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.

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:

Participants

- **Target** (Shape)
  - defines the domain-specific interface that Client uses.
- **Client** (DrawingEditor)
  - collaborates with objects conforming to the Target interface.
- **Adaptee** (TextView)
  - defines an existing interface that needs adapting.
- **Adapter** (TextShape)
  - adapts the interface of Adaptee to the Target interface.

Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

Class and object adapters have different trade-offs. A class adapter

- adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Here are other issues to consider when using the Adapter pattern:

1. *How much adapting does Adapter do?* Adapters vary in the amount of work they do to adapt Adaptee to the Target interface. There is a spectrum of possible work, from simple interface conversion—for example, changing the names of operations—to supporting an entirely different 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.

Implementation

Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:

1. *Implementing class adapters in C++.* In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptee. Thus Adapter would be a subtype of Target but not of Adaptee.

2. *Pluggable adapters.* Let's look at three ways to implement pluggable adapters for the TreeDisplay widget described earlier, which can lay out and display a hierarchical structure automatically.

   The first step, which is common to all three of the implementations discussed here, is to find a "narrow" interface for Adaptee, that is, the smallest subset of operations that lets us do the adaptation. A narrow interface consisting of only a couple of operations is easier to adapt than an interface with dozens of operations. For TreeDisplay, the adaptee is any hierarchical structure. A minimalist interface might include two operations, one that defines how to present a node in the hierarchical structure graphically, and another that retrieves the node's children.

   The narrow interface leads to three implementation approaches:

   a. *Using abstract operations.* Define corresponding abstract operations for the narrow Adaptee interface in the TreeDisplay class. Subclasses must implement the abstract operations and adapt the hierarchically structured object. For example, a DirectoryTreeDisplay subclass will implement these operations by accessing the directory structure.

      DirectoryTreeDisplay specializes the narrow interface so that it can display directory structures made up of FileSystemEntity objects.

   b. *Using delegate objects.* In this approach, TreeDisplay forwards requests for accessing the hierarchical structure to a **delegate** object. TreeDisplay can use a different adaptation strategy by substituting a different delegate.

      For example, suppose there exists a DirectoryBrowser that uses a TreeDisplay. DirectoryBrowser might make a good delegate for adapting TreeDisplay to the hierarchical directory structure. In dynamically typed languages like Smalltalk or Objective C, this approach only requires an interface for registering the delegate with the adapter. Then TreeDisplay simply forwards the requests to the delegate. NEXTSTEP [Add94] uses this approach heavily to reduce subclassing.

      Statically typed languages like C++ require an explicit interface definition for the delegate. We can specify such an interface by putting the narrow interface that TreeDisplay requires into an abstract TreeAccessorDelegate class. Then we can mix this interface into the delegate of our choice—DirectoryBrowser in this case—using inheritance. We use single inheritance if the DirectoryBrowser has no existing parent class, multiple inheritance if it does. Mixing classes together like this is easier than introducing a new TreeDisplay subclass and implementing its operations individually.

c. *Parameterized adapters.* The usual way to support pluggable adapters in Smalltalk is to parameterize an adapter with one or more blocks. The block construct supports adaptation without subclassing. A block can adapt a request, and the adapter can store a block for each individual request. In our example, this means TreeDisplay stores one block for converting a node into a GraphicNode and another block for accessing a node's children.

For example, to create TreeDisplay on a directory hierarchy, we write

```
directoryDisplay :=
    (TreeDisplay on: treeRoot)
         getChildrenBlock:
             [:node | node getSubdirectories]
         createGraphicNodeBlock:
             [:node | node createGraphicNode].
```

If you're building interface adaptation into a class, this approach offers a convenient alternative to subclassing.

Sample Code

We'll give a brief sketch of the implementation of class and object adapters for the Motivation example beginning with the classes `Shape` and `TextView`.

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

`Shape` assumes a bounding box defined by its opposing corners. In contrast, `TextView` is defined by an origin, height, and width. `Shape` also defines a `CreateManipulator` operation for creating a `Manipulator` object, which knows how to animate a shape when the user manipulates it.[1] `TextView` has no equivalent operation. The class `TextShape` is an adapter between these different interfaces.

A class adapter uses multiple inheritance to adapt interfaces. The key to class adapters is to use one inheritance branch to inherit the interface and another branch to inherit the implementation. The usual way to make this distinction in C++ is to inherit the interface publicly and inherit the implementation privately. We'll use this convention to define the `TextShape` adapter.

```
class TextShape : public Shape, private TextView {
```

```
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

The `BoundingBox` operation converts `TextView`'s interface to conform to `Shape`'s.

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

The `IsEmpty` operation demonstrates the direct forwarding of requests common in adapter implementations:

```
bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}
```

Finally, we define `CreateManipulator` (which isn't supported by `TextView`) from scratch. Assume we've already implemented a `TextManipulator` class that supports manipulation of a `TextShape`.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

The object adapter uses object composition to combine classes with different interfaces. In this approach, the adapter `TextShape` maintains a pointer to `TextView`.

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

`TextShape` must initialize the pointer to the `TextView` instance, and it does so in the constructor. It must also call operations on its `TextView` object whenever its own operations are called. In this example, assume that the client creates the `TextView` object and passes it to the `TextShape` constructor:

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
```

```
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

`CreateManipulator`'s implementation doesn't change from the class adapter version, since it's implemented from scratch and doesn't reuse any existing `TextView` functionality.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

Compare this code to the class adapter case. The object adapter requires a little more effort to write, but it's more flexible. For example, the object adapter version of `TextShape` will work equally well with subclasses of `TextView`---the client simply passes an instance of a `TextView` subclass to the `TextShape` constructor.

Known Uses

The Motivation example comes from ET++Draw, a drawing application based on ET++ [WGM88]. ET++Draw reuses the ET++ classes for text editing by using a TextShape adapter class.

InterViews 2.6 defines an Interactor abstract class for user interface elements such as scroll bars, buttons, and menus [VL88]. It also defines a Graphic abstract class for structured graphic objects such as lines, circles, polygons, and splines. Both Interactors and Graphics have graphical appearances, but they have different interfaces and implementations (they share no common parent class) and are therefore incompatible—you can't embed a structured graphic object in, say, a dialog box directly.

Instead, InterViews 2.6 defines an object adapter called GraphicBlock, a subclass of Interactor that contains a Graphic instance. The GraphicBlock adapts the interface of the Graphic class to that of Interactor. The GraphicBlock lets a Graphic instance be displayed, scrolled, and zoomed within an Interactor structure.

Pluggable adapters are common in ObjectWorks\Smalltalk [Par90]. Standard Smalltalk defines a ValueModel class for views that display a single value. ValueModel defines a `value`, `value:` interface for accessing the value. These are abstract methods. Application writers access the value with more domain-specific names like `width` and `width:`, but they shouldn't have to subclass ValueModel to adapt such application-specific names to the ValueModel interface.

Instead, ObjectWorks\Smalltalk includes a subclass of ValueModel called PluggableAdaptor. A PluggableAdaptor object adapts other objects to the ValueModel interface (`value`, `value:`). It can be parameterized with blocks for getting and setting the desired value. PluggableAdaptor uses these blocks internally to implement the `value`, `value:` interface. PluggableAdaptor also lets you pass in the selector names (e.g., `width`, `width:`) directly for syntactic convenience. It converts these selectors into the corresponding blocks automatically.

Another example from ObjectWorks\Smalltalk is the TableAdaptor class. A TableAdaptor can adapt a sequence of objects to a tabular presentation. The table displays one object per row. The client parameterizes TableAdaptor with the set of messages that a table can use to get the column values from an object.

Some classes in NeXT's AppKit [Add94] use delegate objects to perform interface adaptation. An example is the NXBrowser class that can display hierarchical lists of data. NXBrowser uses a delegate object for accessing and adapting the data.

Meyer's "Marriage of Convenience" [Mey88] is a form of class adapter. Meyer describes how a FixedStack class adapts the implementation of an Array class to the interface of a Stack class. The result is a stack containing a fixed number of entries.

Related Patterns

Bridge (151) has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an *existing* object.

Decorator (175) enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.

Proxy (207) defines a representative or surrogate for another object and does not change its interface.

Bridge
Structural Patterns

---

[1] `CreateManipulator` is an example of a Factory Method (107).