**Adapter**

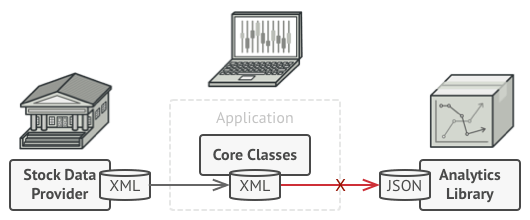
## **Intención**

**El adaptador** es un patrón de diseño estructural que permite que los objetos con interfaces incompatibles colaboren.

## **Problema**

Imagina que tienes una aplicación que funciona con datos en formato XML, pero en algún momento necesitas usar una biblioteca que solo puede funcionar con JSON.

Por ejemplo, su aplicación funciona con un stock de datos. Descarga datos de múltiples proveedores de datos en XML y crea gráficos de aspecto agradable. En una nueva versión, decidió mejorar la aplicación y usar alguna biblioteca de análisis inteligente. Pero hay un problema: la biblioteca de análisis solo funciona con datos en formato JSON.



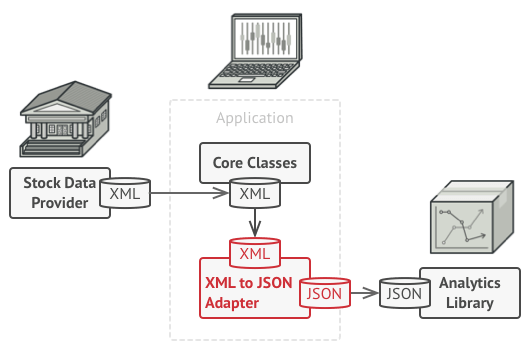
En casos como este, puede volver a escribir su código para admitir JSON o cambiar la biblioteca para que funcione con XML. La primera opción puede romper su código existente, y la segunda podría ser simplemente imposible si el código de la biblioteca no está disponible.

## **Solución**

Puede crear un adaptador. Es un objeto especial que convierte las llamadas enviadas por un objeto al formato que otro objeto puede entender. El adaptador envuelve uno de los objetos para ocultar la complejidad de la conversión que ocurre detrás de la escena.

Los adaptadores pueden convertir no solo formatos de datos sino también interfaces. Por ejemplo, el adaptador recibe una llamada al método A y luego lo pasa a los métodos B, C y D de un objeto envuelto.

A veces incluso es posible crear adaptadores bidireccionales que convierten las llamadas en ambas direcciones.



Por lo tanto, la aplicación del mercado de valores que discutimos anteriormente obtendrá una XML\_To\_JSON\_Adapter clase especial. Convierte todos los datos entrantes de XML a JSON antes de pasarlos a los métodos de un objeto envuelto de la biblioteca de análisis. De esta manera, no necesitará cambiar ningún código existente: ni las clases de la aplicación, ni la biblioteca de análisis.

## **Analogía del mundo real**

**Enchufes y tomas de corriente en diferentes países.**

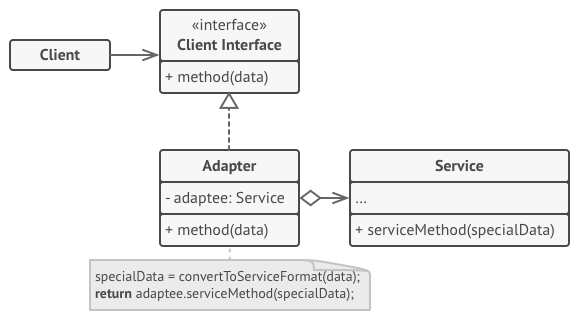
Cuando vuele desde EE. UU. A Europa por primera vez, es posible que se sorprenda cuando intente cargar su computadora portátil. Los estándares de enchufes y tomas de corriente varían según los países. Es por eso que su enchufe de EE. UU. No encaja en el conector alemán.

El problema se puede resolver con un adaptador de corriente que tiene un enchufe de estilo americano y un enchufe de estilo europeo.

## **Estructura**

### **Adaptador de objeto**

Esta implementación utiliza el principio de composición: el adaptador implementa la interfaz de un objeto y envuelve el otro. Se puede implementar en todos los lenguajes de programación principales.



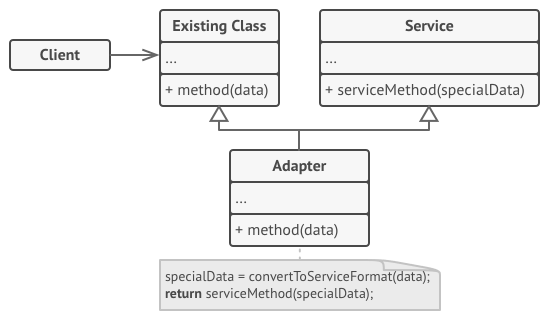
1. **Interfaz** o clase **existente** que ya es compatible con el resto de su código.
2. **El servicio** es una clase útil (generalmente de terceros o heredada) que no puede trabajar directamente con las clases de la aplicación.
3. **El adaptador** implementa la interfaz existente y contiene una referencia a un objeto de la clase de servicio.

El adaptador recibe llamadas del código del cliente a través de los métodos de la interfaz existente. Puede corregir el tipo o formato de los datos, pasados ​​a través de los parámetros, antes de pasarlos a los métodos del objeto Servicio.

1. **El cliente** utiliza el objeto adaptador a través de una interfaz existente. Esto permite agregar nuevas clases de Adaptador al programa sin cambiar el código existente (esta situación puede surgir cuando la clase de Servicio cambia, por ejemplo, cuando actualiza la biblioteca de terceros).

### **Adaptador de clase**

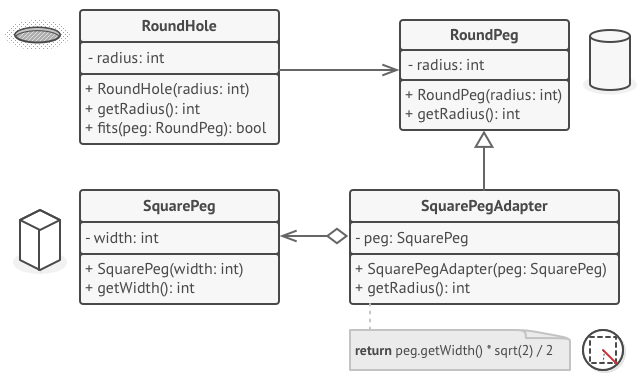
Esta implementación usa herencia. Adaptador hereda ambas interfaces al mismo tiempo. Solo se puede implementar en idiomas que admitan herencia múltiple, como C ++.



1. El adaptador de clase no necesita envolver ningún objeto. Implementa ambas interfaces al mismo tiempo. Por lo tanto es compatible con ambos objetos.

## **Pseudocódigo**

Veamos cómo el adaptador puede convertir una interfaz en otra con una conversión de datos básica. El ejemplo se basa en el conflicto entre los agujeros redondos y las clavijas cuadradas. Los orificios redondos funcionan muy bien con clavijas redondas; puede determinar si uno se ajusta o no a otro simplemente comparando sus radios. Pero las clavijas cuadradas no tienen la medida del radio en absoluto.



Es por eso que creamos una clase de Adaptador que envuelve objetos de clavija cuadrada y simula que tiene un radio, que equivale a la mitad del diámetro del cuadrado (en otras palabras, el radio mínimo del círculo que puede contener ese cuadrado).

// Classes with compatible interfaces: RoundHole and RoundPeg.

**class** **RoundHole** **is**

**constructor** RoundHole(radius) { ... }

**method** getRadius **is**

// Return the radius of the hole.

**method** fits(peg: RoundPeg) **is**

**return** **this**.getRadius() >= peg.radius()

**class** **RoundPeg** **is**

**constructor** RoundPeg(radius) { ... }

**method** getRadius() **is**

// Return the peg radius.

// Obsolete incompatible class: SquarePeg.

**class** **SquarePeg** **is**

**constructor** SquarePeg(width) { ... }

**method** getWidth() **is**

// Return the square peg width.

// Adapter allows fitting square pegs into round holes.

**class** **SquarePegAdapter** **extends** RoundPeg **is**

**private** **field** peg: SquarePeg

**constructor** SquarePegAdapter(peg: SquarePeg) **is**

**this**.peg = peg

**method** getRadius() **is**

// Get half of the diagonal's length using the

// Pythagorean equation.

**return** peg.getWidth() \* Math.sqrt(2) / 2

// Somewhere in client code.

hole = **new** RoundHole(5)

rpeg = **new** RoundPeg(5)

hole.fits(rpeg) // true

small\_sqpeg = **new** SquarePeg(2)

large\_sqpeg = **new** SquarePeg(5)

hole.fits(small\_sqpeg) // this will not compile (incompatible types)

small\_sqpeg\_adapter = **new** SquarePegAdapter(small\_sqpeg)

large\_sqpeg\_adapter = **new** SquarePegAdapter(large\_sqpeg)

hole.fits(small\_sqpeg\_adapter) // true

hole.fits(large\_sqpeg\_adapter) // false

## **Aplicabilidad**

**Cuando desea reutilizar la clase existente, pero su interfaz no es compatible con el resto del código de la aplicación.**

 El patrón del adaptador crea una clase de capa intermedia que traduce las llamadas de la aplicación al formulario que la clase existente entiende.

**Debe reutilizar varias clases existentes, pero carecen de alguna funcionalidad común. Y no puede agregarlo a la superclase porque está cerrado o se usa en otro código.**

Podría poner la funcionalidad que falta en una nueva clase de adaptador. Conectará el código de su aplicación y las clases en las que está interesado. Esta solución es muy similar al patrón de [**visitante**](https://refactoring.guru/design-patterns/visitor) .

## **Cómo implementar**

1. Asegúrate de que tienes dos actores:
   * Objetos de servicio útiles.
   * Código de aplicación que debe utilizar objetos de servicio. La aplicación no debe poder usar los objetos de servicio directamente, debido a interfaces o formatos de datos incompatibles.
2. Declare la *interfaz de cliente* que seguirá la futura clase de adaptador. La aplicación utilizará esta interfaz para comunicarse con un adaptador.
3. Cree una clase de adaptador, haga que implemente la *interfaz de cliente,* pero deje todos los métodos vacíos por ahora.
4. Agregue un campo a la clase de adaptador que almacenará una referencia a un objeto de servicio. En la mayoría de los casos, este campo obtiene el valor en un constructor. En casos más simples, el adapter se puede pasar directamente a los métodos del adaptador.
5. Uno por uno, implementar todos los métodos de la interfaz del cliente en la clase de adaptador. Estos métodos deben pasar la ejecución a los métodos apropiados del objeto de servicio, mientras se convierten los datos pasados ​​a un formato adecuado.
6. Una vez que la clase de adaptador esté lista, úsela en el código de la aplicación a través de la *interfaz del cliente*.

## **Pros y contras**

### **Pros**

Oculta del código del cliente detalles de implementación innecesarios de la interfaz y la conversión de datos.

### **Contra**

 Aumenta la complejidad general del código creando clases adicionales.

# Referencia

# https://refactoring.guru/design-patterns/adapter