DECORATOR

Object Structural

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also Known As

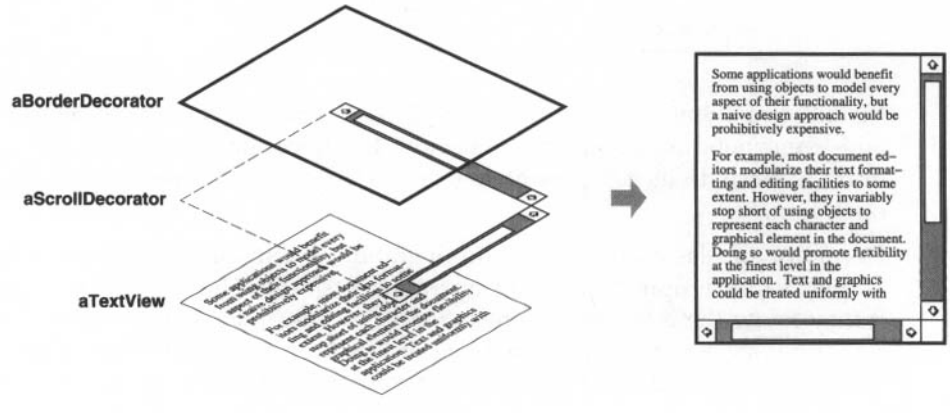
Wrapper

Motivation

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

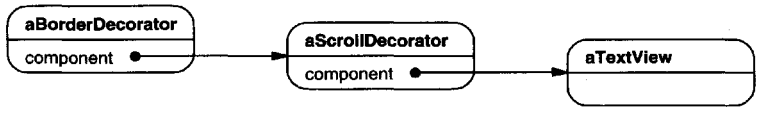
One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.

A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**. The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

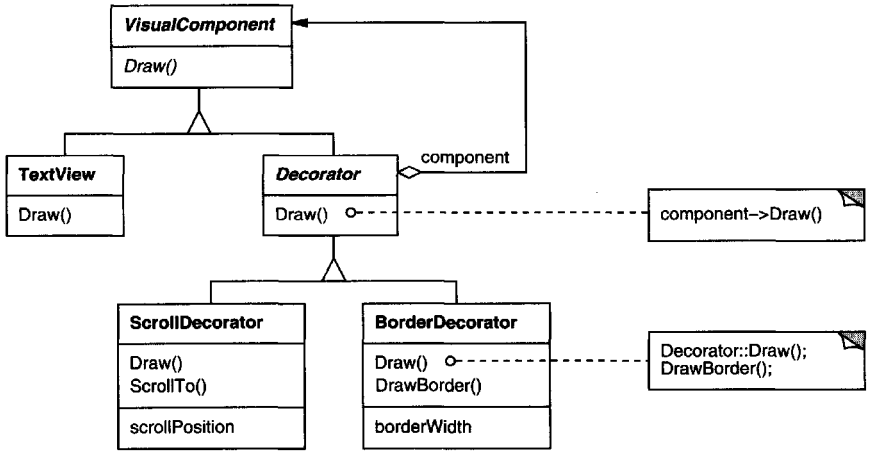


For example, suppose we have a TextView object that displays text in a window. TextView has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them. Suppose we also want to add a thick black border around the TextView. We can use a BorderDecorator to add this as well. We simply compose the decorators with the TextView to produce the desired result.

The following object diagram shows how to compose a TextView object with BorderDecorator and ScrollDecorator objects to produce a bordered, scrollable text view:



The ScrollDecorator and BorderDecorator classes are subclasses of Decorator, an abstract class for visual components that decorate other visual components.



VisualComponent is the abstract class for visual objects. It defines their drawing and event handling interface. Note how the Decorator class simply forwards draw requests to its component, and how Decorator subclasses can extend this operation.

Decorator subclasses are free to add operations for specific functionality. For example, ScrollDecorator's ScrollTo operation lets other objects scroll the interface *if* they know there happens to be a ScrollDecorator object in the interface. The important aspect of this pattern is that it lets decorators appear anywhere a VisualComponent can. That way clients generally can't tell the difference between

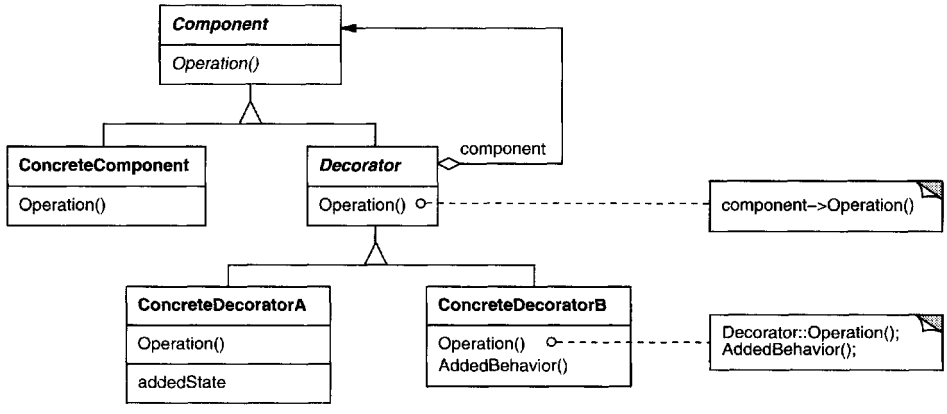
a decorated component and an undecorated one, and so they don't depend at all on the decoration.

Applicability

Use Decorator

- to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination. Or a class definition may be hidden or otherwise unavailable for subclassing.

Structure



Participants

- **Component** (VisualComponent)
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent** (TextView)
  - defines an object to which additional responsibilities can be attached.
- **Decorator**
  - maintains a reference to a Component object and defines an interface that conforms to Component's interface.

- **ConcreteDecorator** (BorderDecorator, ScrollDecorator)
  - adds responsibilities to the component.

Collaborations

- Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

Consequences

The Decorator pattern has at least two key benefits and two liabilities:

1. *More flexibility than static inheritance.* The Decorator pattern provides a more flexible way to add responsibilities to objects than can be had with static (multiple) inheritance. With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them. In contrast, inheritance requires creating a new class for each additional responsibility (e.g., BorderedScrollableTextView, BorderedTextView). This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities.

   Decorators also make it easy to add a property twice. For example, to give a TextView a double border, simply attach two BorderDecorators. Inheriting from a Border class twice is error-prone at best.

2. *Avoids feature-laden classes high up in the hierarchy.* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use. It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions. Extending a complex class tends to expose details unrelated to the responsibilities you're adding.

3. *A decorator and its component aren't identical.* A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you shouldn't rely on object identity when you use decorators.

4. *Lots of little objects.* A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Implementation

Several issues should be considered when applying the Decorator pattern:

1. *Interface conformance.* A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class (at least in C++).

2. *Omitting the abstract Decorator class.* There's no need to define an abstract Decorator class when you only need to add one responsibility. That's often the case when you're dealing with an existing class hierarchy rather than designing a new one. In that case, you can merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.

3. *Keeping Component classes lightweight.* To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data. The definition of the data representation should be deferred to subclasses; otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity. Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

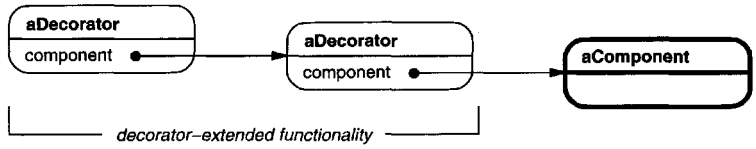
4. *Changing the skin of an object versus changing its guts.* We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy (315) pattern is a good example of a pattern for changing the guts.

   Strategies are a better choice in situations where the Component class is intrinsically heavyweight, thereby making the Decorator pattern too costly to apply. In the Strategy pattern, the component forwards some of its behavior to a separate strategy object. The Strategy pattern lets us alter or extend the component's functionality by replacing the strategy object.

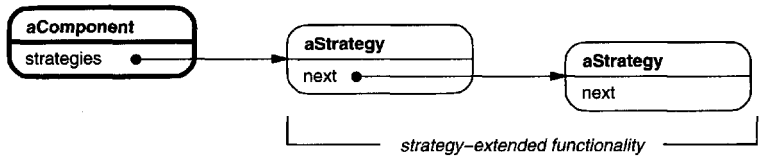
   For example, we can support different border styles by having the component defer border-drawing to a separate Border object. The Border object is a Strategy object that encapsulates a border-drawing strategy. By extending the number of strategies from just one to an open-ended list, we achieve the same effect as nesting decorators recursively.

   In MacApp 3.0 [App89] and Bedrock [Sym93a], for example, graphical components (called "views") maintain a list of "adorner" objects that can attach additional adornments like borders to a view component. If a view has any adorners attached, then it gives them a chance to draw additional embellishments. MacApp and Bedrock must use this approach because the View class is heavyweight. It would be too expensive to use a full-fledged View just to add a border.

   Since the Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorators; that is, the decorators are transparent to the component:



With strategies, the component itself knows about possible extensions. So it has to reference and maintain the corresponding strategies:



The Strategy-based approach might require modifying the component to accommodate new extensions. On the other hand, a strategy can have its own specialized interface, whereas a decorator's interface must conform to the component's. A strategy for rendering a border, for example, need only define the interface for rendering a border (DrawBorder, GetWidth, etc.), which means that the strategy can be lightweight even if the Component class is heavyweight.

MacApp and Bedrock use this approach for more than just adorning views. They also use it to augment the event-handling behavior of objects. In both systems, a view maintains a list of "behavior" objects that can modify and intercept events. The view gives each of the registered behavior objects a chance to handle the event before nonregistered behaviors, effectively overriding them. You can decorate a view with special keyboard-handling support, for example, by registering a behavior object that intercepts and handles key events.

Sample Code

The following code shows how to implement user interface decorators in C++. We'll assume there's a Component class called `VisualComponent`.

```
class VisualComponent {
public:
    VisualComponent();

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

We define a subclass of `VisualComponent` called `Decorator`, which we'll subclass to obtain different decorations.

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

`Decorator` decorates the `VisualComponent` referenced by the `_component` instance variable, which is initialized in the constructor. For each operation in `VisualComponent`'s interface, `Decorator` defines a default implementation that passes the request on to `_component`:

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

Subclasses of `Decorator` define specific decorations. For example, the class `BorderDecorator` adds a border to its enclosing component. `BorderDecorator` is a subclass of `Decorator` that overrides the `Draw` operation to draw the border. `BorderDecorator` also defines a private `DrawBorder` helper operation that does the drawing. The subclass inherits all other operation implementations from `Decorator`.

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

A similar implementation would follow for `ScrollDecorator` and `DropShadowDecorator`, which would add scrolling and drop shadow capabilities to a visual component.

Now we can compose instances of these classes to provide different decorations. The following code illustrates how we can use decorators to create a bordered scrollable `TextView`.

First, we need a way to put a visual component into a window object. We'll assume our `Window` class provides a `SetContents` operation for this purpose:

```
void Window::SetContents (VisualComponent* contents) {
    // ...
}
```

Now we can create the text view and a window to put it in:

```
Window* window = new Window;
TextView* textView = new TextView;
```

`TextView` is a `VisualComponent`, which lets us put it into the window:

```
window->SetContents(textView);
```

But we want a bordered and scrollable `TextView`. So we decorate it accordingly before putting it in the window.

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

Because `Window` accesses its contents through the `VisualComponent` interface, it's unaware of the decorator's presence. You, as the client, can still keep track of the text view if you have to interact with it directly, for example, when you need to invoke operations that aren't part of the `VisualComponent` interface. Clients that rely on the component's identity should refer to it directly as well.

Known Uses

Many object-oriented user interface toolkits use decorators to add graphical embellishments to widgets. Examples include InterViews [LVC89, LCI+92], ET++ [WGM88], and the ObjectWorks\Smalltalk class library [Par90]. More exotic applications of Decorator are the DebuggingGlyph from InterViews and the PassivityWrapper from ParcPlace Smalltalk. A DebuggingGlyph prints out debugging information before and after it forwards a layout request to its component. This trace information can be used to analyze and debug the layout behavior

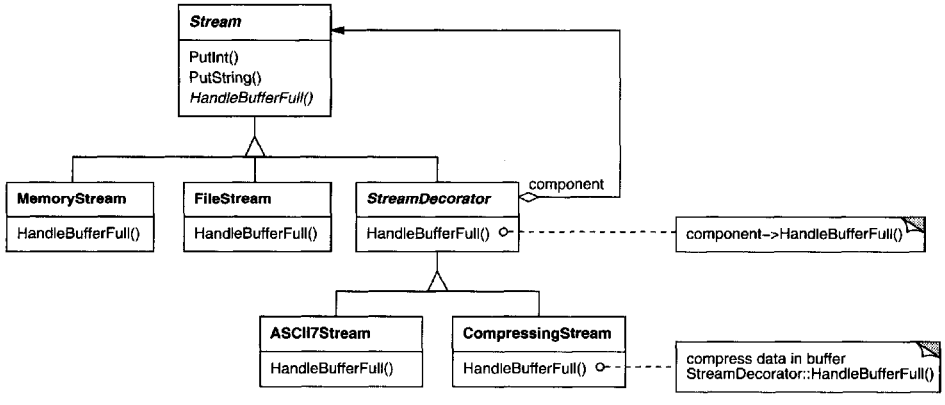
of objects in a complex composition. The PassivityWrapper can enable or disable user interactions with the component.

But the Decorator pattern is by no means limited to graphical user interfaces, as the following example (based on the ET++ streaming classes [WGM88]) illustrates.

Streams are a fundamental abstraction in most I/O facilities. A stream can provide an interface for converting objects into a sequence of bytes or characters. That lets us transcribe an object to a file or to a string in memory for retrieval later. A straightforward way to do this is to define an abstract Stream class with subclasses MemoryStream and FileStream. But suppose we also want to be able to do the following:

- Compress the stream data using different compression algorithms (run-length encoding, Lempel-Ziv, etc.).
- Reduce the stream data to 7-bit ASCII characters so that it can be transmitted over an ASCII communication channel.

The Decorator pattern gives us an elegant way to add these responsibilities to streams. The diagram below shows one solution to the problem:



The Stream abstract class maintains an internal buffer and provides operations for storing data onto the stream (PutInt, PutString). Whenever the buffer is full, Stream calls the abstract operation HandleBufferFull, which does the actual data transfer. The FileStream version of this operation overrides this operation to transfer the buffer to a file.

The key class here is StreamDecorator, which maintains a reference to a component stream and forwards requests to it. StreamDecorator subclasses override HandleBufferFull and perform additional actions before calling StreamDecorator's HandleBufferFull operation.

For example, the CompressingStream subclass compresses the data, and the ASCII7Stream converts the data into 7-bit ASCII. Now, to create a FileStream that compresses its data *and* converts the compressed binary data to 7-bit ASCII, we decorate a FileStream with a CompressingStream and an ASCII7Stream:

```
Stream* aStream = new CompressingStream(
    new ASCII7Stream(
         new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
```

Related Patterns

Adapter (139): A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

Composite (163): A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.

Strategy (315): A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.