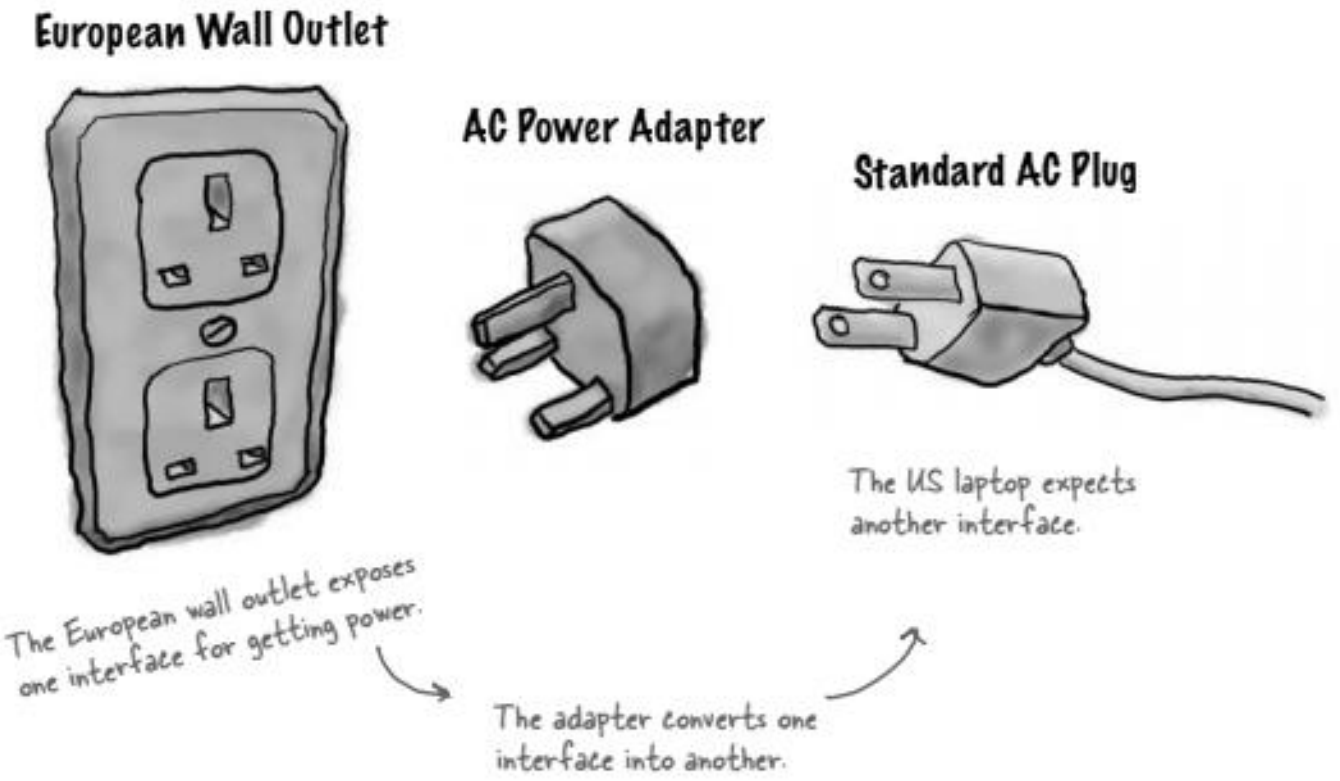
DISEÑO Y ARQUITECTURA DE SOFTWARE

ADAPTER PATTERN

PATRÓN DE DISEÑO ESTRUCTURAL

Adapter pattern



Se clasifica dentro de los patrones de diseño estructural.

Tratan sobre la composición de clases y objetos.

Usan herencia para componer interfaces.

Definen maneras de componer un objeto para obtener nuevas funcionalidades.

INTENCIÓN

- CONVIERTE LA INTERFAZ DE UNA CLASE EN OTRA INTERFAZ QUE EL CLIENTE ESPERA.
- EL ADAPTADOR PERMITE A LAS CLASES TRABAJAR JUNTAS, LO QUE DE OTRA MANERA NO PODRÍA HACERSE DEBIDO A SUS INTERFACES INCOMPATIBLES.

OTROS NOMBRES

WRAPPER

MOTIVACIÓN

Es muy frecuente la necesidad de adaptadores para elementos de la vida cotidiana: cargadores de baterías, tipos de enchufe, etc.

Si traspasamos esta visión al mundo del software, en algunas ocasiones un conjunto de clases no es reutilizable simplemente por la interfaz que no concuerda con el dominio específico que una aplicación requiere.

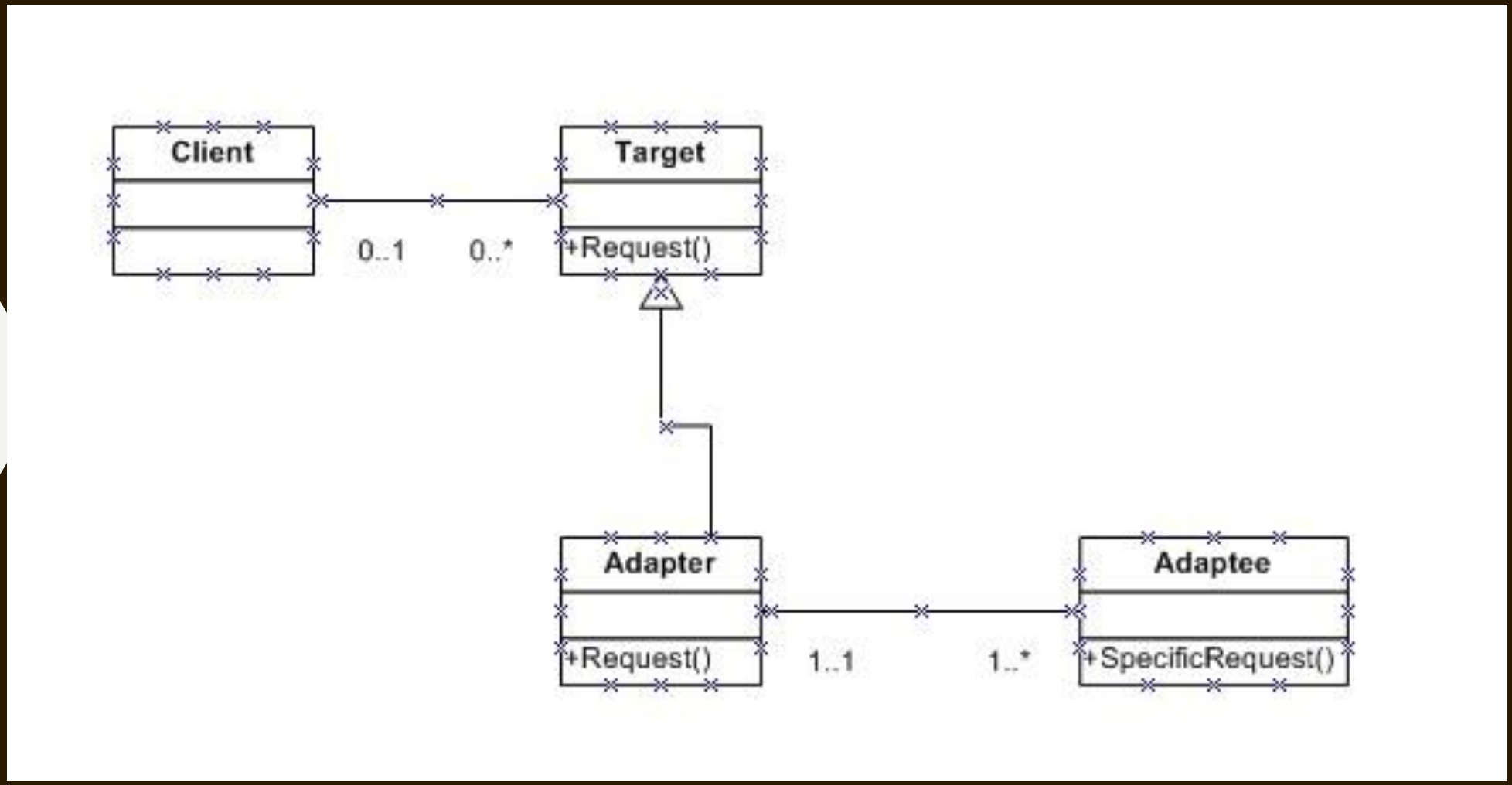
Es necesario crear un patrón que facilite esta reutilización y que permita no modificar la estructura de códigos del cliente y del servicio.

APLICABILIDAD

Es recomendable utilizar el patrón adaptador cuando:

- Se desea usar una clase existente, y su interfaz no sea igual a la necesitada.
- Se desea crear una clase reutilizable que coopere con clases no relacionadas. Es decir, que las clases no tienen necesariamente interfaces compatibles.

ESTRUCTURA


Client
Target
+Request()
0..1
0..*
Adapter
+Request()
Adaptee
+SpecificRequest()
1..1
1..*

PARTICIPANTES

- ***Target:*** Define la interfaz específica del dominio que *Client* usa.
- ***Client***: Colabora con la conformación de objetos para la interfaz *Target*.
- ***Adaptee:*** Define una interfaz existente que necesita adaptarse.
- ***Adapter:*** Adapta la interfaz de *Adaptee* a la interfaz *Target*.

COLABORACIONES

- **Client** llama a las operaciones sobre una instancia Adapter. De hecho, el adaptador llama a las operaciones de Adaptee que llevan a cabo el pedido.

CONSECUENCIAS

Un adaptador de clase:

- Adapta *Adaptee* a *Target* encargando a una clase *Adaptee* concreta. Como consecuencia, una clase adaptadora no funcionará cuando se desea adaptar una clase y todas sus subclases.
- Permite a los *Adapter* sobreescribir algo de comportamiento de *Adaptee*, ya que *Adapter* es una subclase de *Adaptee*.

CONSECUENCIAS

Un adaptador de objeto:

- Permite que un único *Adapter* trabaje con muchos *Adaptees*, es decir, el *Adapter* por sí mismo y las subclases (si es que la tiene). El *Adapter* también puede agregar funcionalidad a todos los *Adaptees* de una sola vez.
- Hace difícil sobrescribir el comportamiento de *Adaptee*. Esto requerirá derivar *Adaptee* y hacer que *Adapter* se refiera a la subclase en lugar que al *Adaptee* por sí mismo.

IMPLEMENTACIÓN

- Crear una nueva clase que será el Adaptador, que extienda del componente existente e implemente la interfaz obligatoria. De este modo se tiene la funcionalidad que se quería y se cumple la condición de implementar la interfaz.

- La diferencia entre los patrones adaptador y fachada (facade) es que el primero reutiliza una interfaz ya existente, mientras que el segundo define una nueva con el objetivo de simplificarla.

EJEMPLO DE CÓDIGO

```
# Example #1: Language Translator


class EnglishSpeaker:
        def responseToGreeting(self):
                return "Hello to you too!"
        def responseToFarewell(self):
                return "Goodbye my friend.“


class Translator:
        _englishSpeaker = None
        _englishToFrenchPhrases = {
                "Hello to you too!": "Bonjour à vous aussi",
                "Goodbye my friend.": "Au revoir mon ami"
                }


        def __init__(self, englishSpeaker):
                self._englishSpeaker = englishSpeaker
```

```
class  FrenchSpeaker:
        _englishToFrenchTranslator = None

        def __init__(self, englishToFrenchTranslator):
            self._englishToFrenchTranslator = englishToFrenchTranslator

        def exchangeGreetings(self):
            print("Salut!")
            print( self._englishToFrenchTranslator._englishToFrenchPhrases[
self._englishToFrenchTranslator._englishSpeaker.responseToGreeting()  ] )

        def exchangeFarewell(self):
            print("Au revoir!")
            print( self._englishToFrenchTranslator._englishToFrenchPhrases[
self._englishToFrenchTranslator._englishSpeaker.responseToFarewell()  ] )

englishSpeaker = EnglishSpeaker()
englishToFrenchTranslator = Translator(englishSpeaker)
frenchSpeaker = FrenchSpeaker(englishToFrenchTranslator)
frenchSpeaker.exchangeGreetings()
frenchSpeaker.exchangeFarewell()

# OUTPUT
Salut!
Bonjour à vous aussi
Au revoir!
Au revoir mon ami
```

USOS CONOCIDOS

Un ejemplo dentro del JDK:

  - Clases adaptadoras del JDK

- Para gestionar eventos un objeto debe implementar *EventListener*
- Para gestionar eventos de objetos de tipo Window debe implementar la interfaz *WindowListener* que extiende *EventListener*
- *WindowListener* tiene siete métodos, pero en muchas ocasiones sólo no se usan más de tres.
- El JDK proporciona la clase abstracta *WindowAdapter* para dicho fin.

PATRONES RELACIONADOS

- **Bridge:** Estructura similar al Adapter. Su propósito es separar la interface de su implementación así esta puede variar fácilmente y de forma independiente.
- **Decorator:** Agrega responsabilidades a un objeto dinámicamente sin cambiar su interface.
- **Proxy:** Define un lugar para otro objeto para controlar el acceso y no cambia su interface.