Ingeniería de Software II

Patrones de diseño de Comportamiento

2023

Facultad de Ciencia y Tecnología
Universidad Autónoma de Entre Ríos

Sumérgete en los patrones de diseño

Alexander Shvets, Refactoring.Guru, 2019

support@refactoring.guru

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

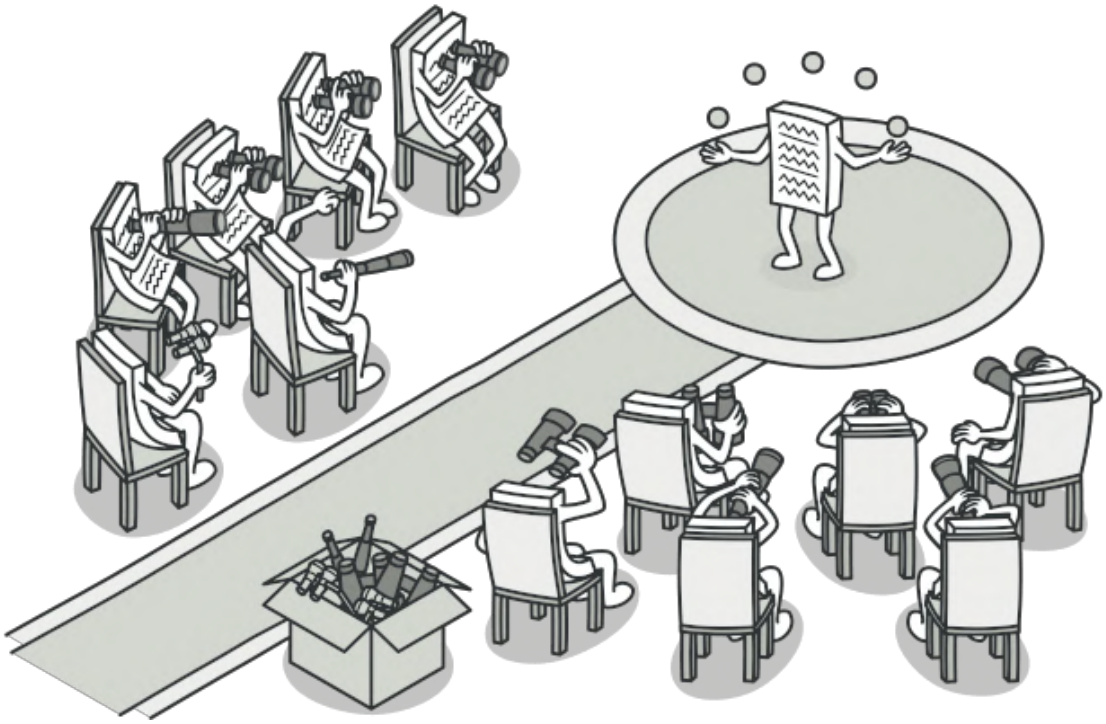
PATRONES DE COMPORTAMIENTO

Observer

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

Propósito

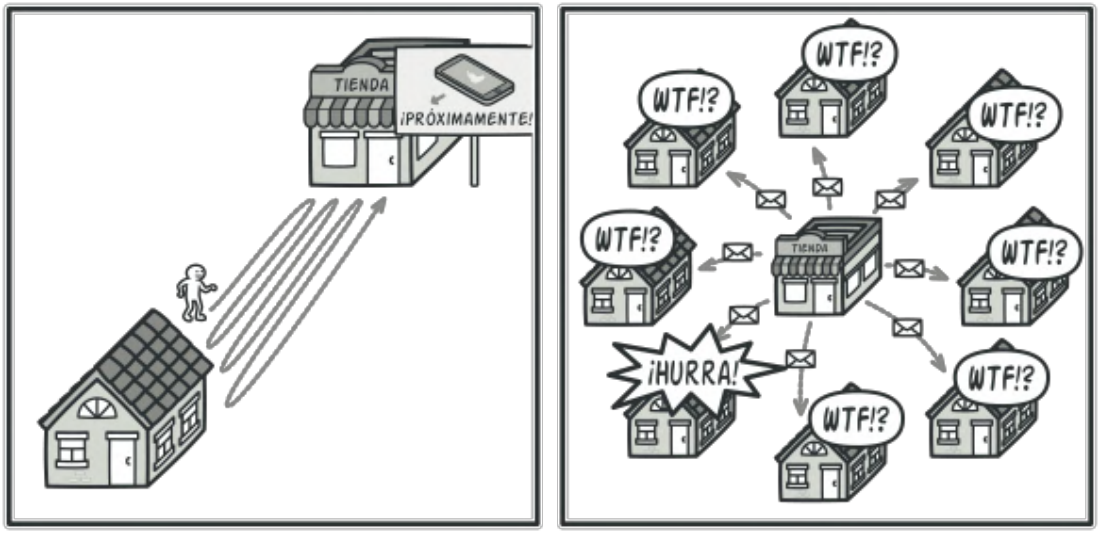
Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



Problema

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

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



Visita a la tienda vs. envío de spam

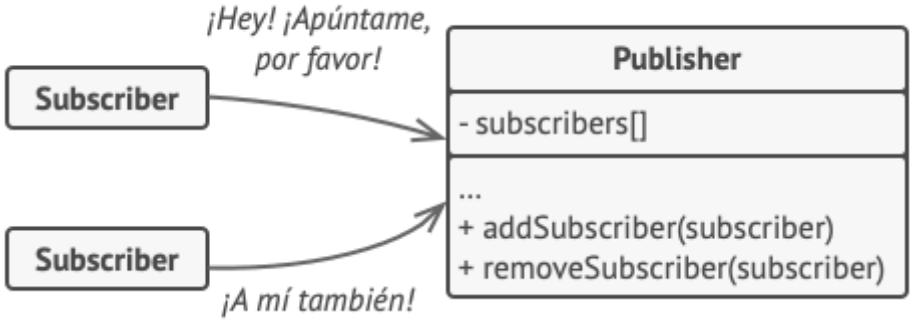
Por otro lado, la tienda podría enviar cientos de correos (lo cual se podría considerar spam) a todos los clientes cada vez que hay un nuevo producto disponible. Esto ahorraría a los clientes los interminables viajes a la tienda, pero, al mismo tiempo, molestaría a otros clientes que no están interesados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

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

Solución

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

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

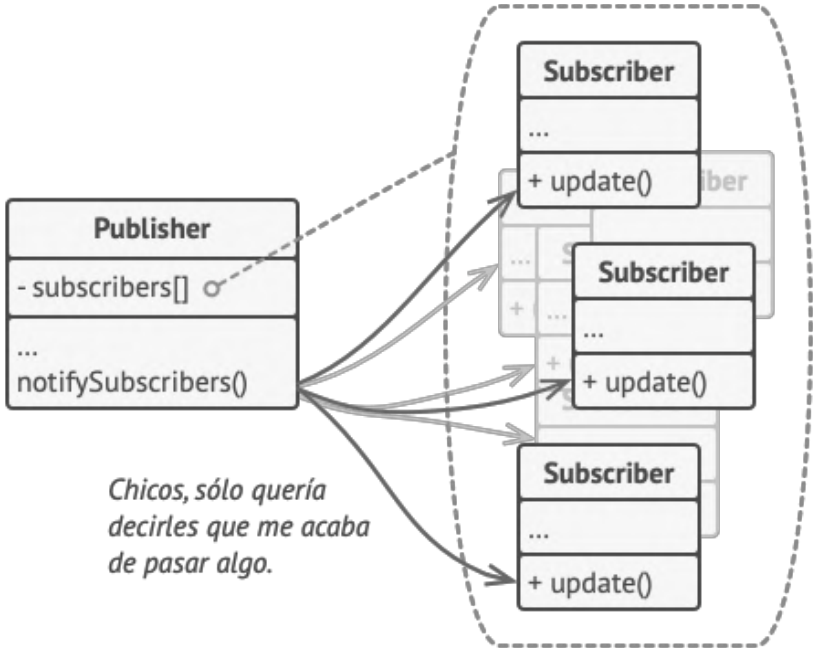


Un mecanismo de suscripción permite a los objetos individuales suscribirse a notificaciones de eventos.

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

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

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



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

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

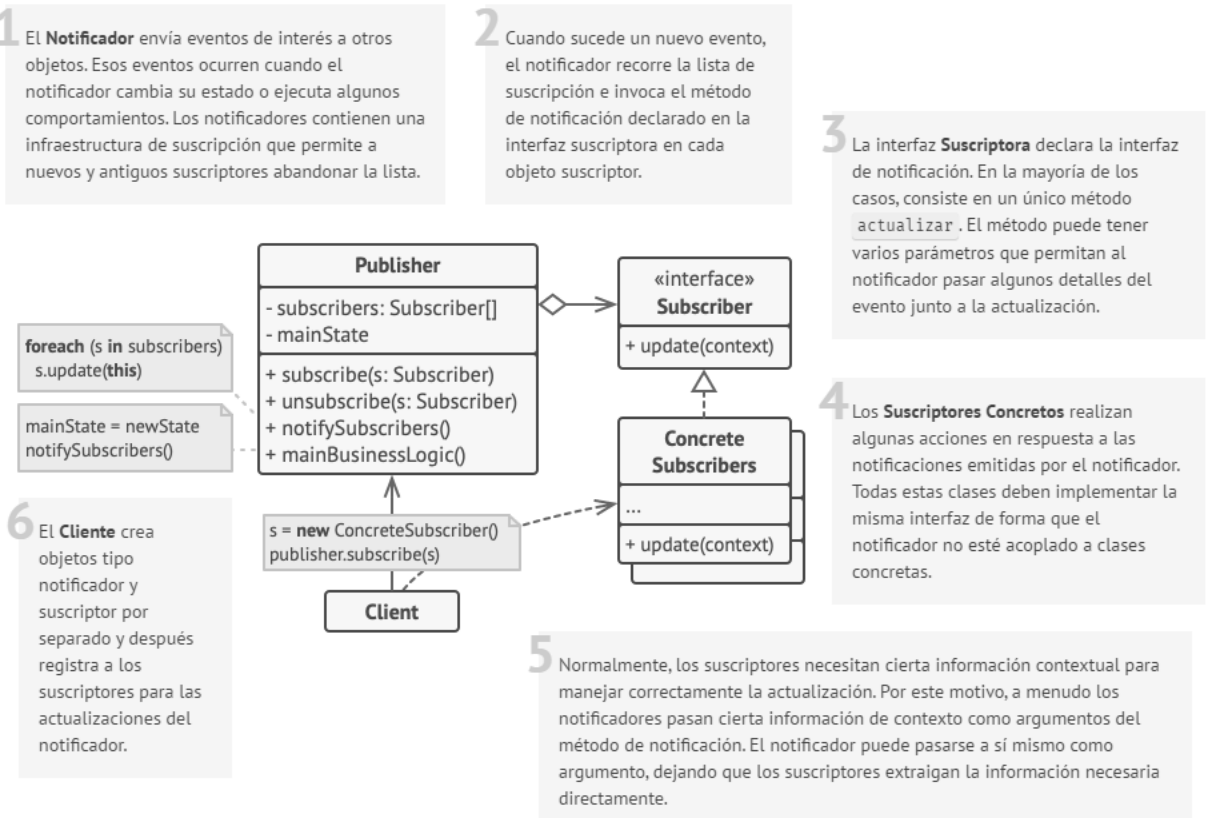
Analogía en el mundo real

Suscripciones a revistas y periódicos.

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

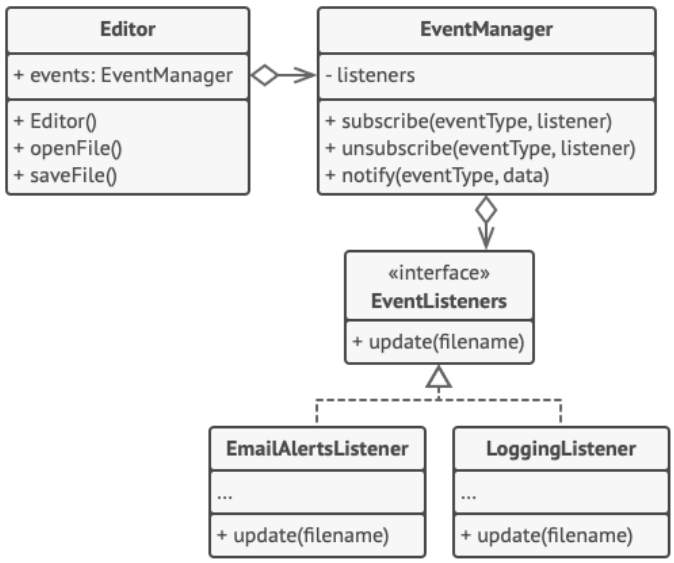
El notificador mantiene una lista de suscriptores y sabe qué revistas les interesan. Los suscriptores pueden abandonar la lista en cualquier momento si quieren que el notificador deje de enviarles nuevos números.

Estructura



Pseudocódigo

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



Notificar a objetos sobre eventos que suceden a otros objetos.

La lista de suscriptores se compila dinámicamente: los objetos pueden empezar o parar de escuchar notificaciones durante el tiempo de ejecución, dependiendo del comportamiento que 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()

    // Los métodos de la lógica de negocio pueden notificar los
    // cambios a los suscriptores.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)

    // ...


// Aquí está la interfaz suscriptora. Si tu lenguaje de
// programación soporta tipos funcionales, puedes sustituir toda
// la jerarquía suscriptora por un grupo de funciones.


interface EventListener is
    method update(filename)

// Los suscriptores concretos reaccionan a las actualizaciones
// emitidas por el notificador al que están unidos.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: 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.

Implementación

1. Repasa tu lógica de negocio e intenta dividirla en dos partes: la funcionalidad central, independiente del resto de código, actuará como notificador; el resto se convertirá en un grupo de clases suscriptoras.
2. Declara la interfaz suscriptora. Como mínimo, deberá declarar un único método `actualizar`.
3. Declara la interfaz notificadora y describe un par de métodos para añadir y eliminar de la lista un objeto suscriptor. Recuerda que los notificadores deben trabajar con suscriptores únicamente a través de la interfaz suscriptora.
4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción. Normalmente, este código tiene el mismo aspecto para todos los tipos de notificadores, por lo que el lugar obvio para colocarlo es en una clase abstracta derivada directamente de la interfaz notificadora. Los notificadores concretos extienden esa clase, heredando el comportamiento de suscripción.
   Sin embargo, si estás aplicando el patrón a una jerarquía de clases existentes, considera una solución basada en la composición: coloca la lógica de la suscripción en un objeto separado y haz que todos los notificadores reales la utilicen.
5. Crea clases notificadoras concretas. Cada vez que suceda algo importante dentro de una notificadora, deberá notificar a todos sus suscriptores.
6. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas. La mayoría de las suscriptoras necesitarán cierta información de contexto sobre el evento, que puede pasarse como argumento del método de notificación.
   Pero hay otra opción. Al recibir una notificación, el suscriptor puede extraer la información directamente de ella. En este caso, el notificador debe pasarse a sí mismo a través del método de actualización. La opción menos flexible es vincular un notificador con el suscriptor de forma permanente a través del constructor.
7. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

Pros y contras

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

Relaciones con otros patrones

- **Chain of Responsibility**, **Command**, **Mediator** y **Observer** abordan distintas formas de conectar emisores y receptores de solicitudes:
  - *Chain of Responsibility* pasa una solicitud secuencialmente a lo largo de una cadena dinámica de receptores potenciales hasta que uno de ellos la gestiona.

- *Command* establece conexiones unidireccionales entre emisores y receptores.
- *Mediator* elimina las conexiones directas entre emisores y receptores, forzándolos a comunicarse indirectamente a través de un objeto mediador.
- *Observer* permite a los receptores suscribirse o darse de baja dinámicamente a la recepción de solicitudes.

- La diferencia entre **Mediator** y **Observer** a menudo resulta difusa. En la mayoría de los casos, puedes implementar uno de estos dos patrones; pero en ocasiones puedes aplicarlos ambos a la vez. Veamos cómo podemos hacerlo.

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

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

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

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

Strategy

También llamado: Estrategia

Propósito

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

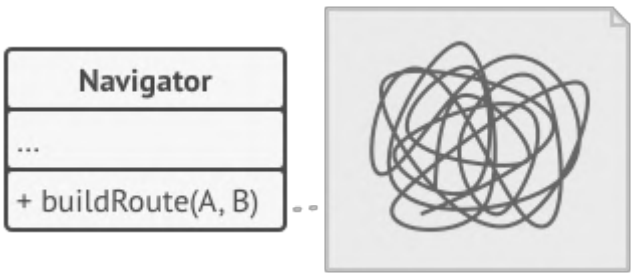
Problema

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

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

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

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



El código del navegador se saturó.

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

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

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

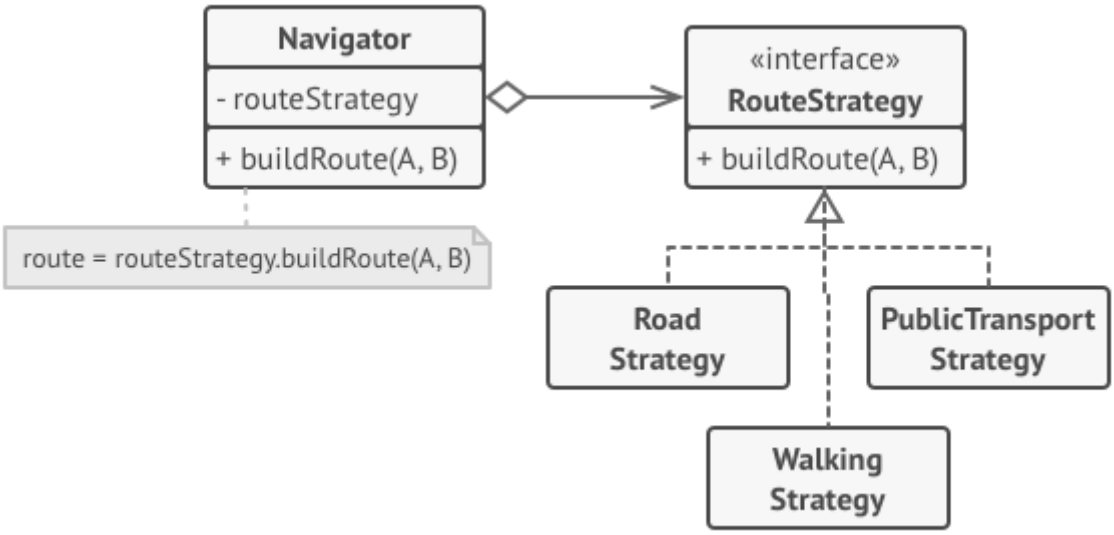
Solución

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

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

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

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

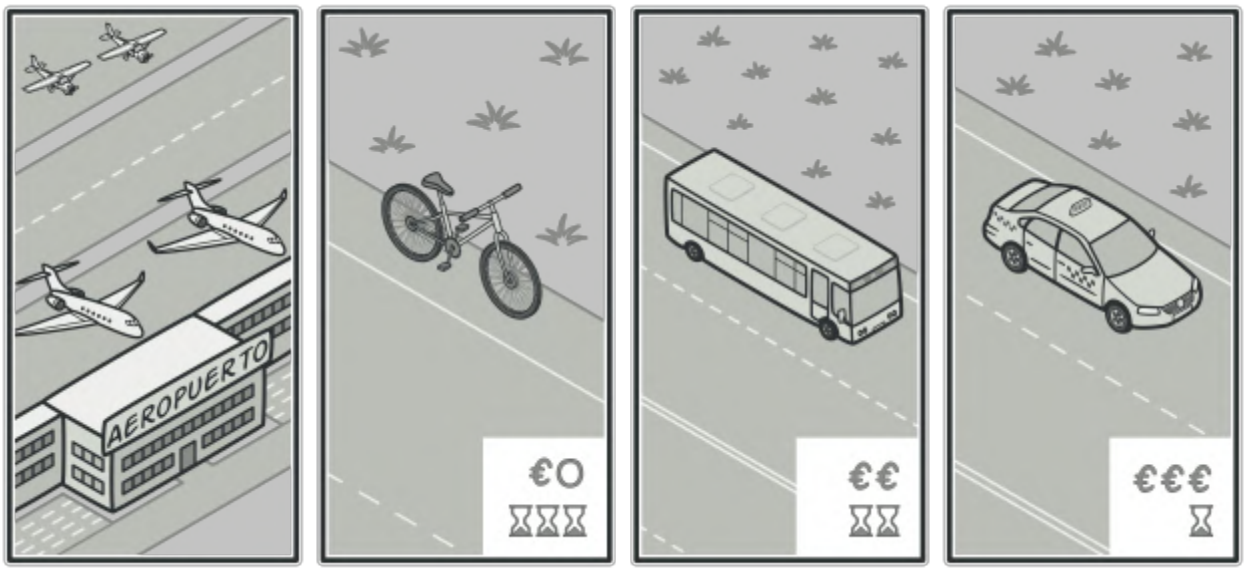


Estrategias de planificación de rutas.

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

Incluso contando con los mismos argumentos, cada clase de enrutamiento puede crear una ruta diferente. A la clase 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.

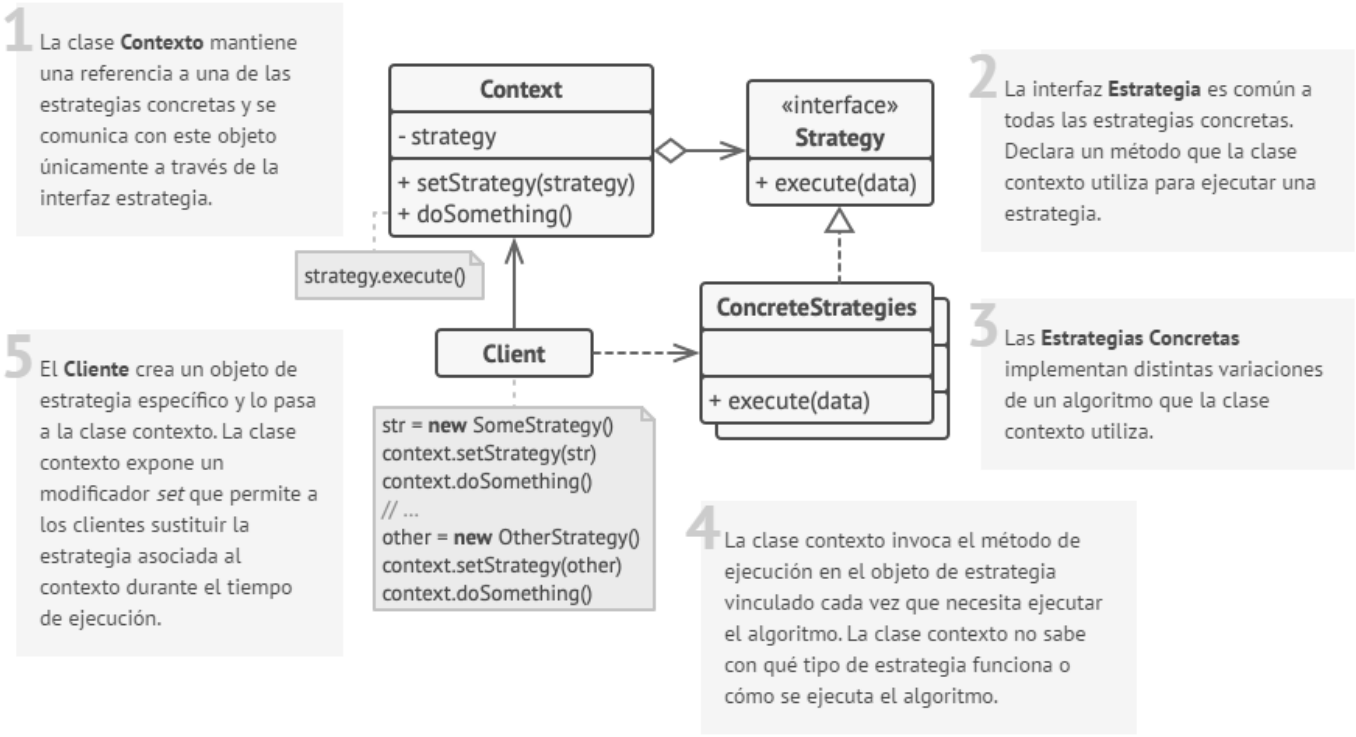
Analogía en el mundo real



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



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 asociándolo 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.

Implementación

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

Pros y contras

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

Relaciones con otros patrones

- **Bridge**, **State**, **Strategy** (y, hasta cierto punto, **Adapter**) tienen estructuras muy similares. De hecho, todos estos patrones se basan en la composición, que consiste en delegar trabajo a otros objetos. Sin embargo, todos ellos solucionan problemas diferentes. Un patrón no es simplemente una receta para estructurar tu código de una forma específica. También puede comunicar a otros desarrolladores el problema que resuelve.
- **Command** y **Strategy** pueden resultar similares porque puedes usar ambos para parametrizar un objeto con cierta acción. No obstante, tienen propósitos muy diferentes.
    - Puedes utilizar *Command* para convertir cualquier operación en un objeto. Los parámetros de la operación se convierten en campos de ese objeto. La conversión te permite aplazar la ejecución de la operación, ponerla en cola, almacenar el historial de comandos, enviar comandos a servicios remotos, etc.
    - Por su parte, *Strategy* normalmente describe distintas formas de hacer lo mismo, permitiéndote intercambiar estos algoritmos dentro de una única clase contexto.
- **Decorator** te permite cambiar la piel de un objeto, mientras que **Strategy** te permite cambiar sus entrañas.
- **Template Method** se basa en la herencia: te permite alterar partes de un algoritmo extendiendo esas partes en subclases. **Strategy** se basa en la composición: puedes alterar partes del comportamiento del objeto suministrándole distintas estrategias que se correspondan con ese comportamiento. *Template Method* trabaja al nivel de la clase, por lo que es estático. *Strategy* 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.