ADDAPPTTER

**Intent**

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Also Known As**

Wrapper

**Motivation**

Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.

Classes for elementary geometric shapes like LineShape and PolygonShape are rather easy to implement, because their drawing and editing capabilities are inherently limited. But a TextShape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management. Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextView class for displaying and editing text. Ideally we'd like to reuse TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind. So we can't use TextView and Shape objects interchangeably.

How can existing and unrelated classes like TextView work in an application that expects classes with a different and incompatible interface? We could change the TextView class so that it conforms to the Shape interface, but that isn't an option unless we have the toolkit's source code. Even if we did, it wouldn't make sense to change TextView; the toolkit shouldn't have to adopt domain-specific interfaces just to make one application work.

Instead, we could define TextShape so that it adapts the TextView interface to Shape's. We can do this in one of two ways: (1) by inheriting Shape's interface and TextView's implementation or (2) by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface. These two approaches correspond to the class and object versions of the Adapter pattern. We call TextShape an adapter.

This diagram illustrates the object adapter case. It shows how BoundingBox requests, declared in class Shape, are converted to GetExtent requests defined in TextView. Since TextShape adapts TextView to the Shape interface, the drawing editor can reuse the otherwise incompatible TextView class.

*Fuente: Design Patterns. Erich Gamma, Richard Helm, Ralf Johnson, John Vlissides
Addison – Wesley Professional Computing Series*
Often the adapter is responsible for functionality the adapted class doesn't provide. The diagram shows how an adapter can fulfill such responsibilities. The user should be able to "drag" every Shape object to a new location interactively, but TextView isn't designed to do that. TextShape can add this missing functionality by implementing Shape's CreateManipulator operation, which returns an instance of the appropriate Manipulator subclass.

Manipulator is an abstract class for objects that know how to animate a Shape in response to user input, like dragging the shape to a new location. There are subclasses of Manipulator for different shapes; TextManipulator, for example, is the corresponding subclass for TextShape. By returning a TextManipulator instance, TextShape adds the functionality that TextView lacks but Shape requires.

**Applicability**

Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

**Structure**

A class adapter uses multiple inheritance to adapt one interface to another:

An object adapter relies on object composition:

*Fuente: Design Patterns. Erich Gamma, Richard Helm, Ralf Johnson, John Vlissides
Addison – Wesley Professional Computing Series*
Participantes

- **Target (Shape)**
  - define la interfaz específica que el Cliente utiliza.
- **Cliente (DrawingEditor)**
  - colabora con objetos que conforman a la interfaz del Target.
- **Adaptee (TextView)**
  - define una interfaz existente que necesita adaptarse.
- **Adaptador (TextShape)**
  - adapta la interfaz del Adaptee a la interfaz del Target.

Collaboraciones

- Los Clientes llaman operaciones a un instante de Adaptador. En cambio, el adaptador llama operaciones del Adaptee que realizan la solicitud.

Consecuencias

Clases y adaptadores de objeto tienen diferentes trade-offs. Un adaptador

- adapta el Adaptee al Target al comprometerse con una clase concreta de Adaptador. Como consecuencia, un adaptador no funcionará cuando queramos adaptar una clase y todas sus subclases.
- permite al Adaptador sobrescribir algunas de las operaciones del Adaptee, ya que Adaptador es un subclase de Adaptee.
- introduce solo un objeto, y no se necesita ninguna interpuntesión adicional para llegar al adaptée.

Un adaptador de objeto

- permite que un solo Adaptador trabaje con muchas Adaptees — es decir, el Adaptee mismo y todos sus subclases (si los hay). El Adaptador también puede agregar funcionalidad a todos los Adaptees al mismo tiempo.
- dificulta sobrescribir el comportamiento del Adaptee. Se requerirá subclase el Adaptee y hacer que el Adaptador se refiera a la subclase en lugar del Adaptee mismo.

Aquí hay otras consideraciones para tener en cuenta al usar el patrón de Adaptador:

1. *How much adapting does Adapter do?* Adaptadores varían en el trabajo que hacen para adaptar el Adaptee a la interfaz de Target. Hay un espectro de trabajo posible, desde la conversión de interfaz simple—por ejemplo, cambiar los nombres de operaciones—hasta el soporte de una interfaz completamente diferente.
set of operations. The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.

2. **Pluggable adapters.** A class is more reusable when you minimize the assumptions other classes must make to use it. By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface. Put another way, interface adaptation lets us incorporate our class into existing systems that might expect different interfaces to the class. ObjectWorks/Smalltalk [Par90] uses the term **pluggable adapter** to describe classes with built-in interface adaptation.

Consider a TreeDisplay widget that can display tree structures graphically. If this were a special-purpose widget for use in just one application, then we might require the objects that it displays to have a specific interface; that is, all must descend from a Tree abstract class. But if we wanted to make TreeDisplay more reusable (say we wanted to make it part of a toolkit of useful widgets), then that requirement would be unreasonable. Applications will define their own classes for tree structures. They shouldn't be forced to use our Tree abstract class. Different tree structures will have different interfaces.

In a directory hierarchy, for example, children might be accessed with a GetSubdirectories operation, whereas in an inheritance hierarchy, the corresponding operation might be called GetSubclasses. A reusable TreeDisplay widget must be able to display both kinds of hierarchies even if they use different interfaces. In other words, the TreeDisplay should have interface adaptation built into it.

We'll look at different ways to build interface adaptation into classes in the Implementation section.

3. **Using two-way adapters to provide transparency.** A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can. **Two-way adapters** can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.

Consider the two-way adapter that integrates Unidraw, a graphical editor framework [Vl90], and QOCA, a constraint-solving toolkit [HHMV92]. Both systems have classes that represent variables explicitly: Unidraw has StateVariable, and QOCA has ConstraintVariable. To make Unidraw work with QOCA, ConstraintVariable must be adapted to StateVariable; to let QOCA propagate solutions to Unidraw, StateVariable must be adapted to ConstraintVariable.

The solution involves a two-way class adapter ConstraintStateVariable, a subclass of both StateVariable and ConstraintVariable, that adapts the two interfaces to each other. Multiple inheritance is a viable solution in this case because the interfaces of the adapted classes are substantially different. The two-way class adapter conforms to both of the adapted classes and can work in either system.