Decorator Pattern [GoF]

Intent

Attach additional capabilities to objects dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also Known As

Wrapper.

Motivation

Suppose you have a class that allows an application to write data to a file:

```
public class FileOutputStream {
   public FileOutputStream(String name);
   public write(byte[] data);
   public void flush();
   public void close();
}
```

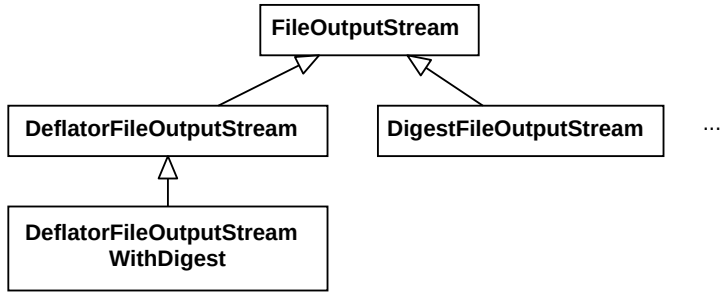

Imaging that the same or another application generates large data to be saved on disk. You decide to introduce another class that compresses the data:

```
public class DeflatorFileOutputStream extends FileOutputStr. {
   public void write(byte[] data);
}
```

Yet other applications may require additional responsibilities for writing data to files:

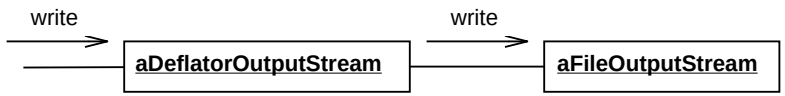
- some want to URL-encode strings before saving them on disk
- some want to create a digest (hash code) generated from the original data, and append it to the data to be saved on disk

Thus, you might end up with a “polluted” class hierarchy:



A more flexible strategy for achieving the required responsibility is to use object composition: You enclose the an object with a given responsibility into another one which *adds* additional responsibilities. For example, a `FileOutputStream` object has the responsibility to save data on disk. In order to save data in a com-

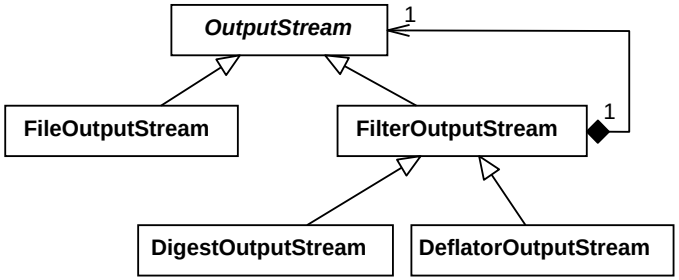
pressed way, compress it first by using a `DeflatorOutputStream` object which possesses the deflating responsibility only:



The client first creates a `FileOutputStream` object, and then encapsulates it with a `DeflatorOutputStream` object. Then, the client uses the `DeflatorOutputStream` object as it were merely a `FileOutputStream` object. The `DeflatorOutputStream` object is a sort of *decorator* for the `FileOutputStream` object.

However, it must be ensured that both objects provide the *same interface*. In addition, the decorating object must *delegate* its method invocations sooner or later to the decorated object.

This can be achieved by introducing the following class structure:



`OutputStream` is an abstract class for all kind of output stream objects. It defines the common interface for (output) stream-related methods such as several versions of `write`, `flush`, and `close`.

`FilterOutputStream` is a concrete (or abstract) class, the so-called *decorator*, that simply forwards any `OutputStream` method to its delegate, an `OutputStream` object.

Subclasses of `FilterOutputStream` can extend the behavior of the methods inherited by `FilterOutputStream`.

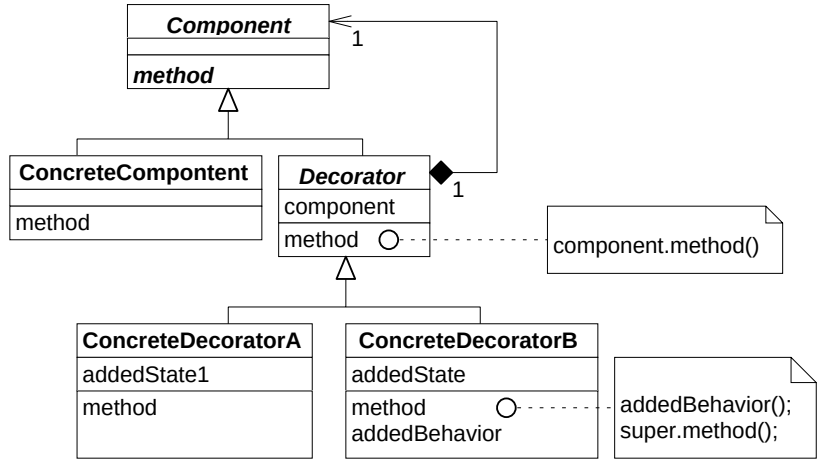
In addition, decorator subclasses are free to add specific functionality. For example, class `DeflatorOutputStream` may allow clients to call the `finish` method which finalizes the compression of data without closing the output stream.

Applicability

Use the Decorator pattern

- to add responsibilities to individual objects dynamically and transparently.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical.

Structure



Participants

- **`Component`** (`OutputStream`)
  - defines the interface for objects that can have capabilities added to them
- **`ConcreteComponent`** (`FileOutputStream`)
  - defines an object to which additional capabilities can be attached
- **`Decorator`**
  - maintains a reference to a `Component` object and defines an interface that conforms to `Component`'s interface
- **`ConcreteDecorator`** (`DeflatorOutputStream`, `DigestOutputStream`)
  - adds capabilities to the component

Collaborations

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

Consequences

- More flexible than static inheritance.
- Avoids feature-laden classes high up in the hierarchy.
- A `Decorator` and its `Component` aren't identical.
- Lots of little objects.

Implementation

- A `Decorator` object's interface must conform to the interface of the component it decorates. `ConcreteDecorator` classes must inherit from a common class.
- Class `Decorator` can be made concrete when you need only to add one responsibility.
- Keep `Component` class lightweight.
- Decorator pattern is an alternative to the Strategy pattern: The decorator allows to change an object's behavior by wrapping the original object with a decorator. By contrast, with the Strategy pattern an object's behavior is changed by replacing a strategy object (which is not visible by the client) by another strategy object.


```
public abstract class OutputStream {
   public abstract void write(int b) throws IOException;

   public void write(byte b[]) throws IOException {
      write(b, 0, b.length);
   }

   public void write(byte b[], int off, int len)
      throws IOException
   {
      // Perform some sanity checks. If OK then
      // write each byte in the byte array:
      for (....)
         write(b[some_index]);
   }

   public void flush() throws IOException {
   }

   public void close() throws IOException {
   }
}
```


Sample Code

The following sample code illustrates the use of the Decorator pattern. We give sketches of Java's `java.io` package.

We start with the abstract class `OutputStream`:

Class `FileOutputStream` writes data to files. Details are not given:

```
public class FileOutputStream extends OutputStream {
   // Implements and/or overwrites the methods of
   // class OutputStream.
   // In addition, some additional, class-specific
   // methods are here, too.
}
```

Class `FilterOutputStream` is the decorator base class. It defines the default behavior of decorators. (Due to methodological reasons, class `FilterOutputStream` has been simplified.)

```
public class FilterOutputStream extends OutputStream {
   private OutputStream out;

   public FilterOutputStream(OutputStream out)
   {  this.out = out; }

   public void write(int b) throws IOException
   { out.write(b); }
   public void write(byte b[]) throws IOException
   { out.write(b, 0, b.length); }
   public void write(byte b[], int off, int len)
      throws IOException
   {  out.write(b, off, len); }
   public void flush() throws IOException
   { out.flush(); }
   public void close() throws IOException
   {  out.close(); }
}
```


An application wanting to store text strings as URL-encoded strings may use an URL encoding filter:

```
public class UrlEncodeOutputStream
   extends FilterOutputStream
{
   // Use class java.net.URLEncoder to encode String
   // objects, left as an exercise.
   // Provide a method public void writeString(String s).
}
```

The application then can store text to disk in URL-encoded and deflated form:

```
Deflator defl = ...;
UrlEncodeOutputStream os = new UrlEncodeOutputStream(
                     new DeflatorOutputStream(
                        new FileOutputStream("file"), defl));
String s = ...; // string that must become URL-encoded
os.writeString(s);
```


Classes `DeflatorOutputStream` and `DigestOutputStream` are decorator subclasses. A sketch of `DeflatorOutputStream` is given next:

```
public class DeflatorOutputStream extends FilterOutputStream {
   protected byte[] buf;
   protected Deflator deflator;
   public DeflatorOutputStream(OutputStream os, Deflator d);
   public void write(byte[] b, int off, int len)
      throws IOException;
   public void write(int b) throws IOException;
   public void close() throws IOException;
   public void finish() throws IOException;
   protected void deflate() throws IOException;
}
```


Related Patterns

- Adapter: A decorator is different from an adapter in that a decorator only changes an object's responsibility, not its interface.
- Strategy: A decorator lets you change the skin; a strategy lets you change the interior.