Decorator

Intent

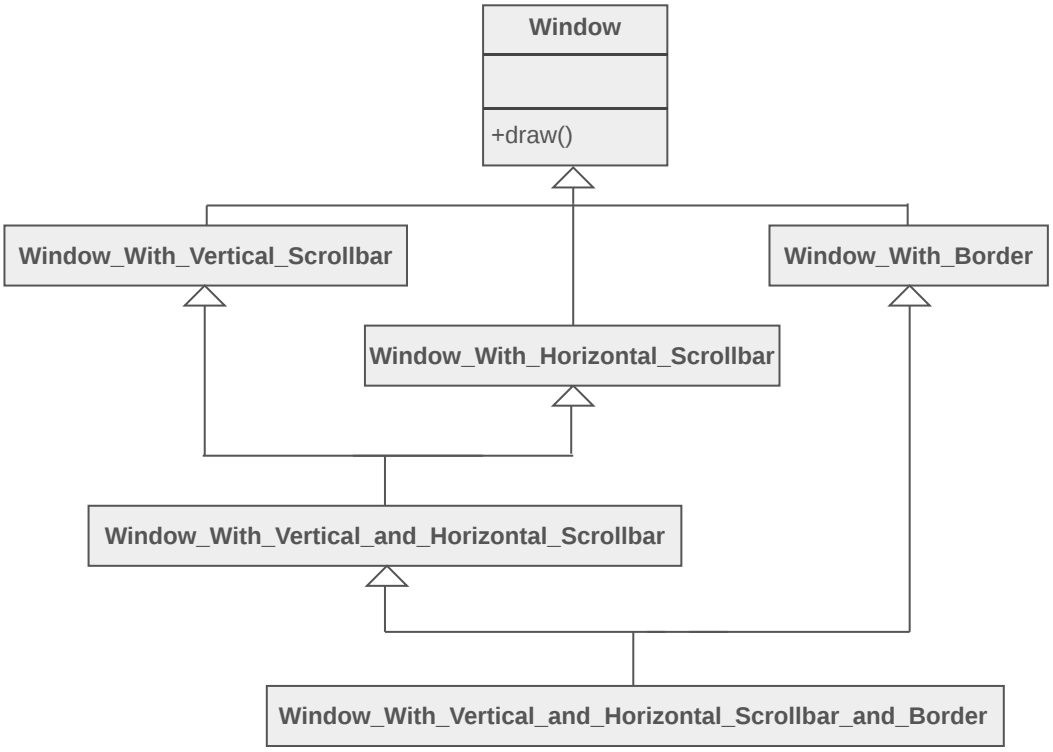
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Discussion

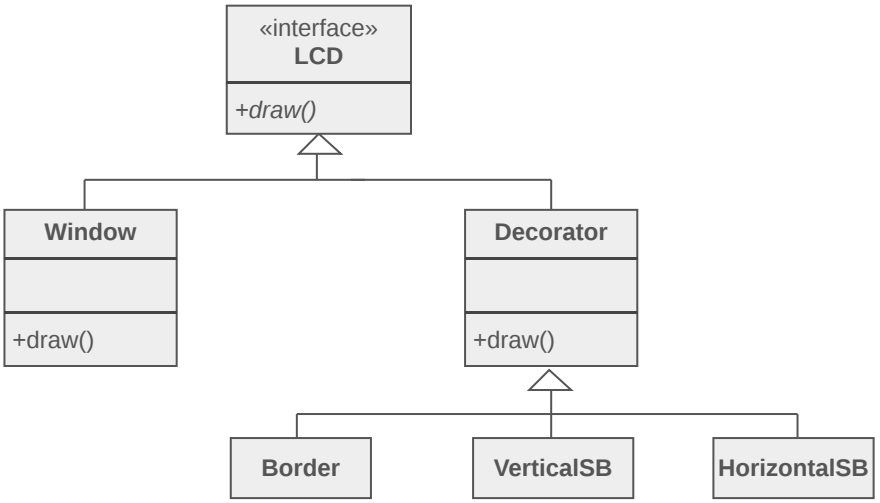
Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an inheritance hierarchy like ...



But the Decorator pattern suggests giving the client the ability to specify whatever combination of "features" is desired.

```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
    new VerticalScrollBarDecorator(
      new Window( 80, 24 ))));
aWidget->draw();
```

This flexibility can be achieved with the following design



Another example of cascading (or chaining) features together to produce a custom object might look like ...

```
Stream* aStream = new CompressingStream(
  new ASCII7Stream(
    new FileStream("fileName.dat")));
aStream->putString( "Hello world" );
```

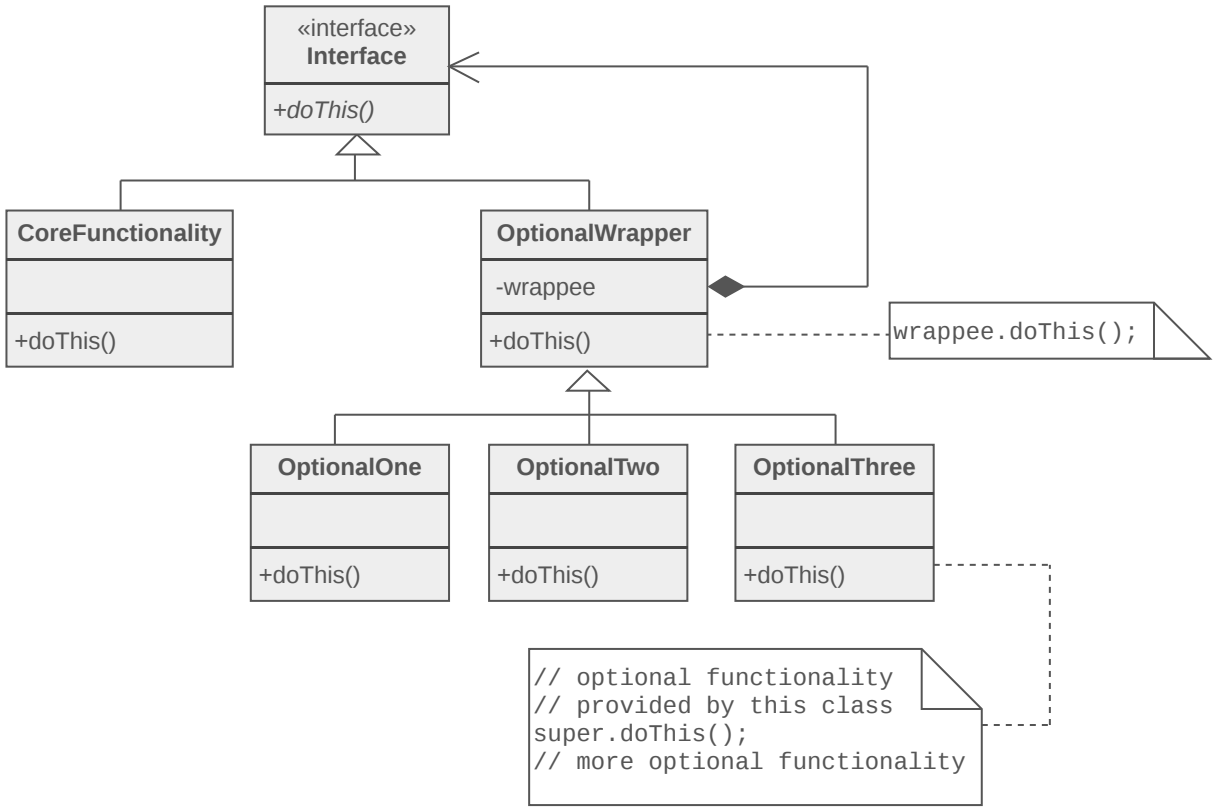
The solution to this class of problems involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

Structure

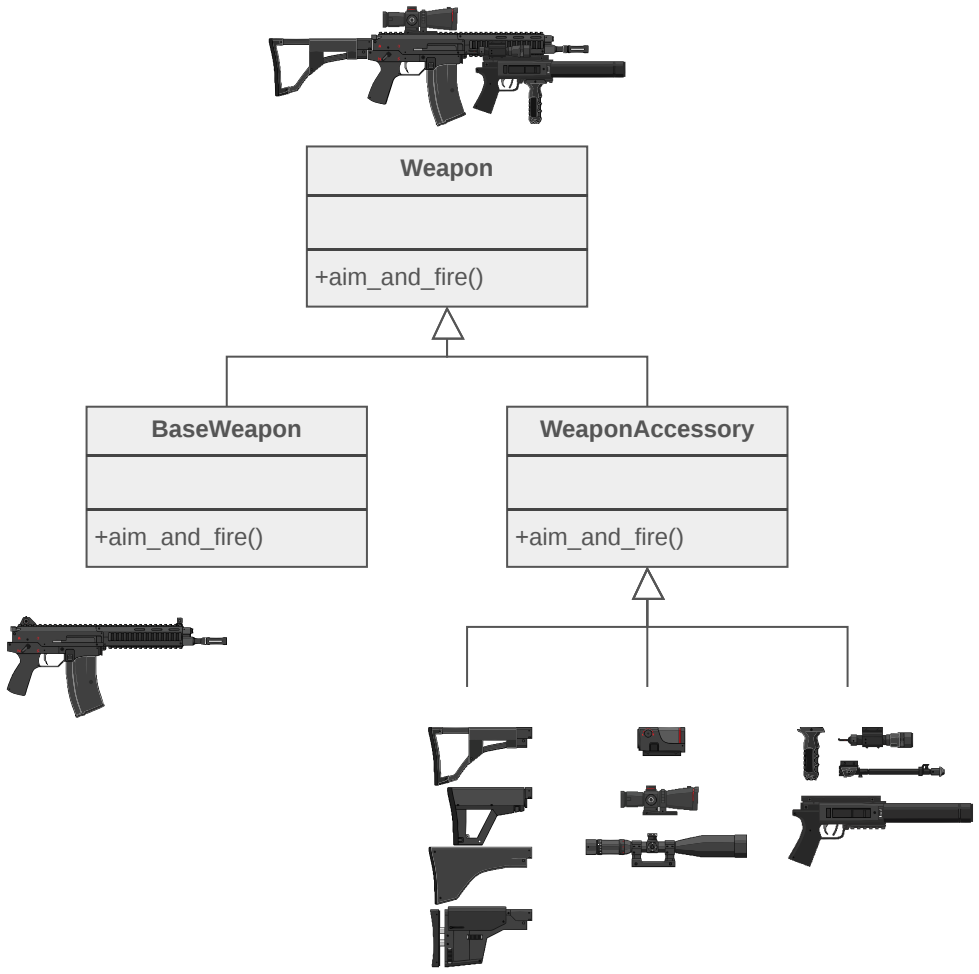
The client is always interested in `CoreFunctionality.doThis()`. The client may, or may not, be interested in `OptionalOne.doThis()` and `OptionalTwo.doThis()`. Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.



Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.

Another example: assault gun is a deadly weapon on it's own. But you can apply certain "decorations" to make it more accurate, silent and devastating.



Check list

1. Ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.
2. Create a "Lowest Common Denominator" interface that makes all classes interchangeable.
3. Create a second level base class (Decorator) to support the optional wrapper classes.
4. The Core class and Decorator class inherit from the LCD interface.
5. The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.
6. The Decorator class delegates to the LCD object.
7. Define a Decorator derived class for each optional embellishment.
8. Decorator derived classes implement their wrapper functionality - and - delegate to the Decorator base class.
9. The client configures the type and ordering of Core and Decorator objects.

Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

- Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.
- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- 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.
- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.
- Decorator lets you change the skin of an object. Strategy lets you change the guts.