constructor Command(app: Application, editor: Editor) is
    this.app = app
    this.editor = editor

  // Realiza una copia de seguridad del estado del editor.
  method saveBackup() is
    backup = editor.text

  // Restaura el estado del editor.
  method undo() is
```

```
    editor.text = backup

  // El método de ejecución se declara abstracto para forzar a
  // todos los comandos concretos a proporcionar sus propias
  // implementaciones. El método debe devolver verdadero o
  // falso dependiendo de si el comando cambia el estado del
  // editor.
  abstract method execute()


// Los comandos concretos van aquí.
class CopyCommand extends Command is
  // El comando copiar no se guarda en el historial ya que no
  // cambia el estado del editor.
  method execute() is
    app.clipboard = editor.getSelection()
    return false

class CutCommand extends Command is
  // El comando cortar no cambia el estado del editor, por lo
  // que debe guardarse en el historial. Y se guardará siempre
  // y cuando el método devuelva verdadero.
  method execute() is
    saveBackup()
    app.clipboard = editor.getSelection()
    editor.deleteSelection()
    return true

class PasteCommand extends Command is
  method execute() is
    saveBackup()
    editor.replaceSelection(app.clipboard)
```

```
    return true

// La operación deshacer también es un comando.
class UndoCommand extends Command is
  method execute() is
    app.undo()
    return false


// El historial global de comandos tan solo es una pila.
class CommandHistory is
  private field history: array of Command

  // El último dentro...
  method push(c: Command) is
    // Empuja el comando al final de la matriz del
    // historial.

  // ...el primero fuera.
  method pop():Command is
    // Obtiene el comando más reciente del historial.


// La clase editora tiene operaciones reales de edición de
// texto. Juega el papel de un receptor: todos los comandos
// acaban delegando la ejecución a los métodos del editor.
class Editor is
  field text: string

  method getSelection() is
    // Devuelve el texto seleccionado.

```

```
    method deleteSelection() is
      // Borra el texto seleccionado.

    method replaceSelection(text) is
      // Inserta los contenidos del portapapeles en la
      // posición actual.


// La clase Aplicación establece relaciones entre objetos. Actúa
// como un emisor: cuando algo debe hacerse, crea un objeto de
// comando y lo ejecuta.
class Application is
    field clipboard: string
    field editors: array of Editors
    field activeEditor: Editor
    field history: CommandHistory

    // El código que asigna comandos a objetos UI puede tener
    // este aspecto.
    method createUI() is
      // ...
      copy = function() { executeCommand(
        new CopyCommand(this, activeEditor)) }
      copyButton.setCommand(copy)
      shortcuts.onKeyPress("Ctrl+C", copy)

      cut = function() { executeCommand(
        new CutCommand(this, activeEditor)) }
      cutButton.setCommand(cut)
      shortcuts.onKeyPress("Ctrl+X", cut)

      paste = function() { executeCommand(
```

```
      new PasteCommand(this, activeEditor)) }
    pasteButton.setCommand(paste)
    shortcuts.onKeyPress("Ctrl+V", paste)

    undo = function() { executeCommand(
      new UndoCommand(this, activeEditor)) }
    undoButton.setCommand(undo)
    shortcuts.onKeyPress("Ctrl+Z", undo)

// Ejecuta un comando y comprueba si debe añadirse al
// historial.
method executeCommand(command) is
  if (command.execute)
    history.push(command)

// Toma el comando más reciente del historial y ejecuta su
// método deshacer. Observa que no conocemos la clase de ese
// comando. Pero no tenemos por qué, ya que el comando sabe
// cómo deshacer su propia acción.
method undo() is
  command = history.pop()
  if (command != null)
    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.

www.detodoprogramacion.org

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.

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.

www.detodoprogramacion.org

✕ 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.

www.detodoprogramacion.org

- 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.

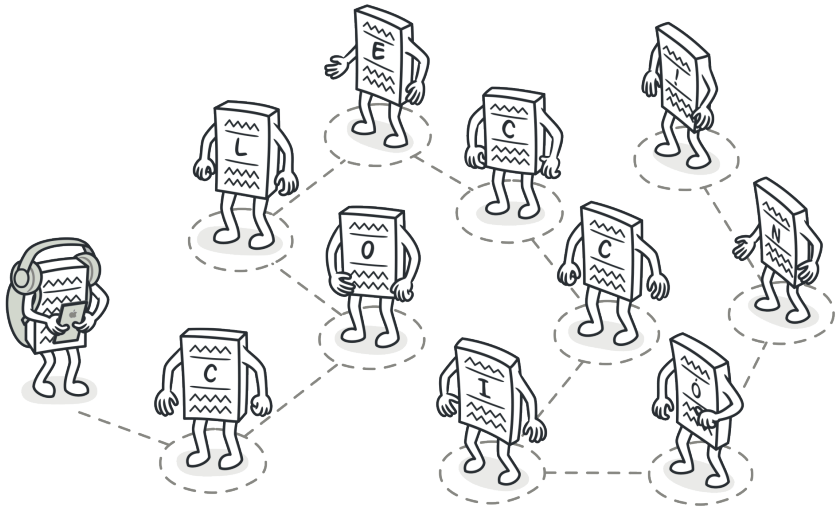
    - Puedes utilizar *Command* para convertir cualquier operación en un objeto. Los parámetros de la operación se convierten en campos de ese objeto. La conversión te permite aplazar la ejecución de la operación, ponerla en cola, almacenar el historial de comandos, enviar comandos a servicios remotos, etc.

    - 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.

www.detodoprogramacion.org



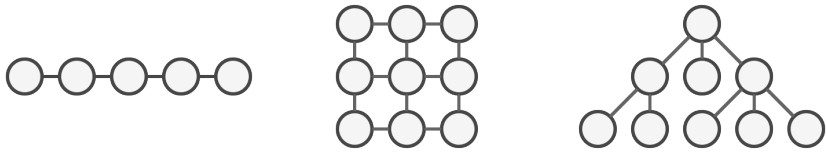
ITERATOR

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.

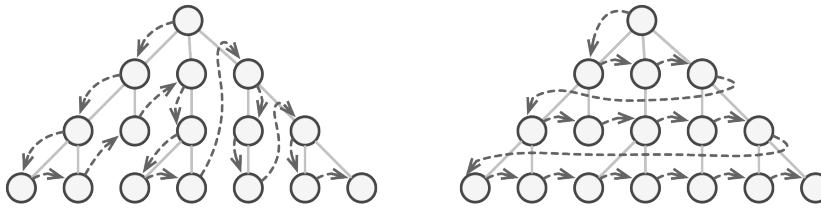
La mayoría de las colecciones almacena sus elementos en simples listas, pero algunas de ellas se basan en pilas, árboles, grafos y otras estructuras complejas de datos.

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, quizá necesites un recorrido en anchura. Y, la semana siguiente, puedes

www.detodoprogramacion.org

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



La misma colección puede recorrerse de varias formas diferentes.

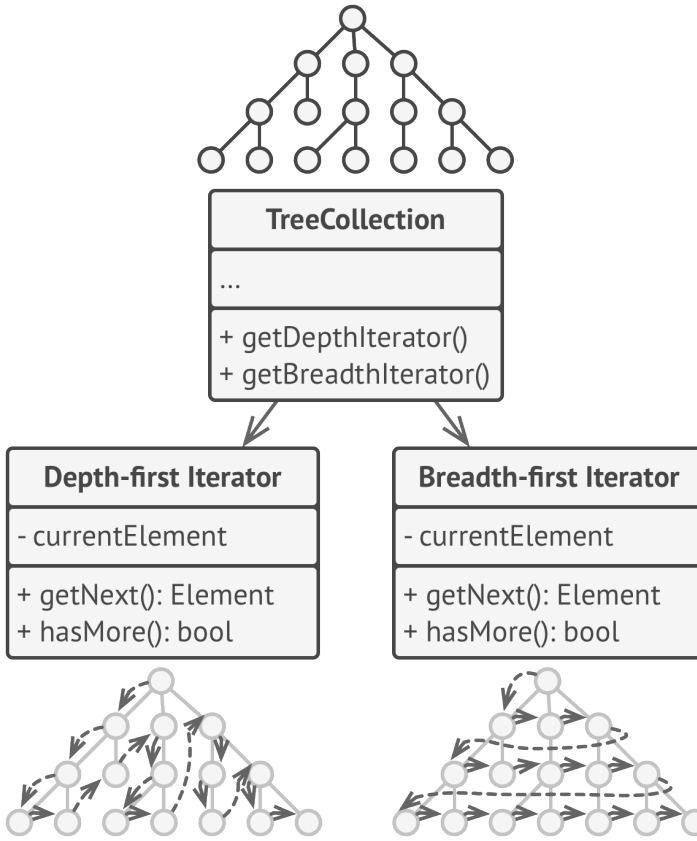
Añadir más y más algoritmos de recorrido a la colección nubla gradualmente su responsabilidad principal, que es el almacenamiento eficiente de la información. Además, puede que algunos algoritmos estén personalizados para una aplicación específica, por lo que incluirlos en una clase genérica de colección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

www.detodoprogramacion.org



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.

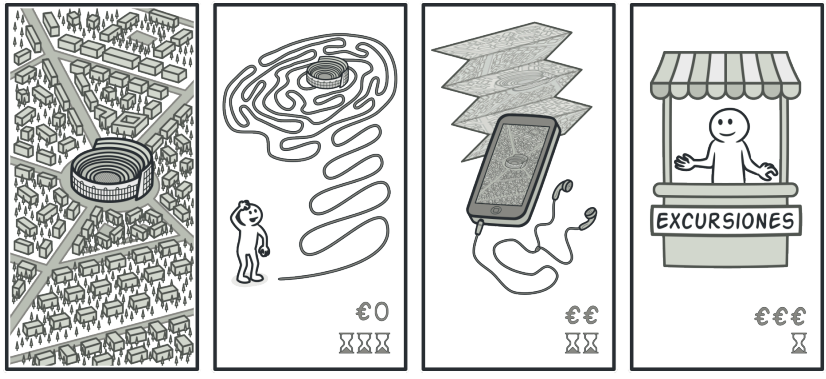
Normalmente, los iteradores aportan un método principal para extraer elementos de la colección. El cliente puede continuar

www.detodoprogramacion.org

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

www.detodoprogramacion.org

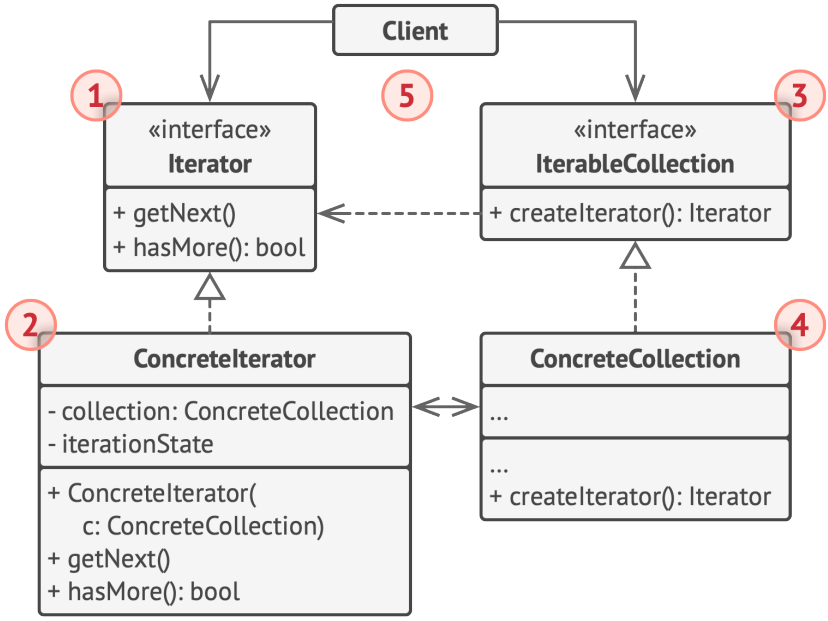
barata y puedes quedarte en sitios interesantes todo el tiempo que quieras.

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.

www.detodoprogramacion.org

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.

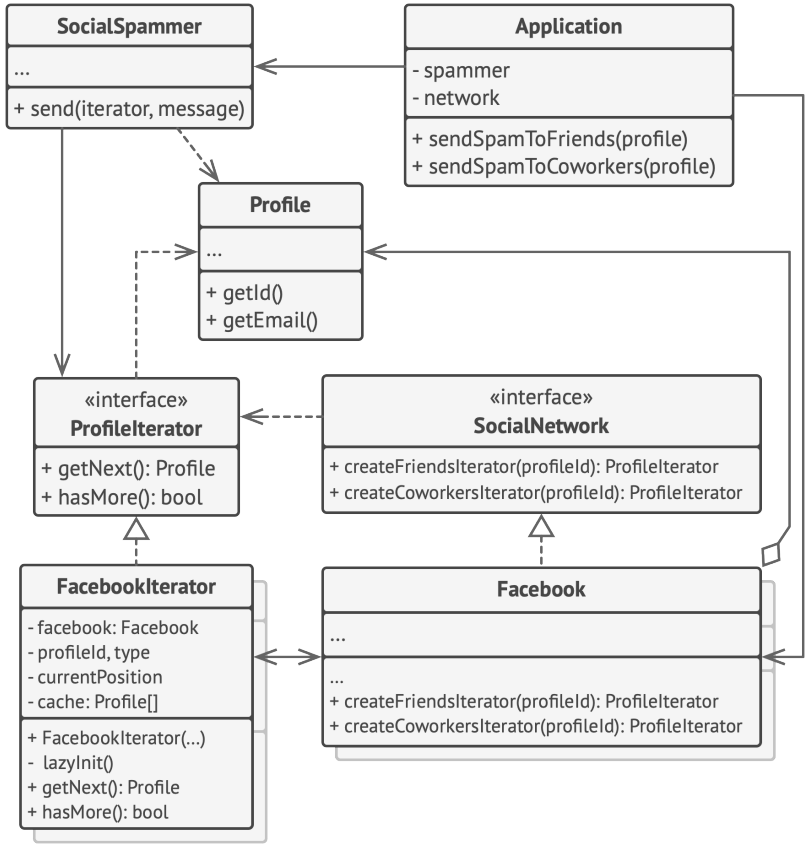
5. El **Cliente** debe funcionar con colecciones e iteradores a través de sus interfaces. De este modo, el cliente no se acopla a clases concretas, permitiéndote utilizar varias colecciones e iteradores con el mismo código cliente.

   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.

www.detodoprogramacion.org



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.

www.detodoprogramacion.org

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.

```
// La interfaz de colección debe declarar un método fábrica para
// producir iteradores. Puedes declarar varios métodos si hay
// distintos tipos de iteración disponibles en tu programa.
interface SocialNetwork is
  method createFriendsIterator(profileId):ProfileIterator
  method createCoworkersIterator(profileId):ProfileIterator


// Cada colección concreta está acoplada a un grupo de clases
// iteradoras concretas que devuelve, pero el cliente no lo
// está, ya que la firma de estos métodos devuelve interfaces
// iteradoras.
class Facebook implements SocialNetwork is
  // ... El grueso del código de la colección debe ir aquí ...
  // Código de creación del iterador.
  method createFriendsIterator(profileId) is
    return new FacebookIterator(this, profileId, "friends")
  method createCoworkersIterator(profileId) is
    return new FacebookIterator(this, profileId, "coworkers")


// La interfaz común a todos los iteradores.
interface ProfileIterator is
  method getNext():Profile
  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
```

```
    lazyInit()
    return currentPosition < cache.length


// Aquí tienes otro truco útil: puedes pasar un iterador a una
// clase cliente en lugar de darle acceso a una colección
// completa. De esta forma, no expones la colección al cliente.
//
// Y hay otra ventaja: puedes cambiar la forma en la que el
// cliente trabaja con la colección durante el tiempo de
// ejecución pasándole un iterador diferente. Esto es posible
// porque el código cliente no está acoplado a clases iteradoras
// concretas.
class SocialSpammer is
  method send(iterator: ProfileIterator, message: string) is
    while (iterator.hasMore())
      profile = iterator.getNext()
      System.sendEmail(profile.getEmail(), message)


// La clase Aplicación configura colecciones e iteradores y
// después los pasa al código cliente.
class Application is
  field network: SocialNetwork
  field spammer: SocialSpammer

  method config() is
    if working with Facebook
      this.network = new Facebook()
    if working with LinkedIn
      this.network = new LinkedIn()
    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 pasarlos al código existente sin descomponer nada.
- ✓ Puedes recorrer la misma colección en paralelo porque cada objeto iterador contiene su propio estado de iteración.

www.detodoprogramacion.org

✓ 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

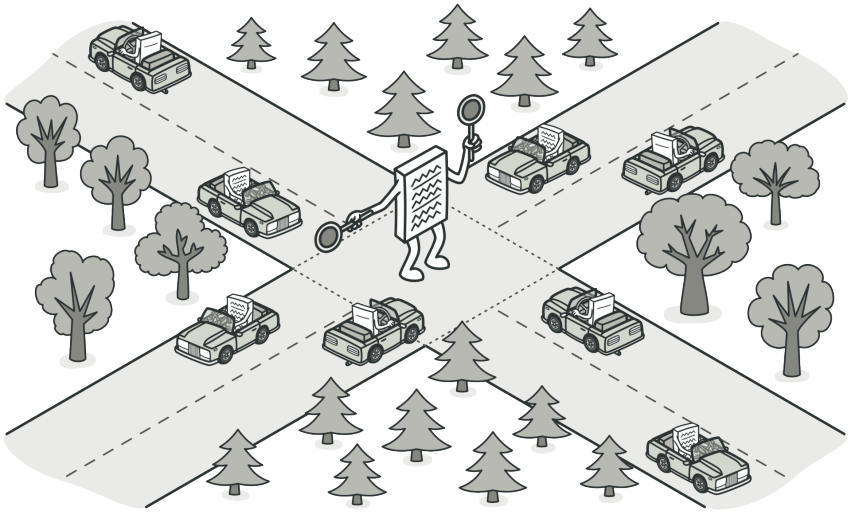
- Puedes utilizar **Iteradores** para recorrer árboles **Composite**.

- 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.

www.detodoprogramacion.org



MEDIATOR

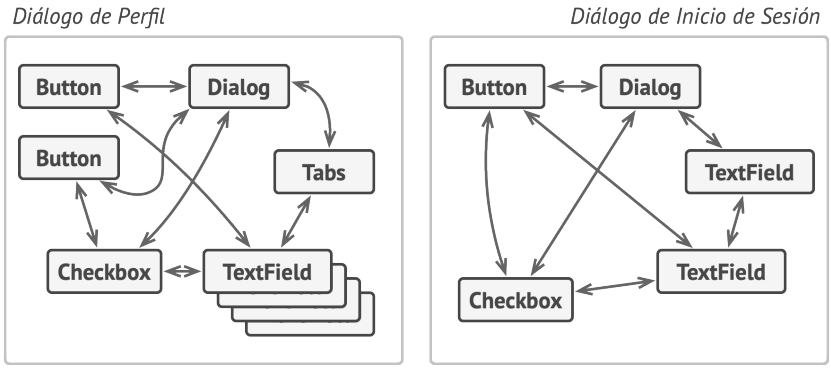
También llamado: Mediador, Intermediary, Controller

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.

www.detodoprogramacion.org

☹ 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.

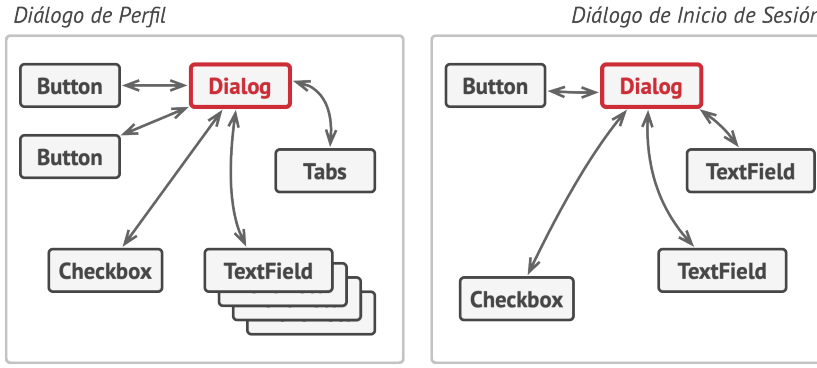
😊 Solución

El patrón Mediator sugiere que detengas toda comunicación directa entre los componentes que quieres 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

www.detodoprogramacion.org

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.

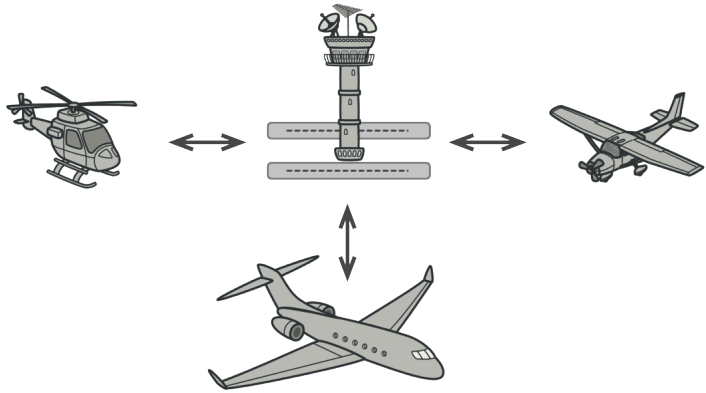
Puedes ir más lejos y reducir en mayor medida la dependencia extrayendo la interfaz común para todos los tipos de diálogo. La interfaz declarará el método de notificación que pueden uti-

www.detodoprogramacion.org

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.

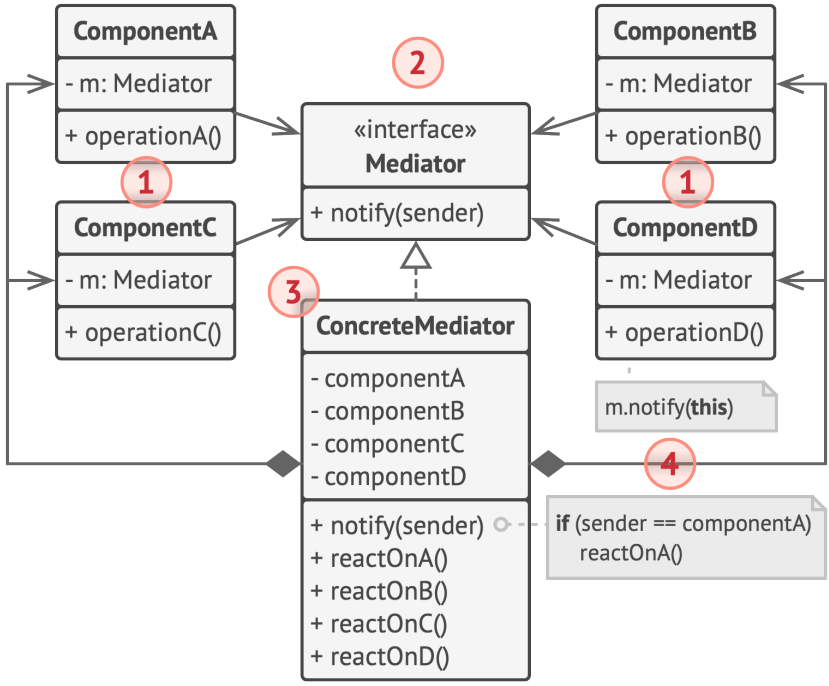
Los pilotos de los aviones que llegan o salen del área de control del aeropuerto no se comunican directamente entre sí. En lugar de eso, hablan con un controlador de tráfico aéreo, que está sentado en una torre alta cerca de la pista de aterrizaje.

www.detodoprogramacion.org

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.

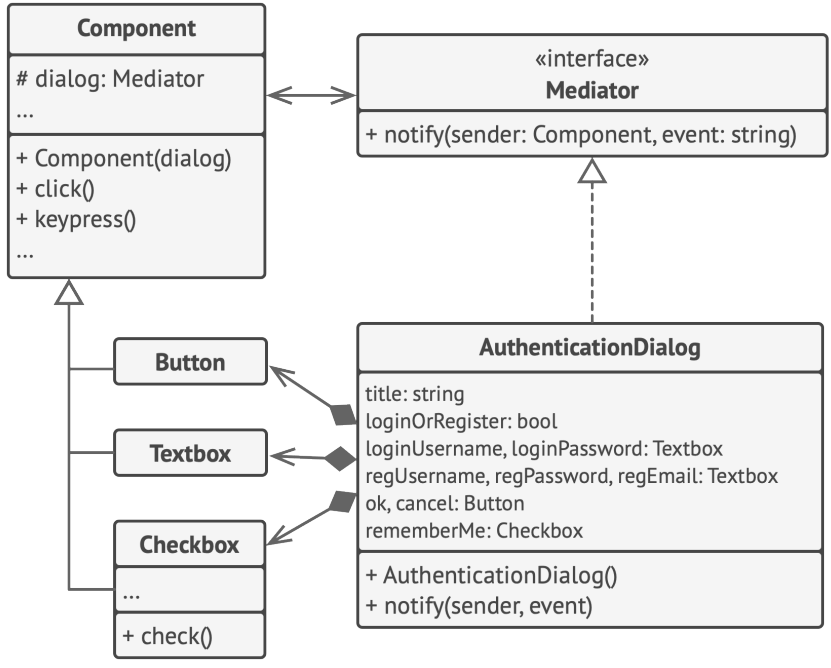
4. Los componentes no deben conocer otros componentes. Si le sucede algo importante a un componente, o dentro de él, sólo debe notificar a la interfaz mediadora. Cuando la mediadora recibe la notificación, puede identificar fácilmente al emisor, lo cual puede ser suficiente para decidir qué componente 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.

www.detodoprogramacion.org

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.

www.detodoprogramacion.org

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.

```
// La interfaz mediadora declara un método utilizado por los
// componentes para notificar al mediador sobre varios eventos.
// El mediador puede reaccionar a estos eventos y pasar la
// ejecución a otros componentes.
interface Mediator is
  method notify(sender: Component, event: string)


// La clase concreta mediadora. La red entrecruzada de
// conexiones entre componentes individuales se ha desenredado y
// se ha colocado dentro de la mediadora.
class AuthenticationDialog implements Mediator is
  private field title: string
  private field loginOrRegisterChkBx: Checkbox
  private field loginUsername, loginPassword: Textbox
  private field registrationUsername, registrationPassword,
          registrationEmail: Textbox
  private field okBtn, cancelBtn: Button

  constructor AuthenticationDialog() is
    // Crea todos los objetos del componente y pasa el
    // mediador actual a sus constructores para establecer
    // vínculos.

  // Cuando sucede algo con un componente, notifica al
```

```
    // mediador, que al recibir la notificación, puede hacer
    // algo por su cuenta o pasar la solicitud a otro
    // componente.
    method notify(sender, event) is
      if (sender == loginOrRegisterChkBx and event == "check")
        if (loginOrRegisterChkBx.checked)
          title = "Log in"
          // 1. Muestra los componentes del formulario de
          // inicio de sesión.
          // 2. Esconde los componentes del formulario de
          // registro.
        else
          title = "Register"
          // 1. Muestra los componentes del formulario de
          // registro.
          // 2. Esconde los componentes del formulario de
          // inicio de sesión.

      if (sender == okBtn && event == "click")
        if (loginOrRegister.checked)
          // Intenta encontrar un usuario utilizando las
          // credenciales de inicio de sesión.
          if (!found)
            // Muestra un mensaje de error sobre el
            // campo de inicio de sesión.
        else
          // 1. Crea una cuenta de usuario utilizando
          // información de los campos de registro.
          // 2. Ingresa a ese usuario.
          // ...
```

```
// Los componentes se comunican con un mediador utilizando la
// interfaz mediadora. Gracias a ello, puedes utilizar los
// mismos componentes en otros contextos vinculándolos con
// diferentes objetos mediadores.
class Component is
  field dialog: Mediator

  constructor Component(dialog) is
    this.dialog = dialog

  method click() is
    dialog.notify(this, "click")

  method keypress() is
    dialog.notify(this, "keypress")

// Los componentes concretos no hablan entre sí. Sólo tienen un
// canal de comunicación, que es el envío de notificaciones al
// mediador.
class Button extends Component is
  // ...

class Textbox extends Component is
  // ...

class Checkbox extends Component is
  method check() is
    dialog.notify(this, "check")
  // ...
```

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.

www.detodoprogramacion.org

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-

www.detodoprogramacion.org

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

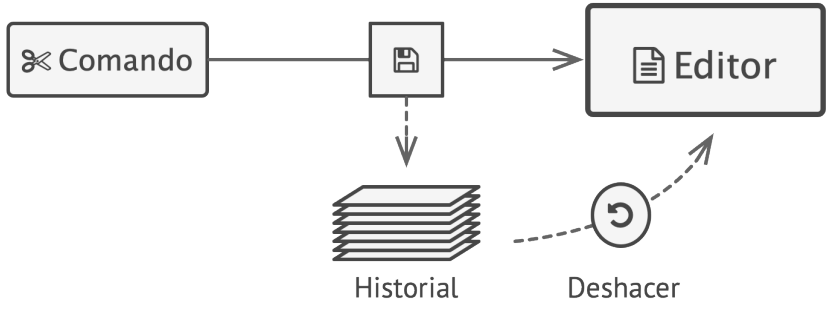
También llamado: Recuerdo, Instantánea, Snapshot

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, asi 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.



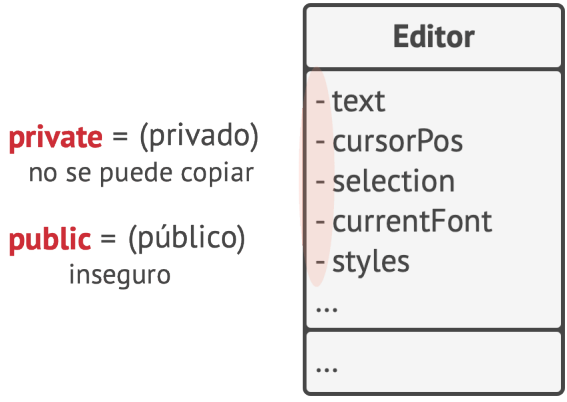
Antes de ejecutar una operación, la aplicación guarda una instantánea del estado de los objetos, que más tarde se puede utilizar para restaurar objetos a su estado previo.

www.detodoprogramacion.org

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.

www.detodoprogramacion.org



¿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

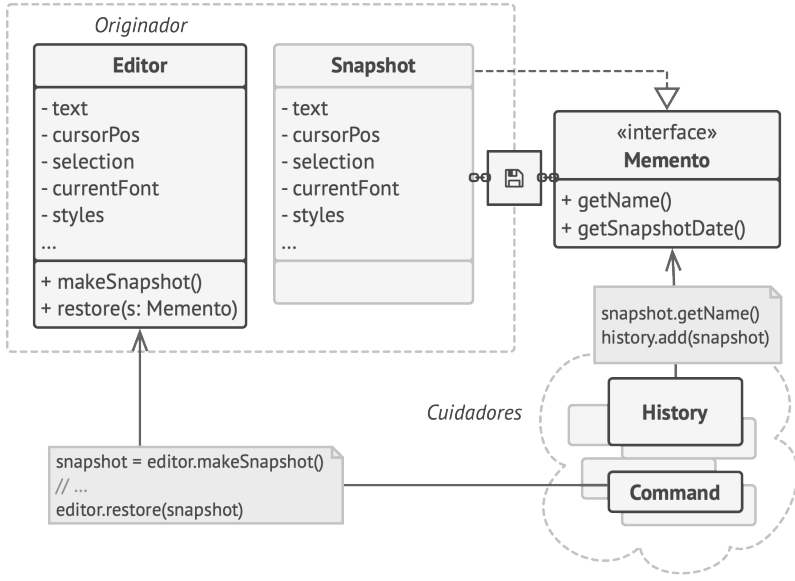
Todos los problemas que hemos experimentado han sido provocados por una encapsulación fragmentada. Algunos objetos intentan hacer más de lo que deben. Para recopilar los datos necesarios para realizar una acción, invaden el espacio privado de otros objetos en lugar de permitir a esos objetos realizar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

www.detodoprogramacion.org

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.

www.detodoprogramacion.org

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.

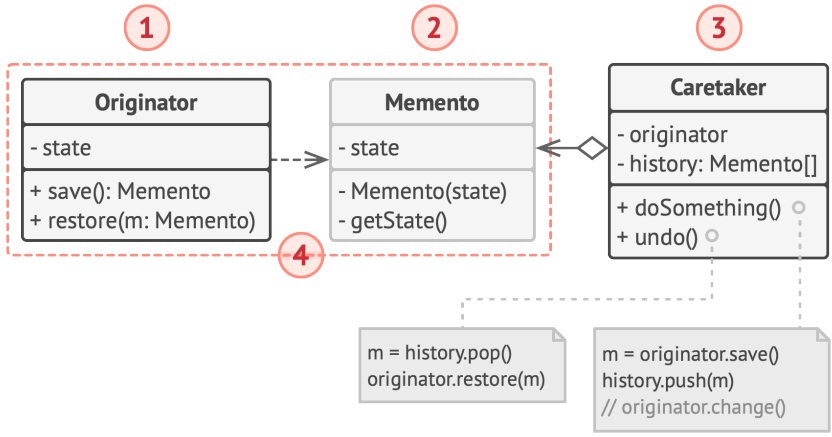
Cuando un usuario activa la función Deshacer, el historial toma el memento más reciente de la pila y lo pasa de vuelta al editor, solicitando una restauración. Debido a que el editor tiene pleno acceso al memento, cambia su propio estado con los valores tomados del memento.

www.detodoprogramacion.org

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).



Descargado de: www.detodoprogramacion.org

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.

www.detodoprogramacion.org

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

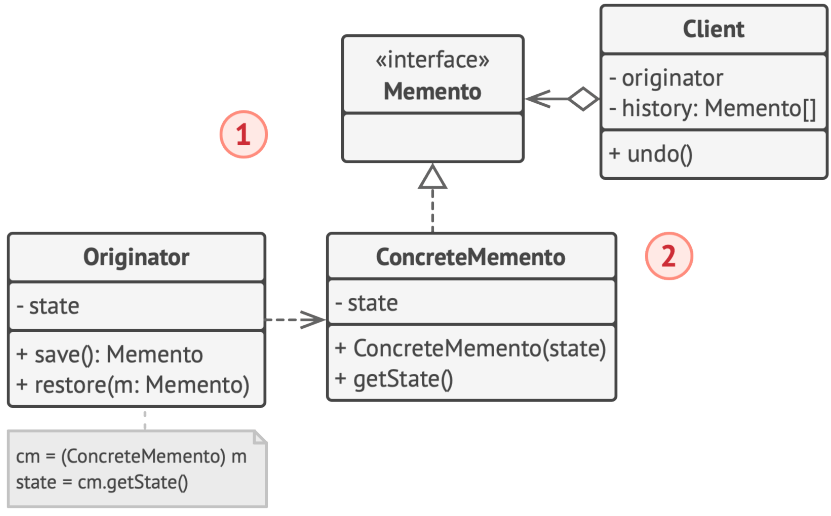
   Una cuidadora puede rastrear el historial de la originadora almacenando una pila de mementos. Cuando la originadora deba retroceder en el historial, la cuidadora extraerá el memento de más arriba de la pila y lo pasará al método de restauración de la originadora.

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.

www.detodoprogramacion.org

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).



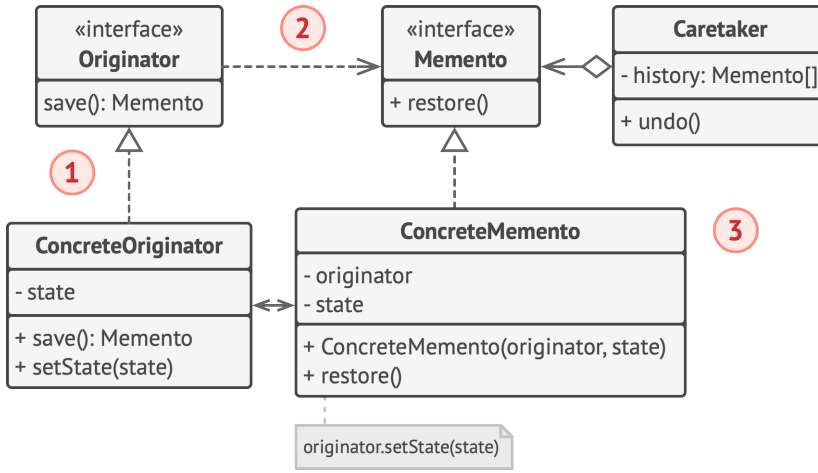
1. En ausencia de clases anidadas, puedes restringir el acceso a los campos de la clase memento estableciendo una convención de que las cuidadoras sólo pueden trabajar con una memento a través de una interfaz intermediaria explícitamente declarada, que sólo declarará métodos relacionados con los metadatos 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.



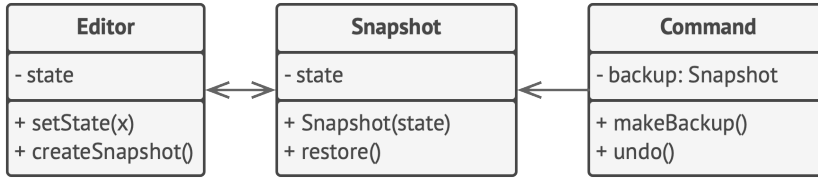
1. Esta implementación permite tener varios tipos de originadoras y mementos. Cada originadora trabaja con una clase memento correspondiente. Ninguna de las dos expone su estado a nadie.

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.

```
// El originador contiene información importante que puede
// cambiar con el paso del tiempo. También define un método para
// guardar su estado dentro de un memento, y otro método para
// restaurar el estado a partir de él.
class Editor is
  private field text, curX, curY, selectionWidth

  method setText(text) is
    this.text = text

  method setCursor(x, y) is
    this.curX = curX
    this.curY = curY

  method setSelectionWidth(width) is
    this.selectionWidth = width

  // Guarda el estado actual dentro de un memento.
  method createSnapshot():Snapshot is
    // El memento es un objeto inmutable; ese es el motivo
    // por el que el originador pasa su estado a los
    // parámetros de su constructor.
    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, consultores 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.

www.detodoprogramacion.org

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-

www.detodoprogramacion.org

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.

www.detodoprogramacion.org

✓ 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.

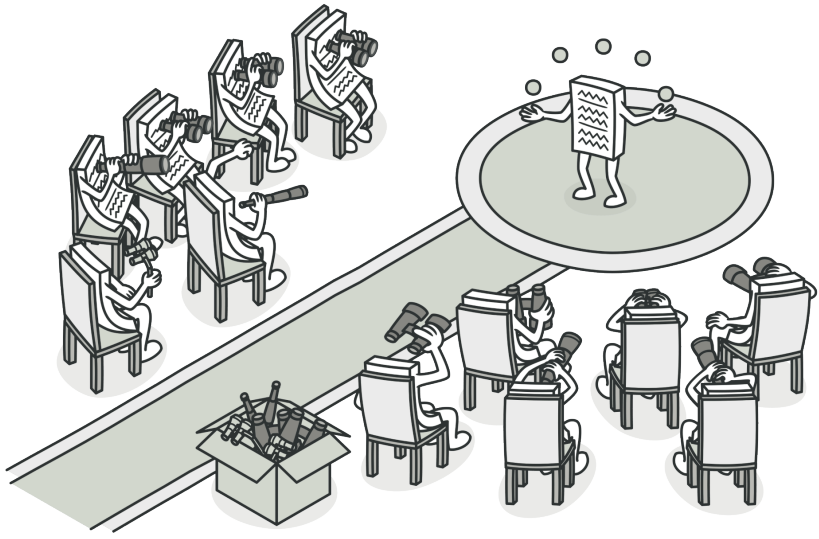
⇄ Relaciones con otros patrones

- 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 quieres almacenar en el historial es suficientemente sencillo y no tiene enlaces a recursos externos, o estos son fáciles de restablecer.

www.detodoprogramacion.org



OBSERVER

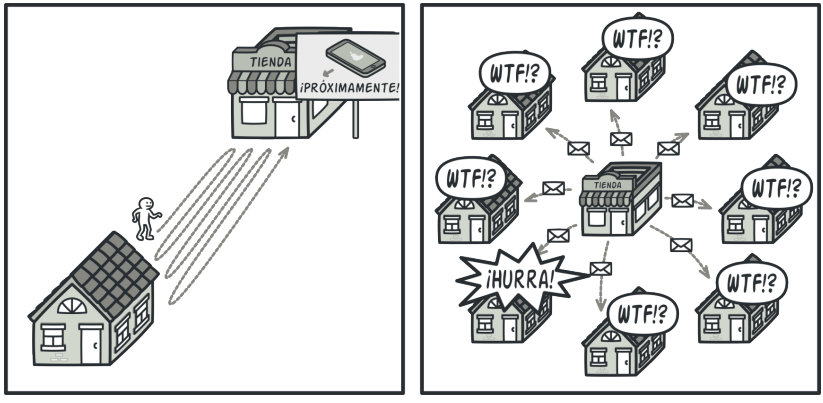
También llamado: Observador, Publicación-Suscripción, Modelo-patrón, Event-Subscriber, Listener

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.

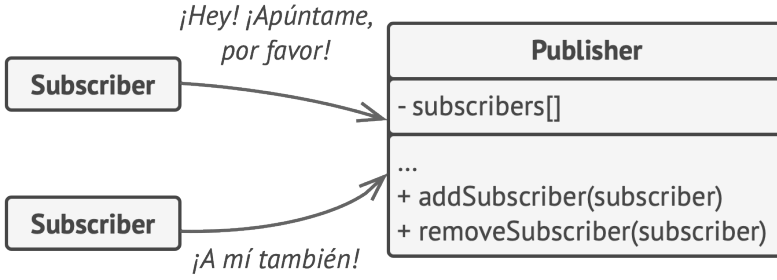
www.detodoprogramacion.org

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.

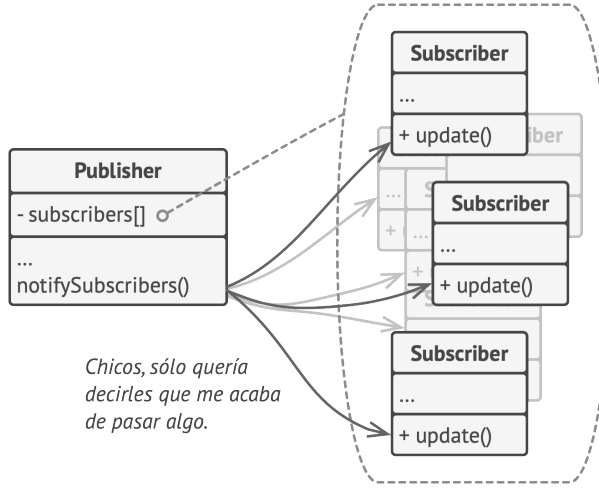


Un mecanismo de suscripción permite a los objetos individuales suscribirse a notificaciones de eventos.

www.detodoprogramacion.org

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-

www.detodoprogramacion.org

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



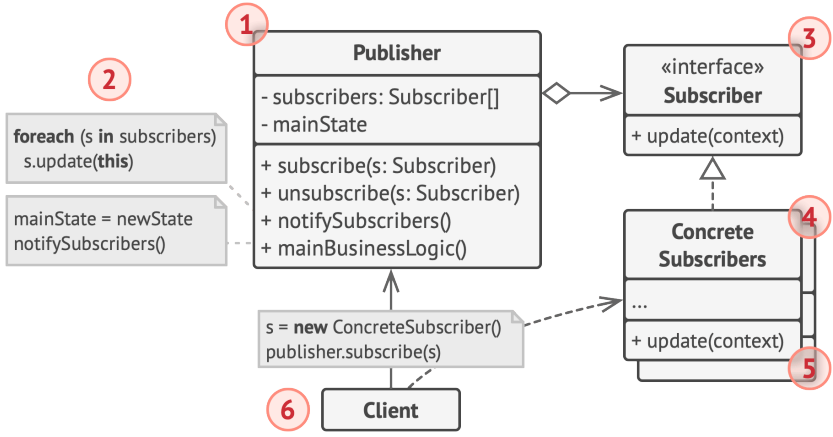
Suscripciones a revistas y periódicos.

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.

www.detodoprogramacion.org

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.

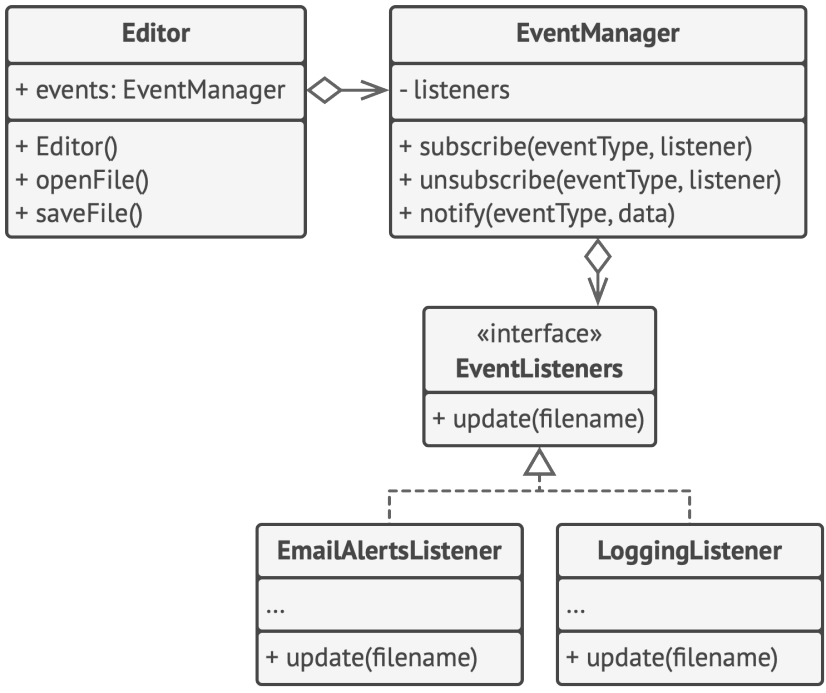
5. Normalmente, los suscriptores necesitan cierta información contextual para manejar correctamente la actualización. Por este motivo, a menudo los notificadores pasan cierta información de contexto como argumentos del método de notificación. El notificador puede pasarse a sí mismo como argumento, dejando 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.

www.detodoprogramacion.org



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 desees 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.

```
// La clase notificadora base incluye código de gestión de
// suscripciones y métodos de notificación.
class EventManager is
  private field listeners: hash map of event types and listeners

  method subscribe(eventType, listener) is
    listeners.add(eventType, listener)

  method unsubscribe(eventType, listener) is
    listeners.remove(eventType, listener)

  method notify(eventType, data) is
    foreach (listener in listeners.of(eventType)) do
      listener.update(data)

// El notificador concreto contiene lógica de negocio real, de
// interés para algunos suscriptores. Podemos derivar esta clase
// de la notificadora base, pero esto no siempre es posible en
// el mundo real porque puede que la notificadora concreta sea
// ya una subclase. En este caso, puedes modificar la lógica de
// la suscripción con composición, como hicimos aquí.
class Editor is
  public field events: EventManager
  private field file: File

  constructor Editor() is
    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
```

```
    method update(filename) is
      log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string

    constructor EmailAlertsListener(email, message) is
      this.email = email
      this.message = message

    method update(filename) is
      system.email(email, replace('%s',filename,message))


// Una  aplicación puede configurar notificadores y suscriptores
// durante el tiempo de ejecución.
class Application is
    method config() is
      editor = new Editor()

      logger = new LoggingListener(
        "/path/to/log.txt",
        "Someone has opened the file: %s")
      editor.events.subscribe("open", logger)

      emailAlerts = new EmailAlertsListener(
        "admin@example.com",
        "Someone has changed the file: %s")
      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 creaste clases personalizadas de botón y quieres 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.

www.detodoprogramacion.org

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.

Pros y contras

Descargado de: www.detodoprogramacion.org

✓ *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.

www.detodoprogramacion.org

⇄ 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

www.detodoprogramacion.org

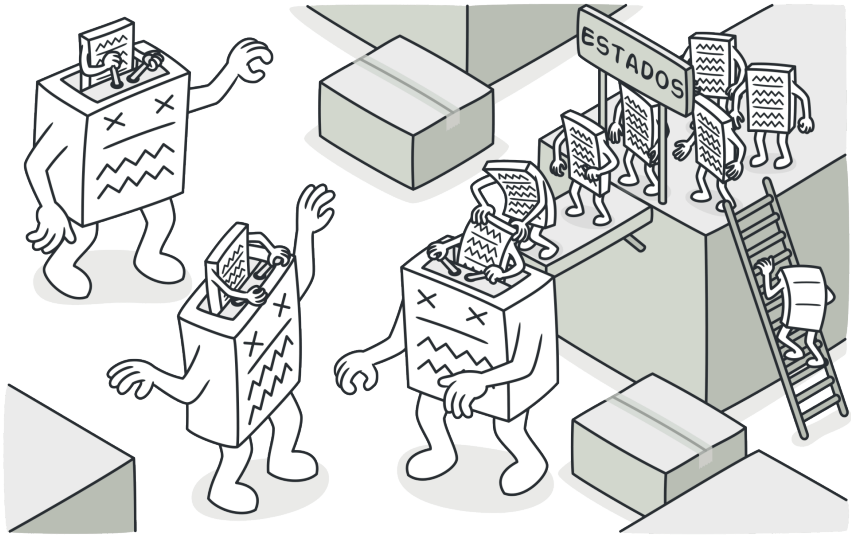
entre objetos, donde algunos objetos actúan como subordinados de otros.

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.

www.detodoprogramacion.org



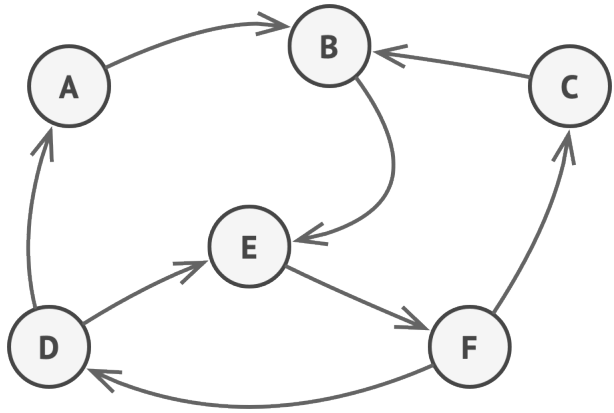
STATE

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**.



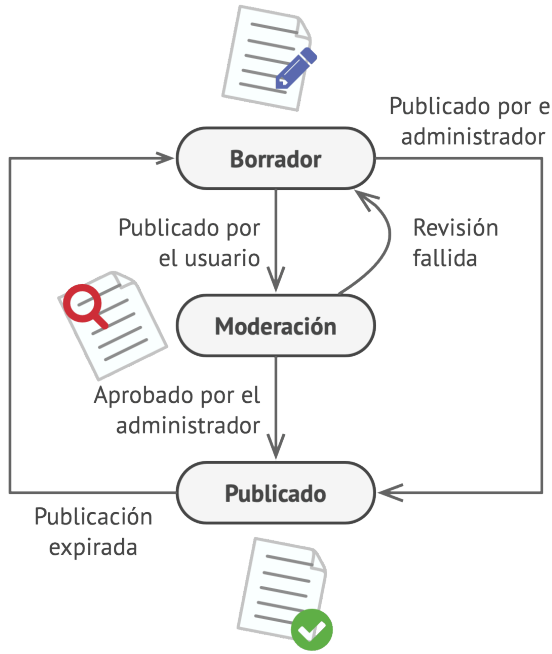
Máquina de estados finitos.

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.



Posibles 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?

```
class Document is
  field state: string
  // ...
  method publish() is
    switch (state)
      "draft":
        state = "moderation"
        break
      "moderation":
        if (currentUser.role == 'admin')
          state = "published"
        break
      "published":
        // No hacer nada.
        break
  // ...
```

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.

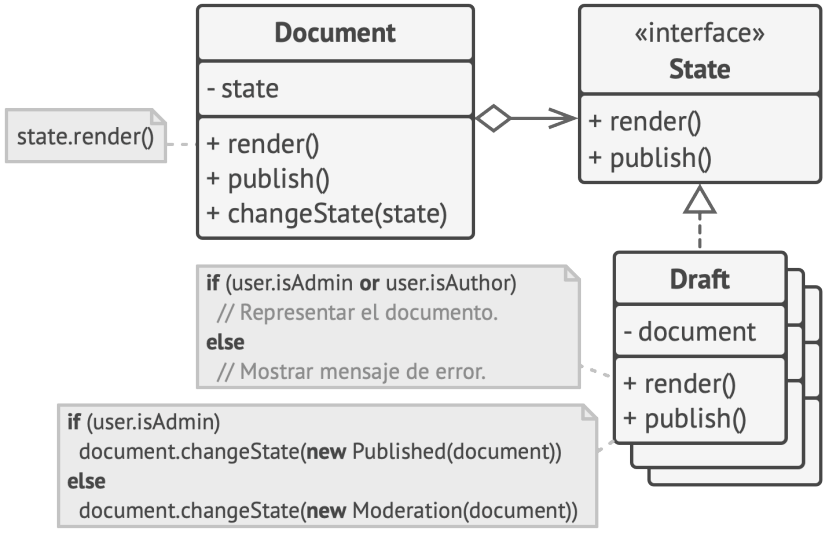
El problema tiende a empeorar con la evolución del proyecto. Es bastante difícil predecir todos los estados y transiciones posibles en la etapa de diseño. Por ello, una máquina de estados esbelta, creada con un grupo limitado de condicionales, puede crecer hasta convertirse en un abotargado desastre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

www.detodoprogramacion.org



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.

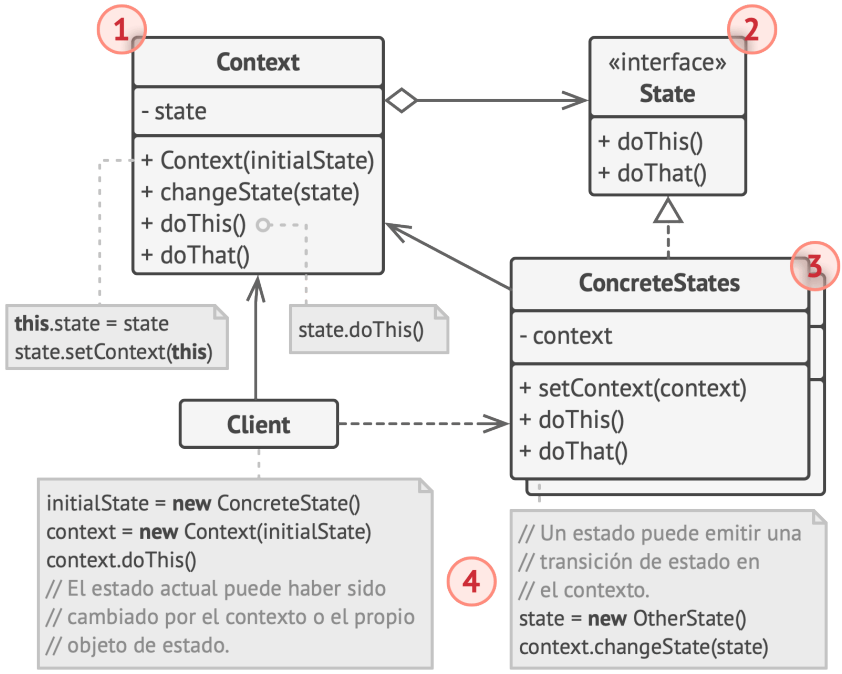
Esta estructura puede resultar similar al patrón **Strategy**, pero hay una diferencia clave. En el patrón State, los estados particulares pueden conocerse entre sí e iniciar transiciones de un estado a otro, mientras que las estrategias casi nunca se conocen.

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.

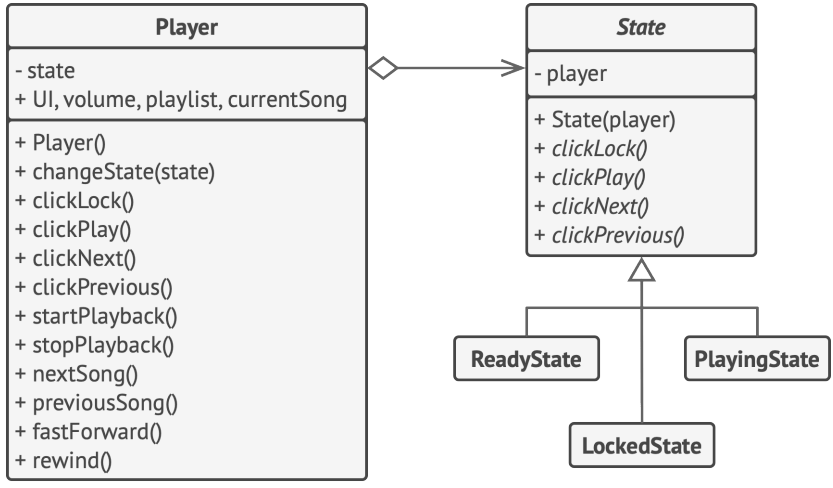
   Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

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.

www.detodoprogramacion.org

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.

```
// La clase ReproductordeAudio actúa como un contexto. También
// mantiene una referencia a una instancia de una de las clases
// estado que representa el estado actual del reproductor de
// audio.
class AudioPlayer is
  field state: State
  field UI, volume, playlist, currentSong

  constructor AudioPlayer() is
    this.state = new ReadyState(this)

    // El contexto delega la gestión de entradas del usuario
    // a un objeto de estado. Naturalmente, el resultado
    // depende del estado que esté activo ahora, ya que cada
    // estado puede gestionar las entradas de manera
    // diferente.
    UI = new UserInterface()
    UI.lockButton.onClick(this.clickLock)
    UI.playButton.onClick(this.clickPlay)
    UI.nextButton.onClick(this.clickNext)
    UI.prevButton.onClick(this.clickPrevious)

  // Otros objetos deben ser capaces de cambiar el estado
  // activo del reproductor.
  method changeState(state: State) is
    this.state = state

  // Los métodos UI delegan la ejecución al estado activo.
  method clickLock() is
    state.clickLock()
  method clickPlay() is
    state.clickPlay()
```

```
    method clickNext() is
      state.clickNext()
    method clickPrevious() is
      state.clickPrevious()

    // Un estado puede invocar algunos métodos del servicio en
    // el contexto.
    method startPlayback() is
      // ...
    method stopPlayback() is
      // ...
    method nextSong() is
      // ...
    method previousSong() is
      // ...
    method fastForward(time) is
      // ...
    method rewind(time) is
      // ...


// La clase estado base declara métodos que todos los estados
// concretos deben implementar, y también proporciona una
// referencia inversa al objeto de contexto asociado con el
// estado. Los estados pueden utilizar la referencia inversa
// para dirigir el contexto a otro estado.
abstract class State is
    protected field player: AudioPlayer

    // El contexto se pasa a sí mismo a través del constructor
    // del estado. Esto puede ayudar al estado a extraer
    // 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.
```

```
class ReadyState extends State is
  method clickLock() is
    player.changeState(new LockedState(player))

  method clickPlay() is
    player.startPlayback()
    player.changeState(new PlayingState(player))

  method clickNext() is
    player.nextSong()

  method clickPrevious() is
    player.previousSong()


class PlayingState extends State is
  method clickLock() is
    player.changeState(new LockedState(player))

  method clickPlay() is
    player.stopPlayback()
    player.changeState(new ReadyState(player))

  method clickNext() is
    if (event.doubleclick)
      player.nextSong()
    else
      player.fastForward(5)

  method clickPrevious() is
    if (event.doubleclick)
      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, céntrate en los que pueden contener comportamientos específicos del estado.

3. Para cada estado actual, crea una clase derivada de la interfaz de estado. Después 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.

www.detodoprogramacion.org

- 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.

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.

www.detodoprogramacion.org

✓ 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.

www.detodoprogramacion.org



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

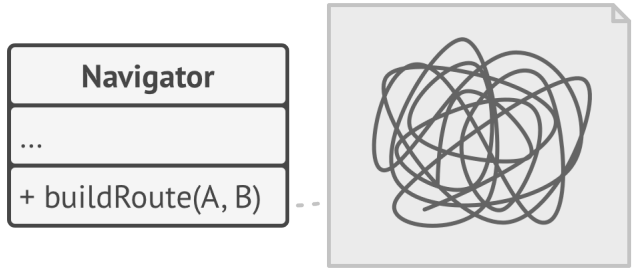
Un día decidiste crear una aplicación de navegación para viajeros ocasionales. La aplicación giraba alrededor de un bonito mapa que ayudaba a los usuarios a orientarse rápidamente en cualquier ciudad.

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.

www.detodoprogramacion.org



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.

www.detodoprogramacion.org

😊 Solución

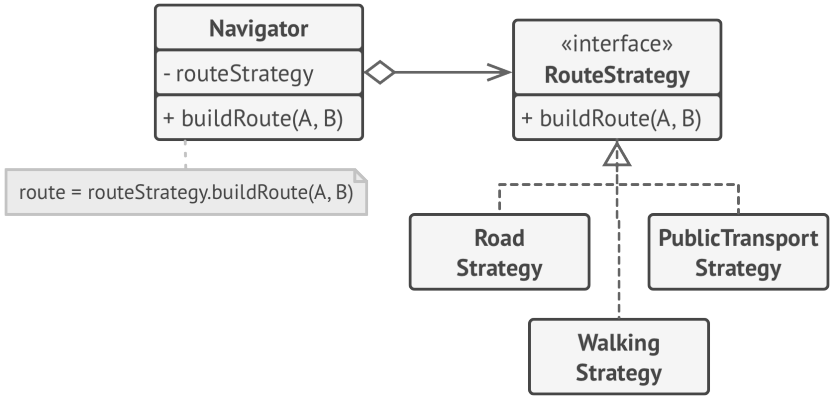
El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas *estrategias*.

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.

www.detodoprogramacion.org



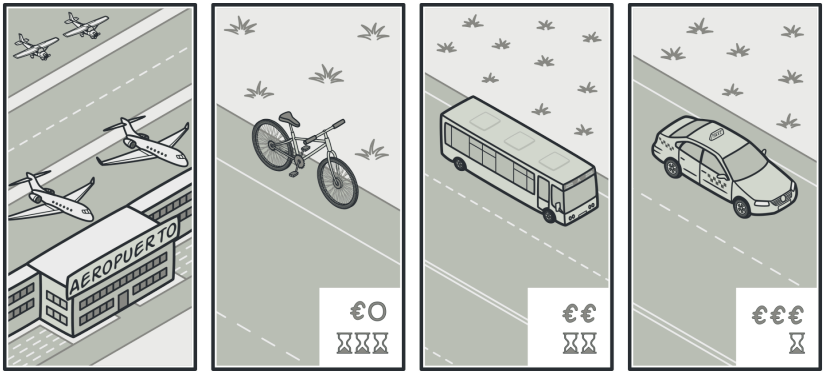
Estrategias de planificación de rutas.

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 navegadora 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.

www.detodoprogramacion.org

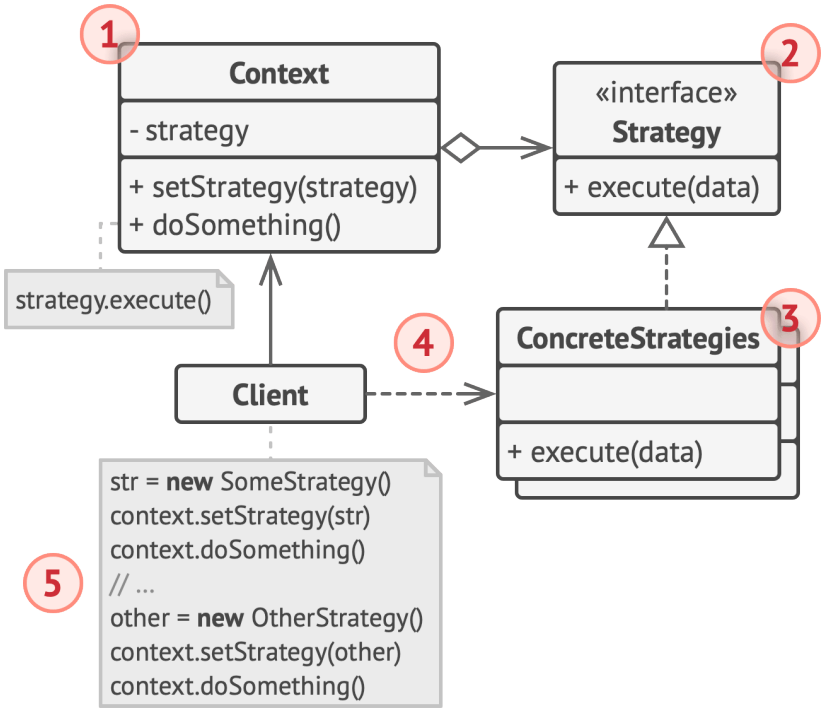
Analogía en el mundo real



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.

```
// La interfaz estrategia declara operaciones comunes a todas
// las versiones soportadas de algún algoritmo. El contexto
// utiliza esta interfaz para invocar el algoritmo definido por
// las estrategias concretas.
interface Strategy is
  method execute(a, b)

// Las estrategias concretas implementan el algoritmo mientras
// siguen la interfaz estrategia base. La interfaz las hace
// intercambiables en el contexto.
class ConcreteStrategyAdd implements Strategy is
  method execute(a, b) is
    return a + b

class ConcreteStrategySubtract implements Strategy is
```

```
    method execute(a, b) is
      return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
      return a * b

// El contexto define la interfaz de interés para los clientes.
class Context is
    // El contexto mantiene una referencia a uno de los objetos
    // de estrategia. El contexto no conoce la clase concreta de
    // una estrategia. Debe trabajar con todas las estrategias a
    // través de la interfaz estrategia.
    private strategy: Strategy

    // Normalmente, el contexto acepta una estrategia a través
    // del constructor y también proporciona un setter
    // (modificador) para poder cambiar la estrategia durante el
    // tiempo de ejecución.
    method setStrategy(Strategy strategy) is
      this.strategy = strategy

    // El contexto delega parte del trabajo al objeto de
    // estrategia en lugar de implementar varias versiones del
    // algoritmo por su cuenta.
    method executeStrategy(int a, int b) is
      return strategy.execute(a, b)


// El código cliente elige una estrategia concreta y la pasa al
// contexto. El cliente debe conocer las diferencias entre
// estrategias para elegir la mejor opción.
```

```
class ExampleApplication is
  method main() is
    Create context object.

    Read first number.
    Read last number.
    Read the desired action from user input.

    if (action == addition) then
      context.setStrategy(new ConcreteStrategyAdd())

    if (action == subtraction) then
      context.setStrategy(new ConcreteStrategySubtract())

    if (action == multiplication) then
      context.setStrategy(new ConcreteStrategyMultiply())

    result = context.executeStrategy(First number, Second number)

    Print result.
```

Aplicabilidad

Utiliza el patrón Strategy cuando quieras utiliza 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 asociá-

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.

www.detodoprogramacion.org

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.

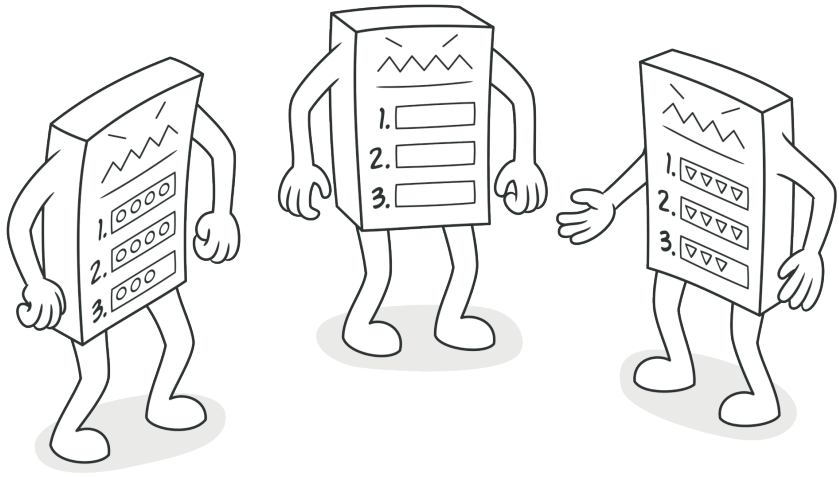
- **Template Method** se basa en la herencia: te permite alterar partes de un algoritmo extendiendo esas partes en subclases. **Strategy** se basa en la composición: puedes alterar partes del comportamiento del objeto suministrándole distintas estrategias que se correspondan con ese comportamiento. *Template Method* trabaja al nivel de la clase, por lo que es estático. *Stra-*

www.detodoprogramacion.org

tegy 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.

www.detodoprogramacion.org



TEMPLATE METHOD

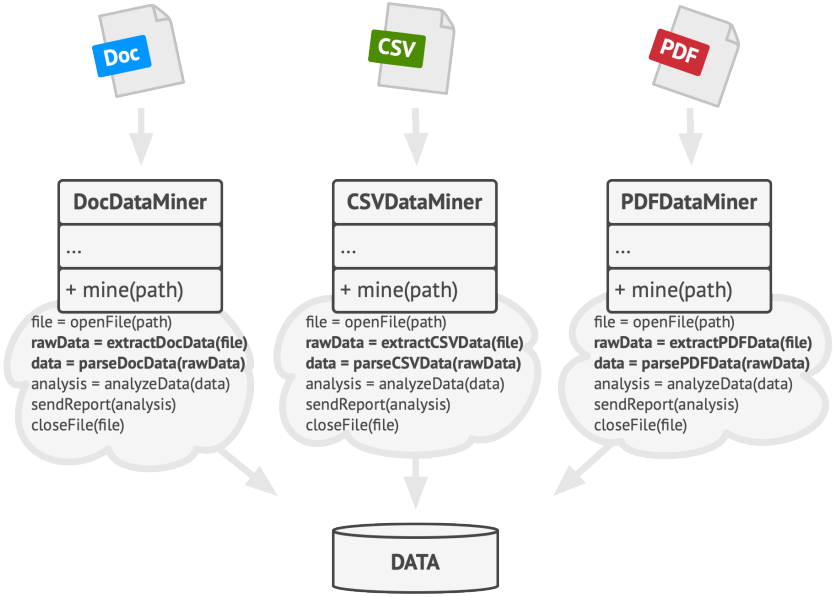
También llamado: Método plantilla

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?

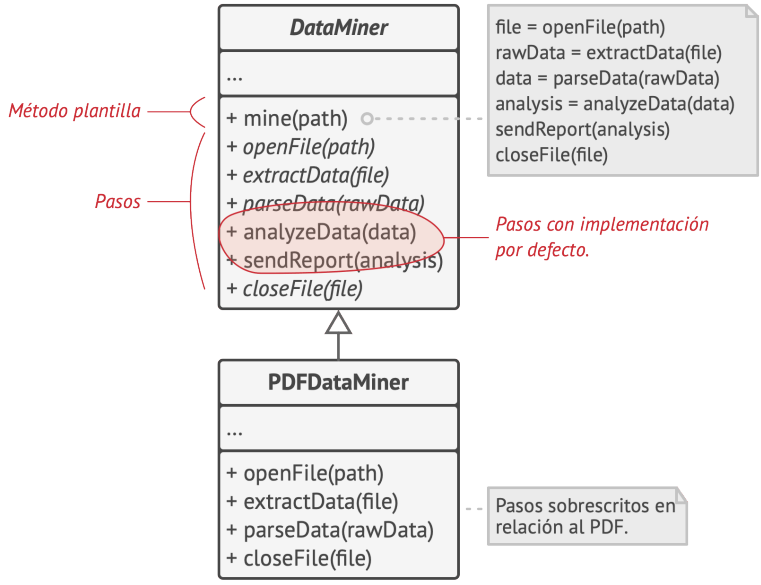
Hay otro problema relacionado con el código cliente que utiliza esas clases. Tiene muchos condicionales que eligen un curso de acción adecuado dependiendo de la clase del objeto de procesamiento. Si las tres clases de procesamiento tienen una interfaz común o una clase base, puedes eliminar los condicionales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

www.detodoprogramacion.org



El método plantilla divide el algoritmo en pasos, permitiendo a las subclases sobrescribir 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.

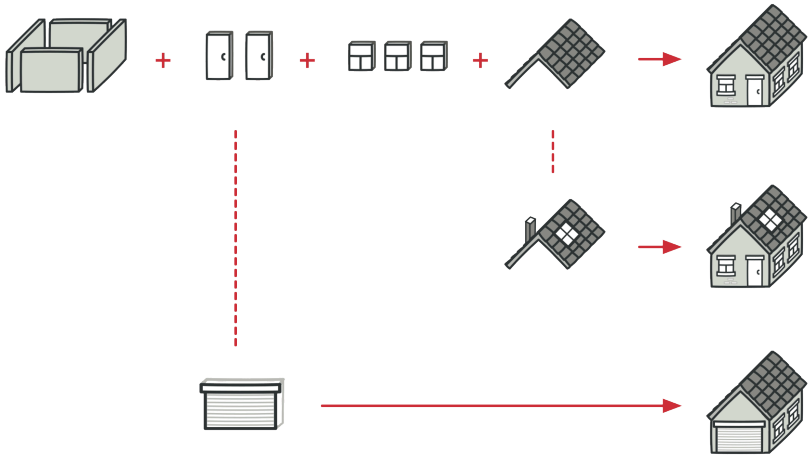
Como puedes ver, tenemos dos tipos de pasos:

- 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.

www.detodoprogramacion.org

Analogía en el mundo real

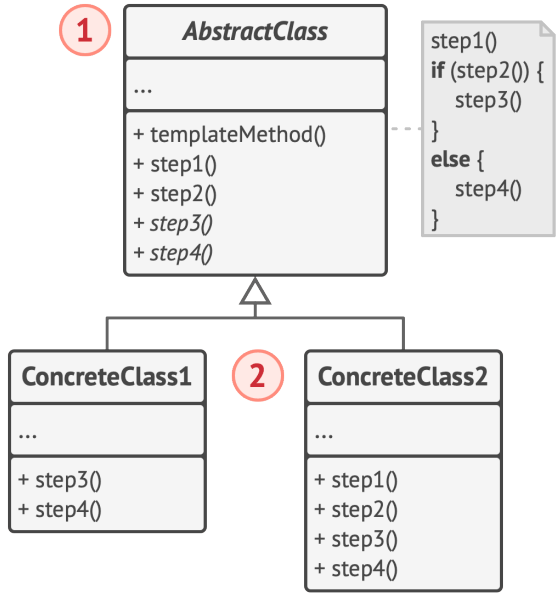


Un plan arquitectónico típico puede alterarse ligeramente para que encaje mejor con las necesidades del cliente.

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

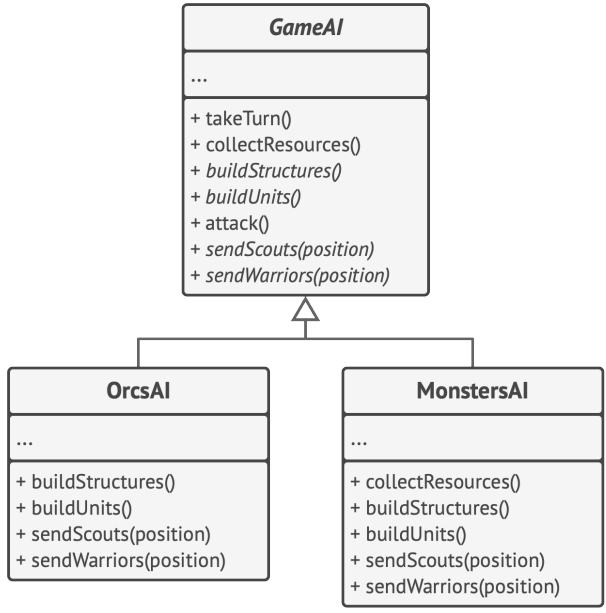


1. La **Clase Abstracta** declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse `abstractos` o contar con una implementación por defecto.

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.

```
// La clase abstracta define un método plantilla que contiene un
// esqueleto de algún algoritmo compuesto por llamadas,
// normalmente a operaciones primitivas abstractas. Las
// subclases concretas implementan estas operaciones, pero dejan
// el propio método plantilla intacto.
class GameAI is
  // El método plantilla define el esqueleto de un algoritmo.
  method turn() is
    collectResources()
    buildStructures()
    buildUnits()
    attack()

  // Algunos de los pasos se pueden implementar directamente
  // en una clase base.
  method collectResources() is
    foreach (s in this.builtStructures) do
      s.collect()

  // Y algunos de ellos pueden definirse como abstractos.
  abstract method buildStructures()
  abstract method buildUnits()

  // Una clase puede tener varios métodos plantilla.
  method attack() is
    enemy = closestEnemy()
    if (enemy == null)
      sendScouts(map.center)
```

```
    else
      sendWarriors(enemy.position)

  abstract method sendScouts(position)
  abstract method sendWarriors(position)

// Las clases concretas tienen que implementar todas las
// operaciones abstractas de la clase base, pero no deben
// sobrescribir el propio método plantilla.
class OrcsAI extends GameAI is
  method buildStructures() is
    if (there are some resources) then
      // Construye granjas, después cuarteles y después
      // fortaleza.

  method buildUnits() is
    if (there are plenty of resources) then
      if (there are no scouts)
        // Crea peón y añádelo al grupo de exploradores.
      else
        // Crea soldado, añádelo al grupo de guerreros.

  // ...

  method sendScouts(position) is
    if (scouts.length > 0) then
      // Envía exploradores a posición.

  method sendWarriors(position) is
    if (warriors.length > 5) then
      // Envía guerreros a posición.

```

```
// Las subclases también pueden sobrescribir algunas operaciones
// con una implementación por defecto.
class MonstersAI extends GameAI is
  method collectResources() is
    // Los monstruos no recopilan recursos.

  method buildStructures() is
    // Los monstruos no construyen estructuras.

  method buildUnits() is
    // 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 sobrescriban.

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.

www.detodoprogramacion.org

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

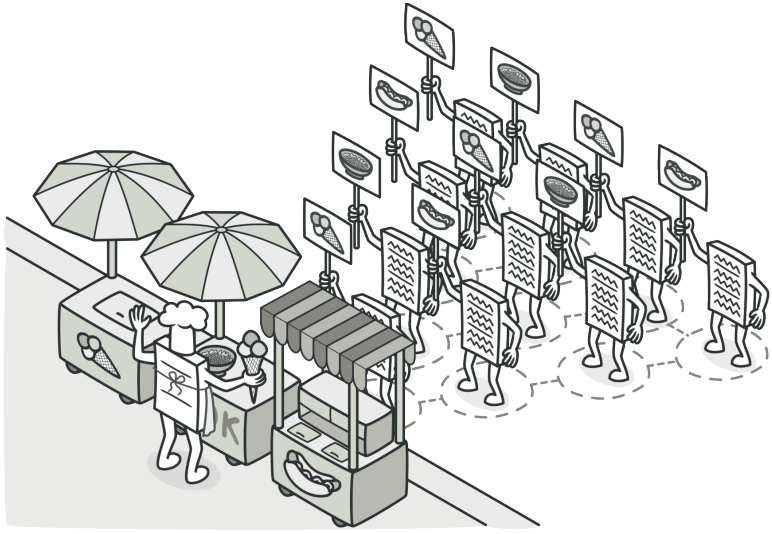
- **Factory Method** es una especialización del **Template Method**. Al mismo tiempo, un *Factory Method* puede servir como paso de un gran *Template Method*.

- **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

www.detodoprogramacion.org

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.

www.detodoprogramacion.org



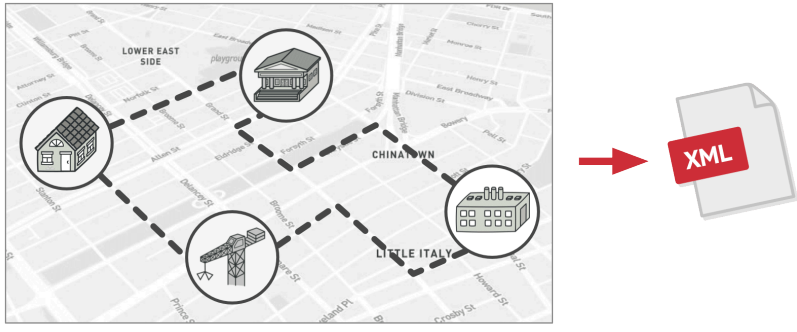
VISITOR

También llamado: Visitante

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

☹ Problema

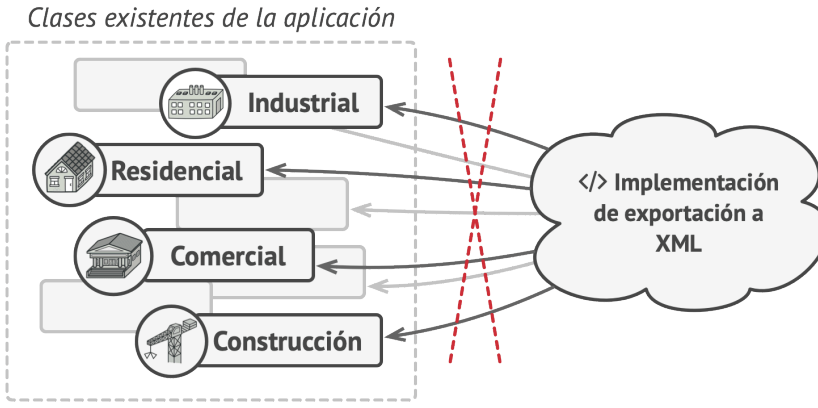
Imagina que tu equipo desarrolla una aplicación que funciona con información geográfica estructurada como un enorme grafo. Cada nodo del grafo puede representar una entidad compleja, como una ciudad, pero también cosas más específicas, como industrias, áreas turísticas, etc. Los nodos están conectados 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.

www.detodoprogramacion.org

😊 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:

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

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?

```
foreach (Node node in graph)
  if (node instanceof City)
    exportVisitor.doForCity((City) node)
  if (node instanceof Industry)
    exportVisitor.doForIndustry((Industry) node)
  // ...
}
```

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. Recurrirá 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.

```
// Código cliente
foreach (Node node in graph)
  node.accept(exportVisitor)

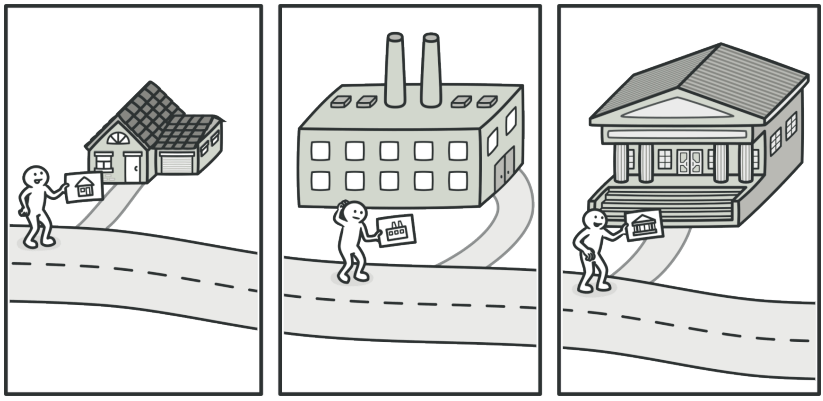
// Ciudad
class City is
  method accept(Visitor v) is
    v.doForCity(this)
  // ...

// Industria
class Industry is
  method accept(Visitor v) is
    v.doForIndustry(this)
  // ...
```

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

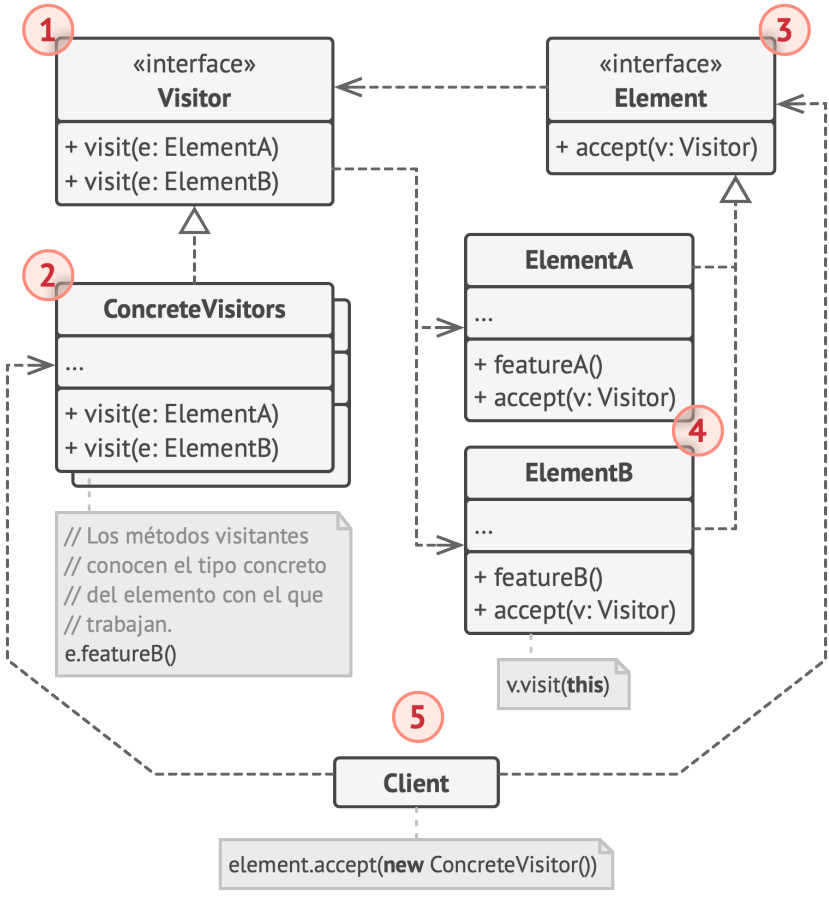


Un buen agente de seguros siempre está listo para ofrecer pólizas diferentes a los distintos tipos de organizaciones.

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.

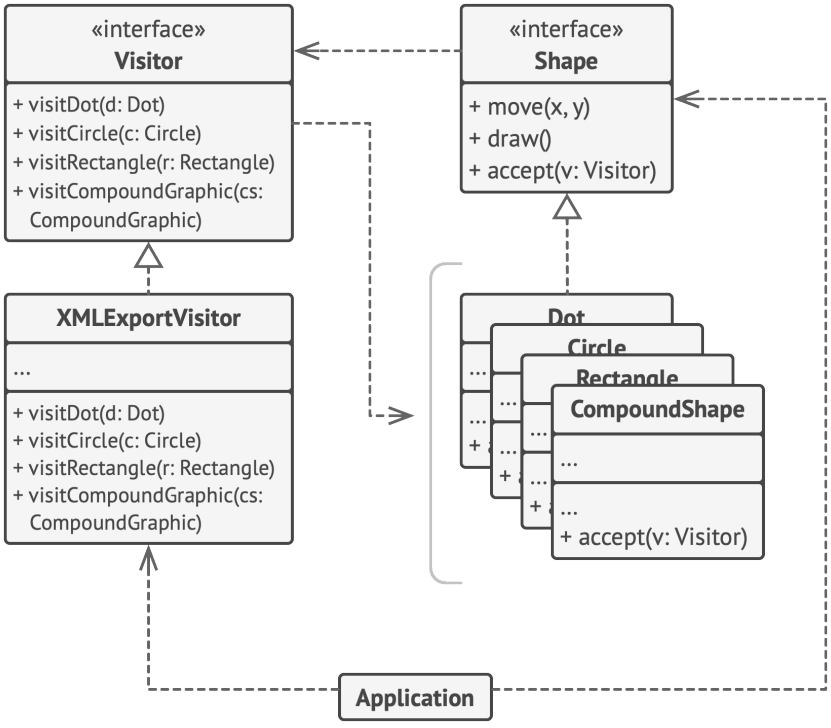
4. Cada **Elemento Concreto** debe implementar el método de aceptación. El propósito de este método es redirigir la llamada al método adecuado del visitante correspondiente a la clase de elemento actual. Piensa que, aunque una clase base de elemento implemente este método, todas las subclases deben sobrescribir 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.

www.detodoprogramacion.org



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

```
// La interfaz elemento declara un método `accept` (aceptar) que
// toma la interfaz visitante base como argumento.
interface Shape is
  method move(x, y)
  method draw()
  method accept(v: Visitor)

// Cada clase de elemento concreto debe implementar el método
// `accept` de tal manera que invoque el método del visitante
// que corresponde a la clase del elemento.
class Dot implements Shape is
```

```
  // ...

  // Observa que invocamos `visitDot`, que coincide con el
  // nombre de la clase actual. De esta forma, hacemos saber
  // al visitante la clase del elemento con el que trabaja.
  method accept(v: Visitor) is
    v.visitDot(this)

class Circle implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitCircle(this)

class Rectangle implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitRectangle(this)

class CompoundShape implements Shape is
  // ...
  method accept(v: Visitor) is
    v.visitCompoundShape(this)


// La interfaz Visitor declara un grupo de métodos de visita que
// se corresponden con clases de elemento. La firma de un método
// de visita permite al visitante identificar la clase exacta
// del elemento con el que trata.
interface Visitor is
  method visitDot(d: Dot)
  method visitCircle(c: Circle)
  method visitRectangle(r: Rectangle)
```

```
    method visitCompoundShape(cs: CompoundShape)

// Los visitantes concretos implementan varias versiones del
// mismo algoritmo, que puede funcionar con todas las clases de
// elementos concretos.
//
// Puedes disfrutar de la mayor ventaja del patrón Visitor si lo
// utilizas con una estructura compleja de objetos, como un
// árbol Composite. En este caso, puede ser de ayuda almacenar
// algún estado intermedio del algoritmo mientras ejecutas los
// métodos del visitante sobre varios objetos de la estructura.
class XMLExportVisitor implements Visitor is
    method visitDot(d: Dot) is
        // Exporta la ID del punto (dot) y centra las
        // coordenadas.

    method visitCircle(c: Circle) is
        // Exporta la ID del círculo y centra las coordenadas y
        // el radio.

    method visitRectangle(r: Rectangle) is
        // Exporta la ID del rectángulo, las coordenadas de
        // arriba a la izquierda, la anchura y la altura.

    method visitCompoundShape(cs: CompoundShape) is
        // Exporta la ID de la forma, así como la lista de las
        // ID de sus hijos.


// El código cliente puede ejecutar operaciones del visitante
// sobre cualquier grupo de elementos sin conocer sus clases
// concretas. La operación `accept` dirige una llamada a la
```

```
// operación adecuada del objeto visitante.
class Application is
  field allShapes: array of Shapes

  method export() is
    exportVisitor = new XMLExportVisitor()

    foreach (shape in allShapes) do
      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.

www.detodoprogramacion.org

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 pasarlos 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

www.detodoprogramacion.org

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.

www.detodoprogramacion.org

Conclusión

¡Felicidades! ¡Has llegado al final del libro!

Sin embargo, hay muchos otros patrones en el mundo. Espero que el libro se convierta en tu punto de partida para aprender patrones y desarrollar superpoderes para el diseño de programas.

Aquí tienes un par de ideas que te ayudarán a decidir qué hacer a continuación.

- </> No olvides que también tienes **acceso a un archivo** de muestras de código descargables en diferentes lenguajes de programación.

- Lee "**Refactoring To Patterns**" de Joshua Kerievsky.

- ¿No sabes nada de refactorización? **Tengo un curso para ti**.

- Imprime estas **hojas de referencia** y colócalas donde puedas verlas todo el tiempo.

- **Deja un comentario** sobre este libro. Me encantará conocer tu opinión, incluso si es altamente crítica ☺



El Mundo de la Programación en tus Manos...!

DETODOPROGRAMACION.ORG

DETODOPROGRAMACION.ORG

Material para los amantes de la Programación Java, C/C++/C#,Visual.Net, SQL, Python, Javascript, Oracle, Algoritmos, CSS, Desarrollo Web, Joomla, jquery, Ajax y Mucho Mas…

VISITA

www.detodoprogramacion.org
www.detodopython.com
www.gratiscodigo.com

GRATISCODIGO.COM
Usa, lo que ya se creó...

Detodo Python.com