An Exercise

Suppose that we are writing a new program using objects from our `Ball` hierarchy. In this program, some `MovableBall`s must decelerate. Every time one of these decelerating balls moves, its speed decreases by 5%.

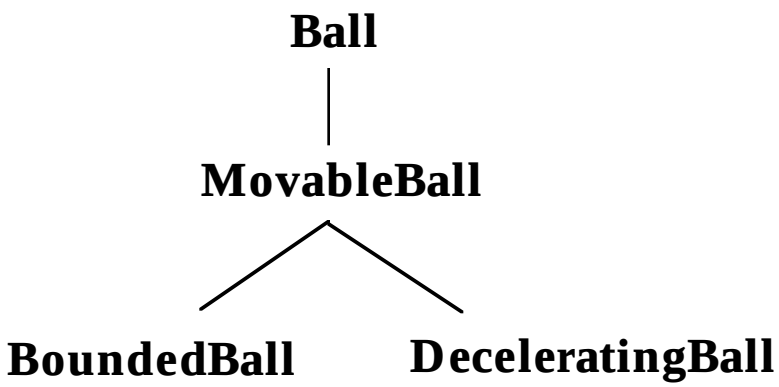
Add a `DeceleratingBall` class to the `Ball` hierarchy for this purpose.

```
public class Ball
{
   private Rectangle location;
   private Color color;

   ...
}

public class MovableBall extends Ball
{
   private double dx;
   private double dy;

   ...
}

public class BoundedBall extends MovableBall
{
   private int maxHeight;
   private int maxWidth;

   ...
}
```

A Possible Solution

```
public class DeceleratingBall extends MovableBall
{
   public DeceleratingBall( int x, int y, int r,
                            double dx, double dy )
   {
      super( x, y, r, dx, dy );
   }

   public void move()
   {
      super.move();
      setMotion( xMotion() * 0.95, yMotion() * 0.95 );
   }
}
```

We now have the following `Ball` hierarchy:



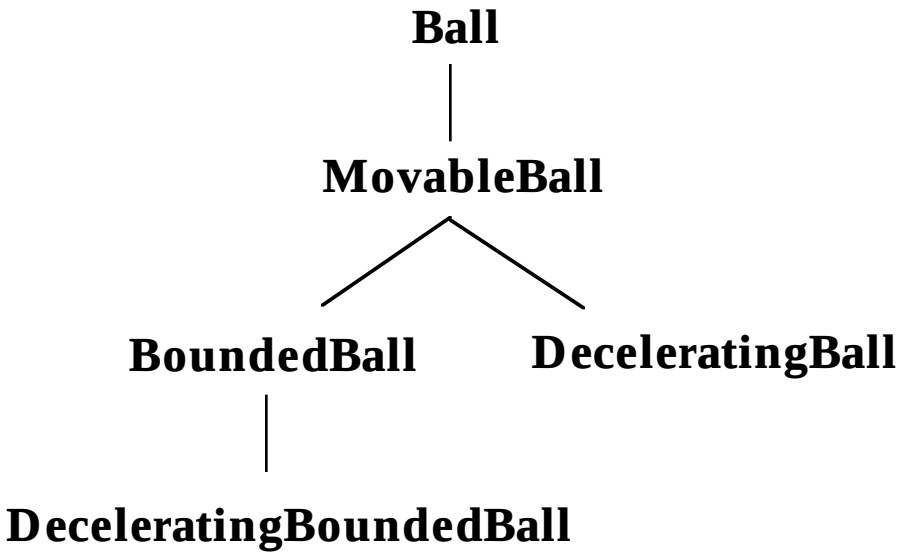
We create a `DeceleratingBall` just as we would a `MovableBall`:

```
DeceleratingBall b = new DeceleratingBall( 10, 15, 5, 5.0, 10.0 );
```

A New Wrinkle

It turns out that another program needs to use `BoundedBall`s that also decelerate. Can you fix the problem?

```
public class DeceleratingBoundedBall extends BoundedBall
{
   public DeceleratingBoundedBall(
                              int     x, int     y, int   r,
                              double dx, double dy, Frame f )
   {
      super( x, y, r, dx, dy, f );
   }

   public void move()
   {
      super.move();
      setMotion( xMotion() * 0.95, yMotion() * 0.95 );
   }
}
```



How Good is Our Solution?

What are the strengths of our general approach?

- It is simple.
- It is easy to implement right now.

What are the weaknesses of our approach?

- It is tedious.
- It repeats codes. The `move()` methods in `DeceleratingBall` and `DeceleratingBoundedBall` are identical!

You may be asking yourself, "So what? It works."

- What happens if we need to change the deceleration factor, say, from 95% to 80%?

  We must remember to make the change in two different classes.

- What happens if we need to add deceleration behavior to other classes that inherit from `MovableBall`? (*)

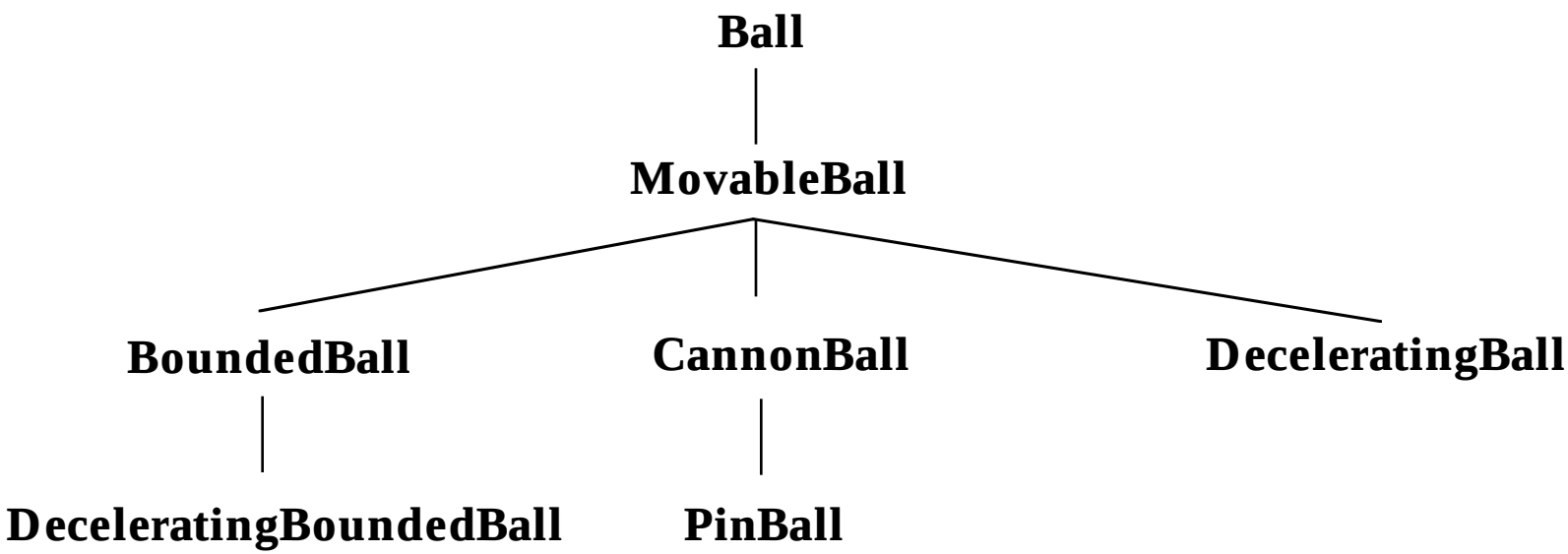
  More subclasses!

- What happens if we need to add another kind of behavior to our ball classes, including the decelerating balls?

  *Even **more** subclasses!*

Solutions that make future extensions to the system unbearable are probably not very good solutions at all...

The Full `Ball` Hierarchy



An Alternative Solution

BoundedBalls respond to the same set of messages as MovableBalls.
So they are *substitutable* for one another.
Can we use this to our advantage?

```
public class DeceleratingBall extends MovableBall
{
   private MovableBall workerBall;

   public DeceleratingBall( MovableBall aBall )
   {
      super();
      workerBall = aBall;
   }

   public void move()
   {
      workerBall.move();
      workerBall.setMotion( workerBall.xMotion() * 0.95,
                            workerBall.yMotion() * 0.95 );
   }

   // *** MESSAGES DELEGATED ENTIRELY TO THE INSTANCE VAR

   public void paint( Graphics g )
   {
      workerBall.paint( g );
   }

   public void setColor( Color newColor )
   {
      workerBall.setColor( newColor );
   }

   protected int radius()     { return workerBall.radius(); }
   protected int x()          { return workerBall.x(); }
   protected int y()          { return workerBall.y(); }
```

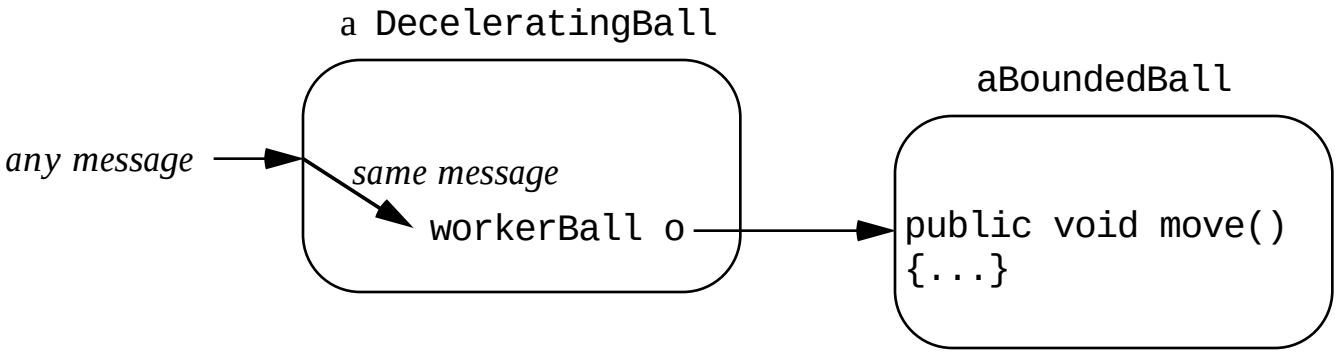
```
   protected Rectangle region() { return workerBall.region(); }
   protected Color color()      { return workerBall.color(); }

   protected void moveTo( int x, int y )
   {
      workerBall.moveTo( x, y );
   }

   // ------------------------------------------------------

   protected double xMotion() { return workerBall.xMotion(); }
   protected double yMotion() { return workerBall.yMotion(); }

   protected void setMotion( double ndx, double ndy )
   {
      workerBall.setMotion( ndx, ndy );
   }
}
```

Now, we create a `DeceleratingBall` by giving it a `MovableBall` to direct:

```
DeceleratingBall b =
   new DeceleratingBall(
          new MovableBall( 10, 15, 5, 2.0, 5.0 ) );


DeceleratingBall b =
   new DeceleratingBall(
          new BoundedBall( 10, 15, 5, 2.0, 5.0, this ) );
```

How the "Decorator" Works

```
MovableBall b = new DeceleratingBall(
                    new BoundedBall( x, y, r, dx, dy, this ) );
```



This approach takes advantage of substitutability to create an object that delegates all of its responsibility to another object... without the client ever knowing the difference!

How Good is Our New Solution?

What are the weaknesses of our new approach?

- It is more complex.
- Decelerating balls are a bit “bigger” and “slower” at run time.

What are the strengths of our new approach?

- It “says it once and only once”. The `move()` method specific to deceleration behavior occurs in one class. The deceleration factor lives in exactly one class.

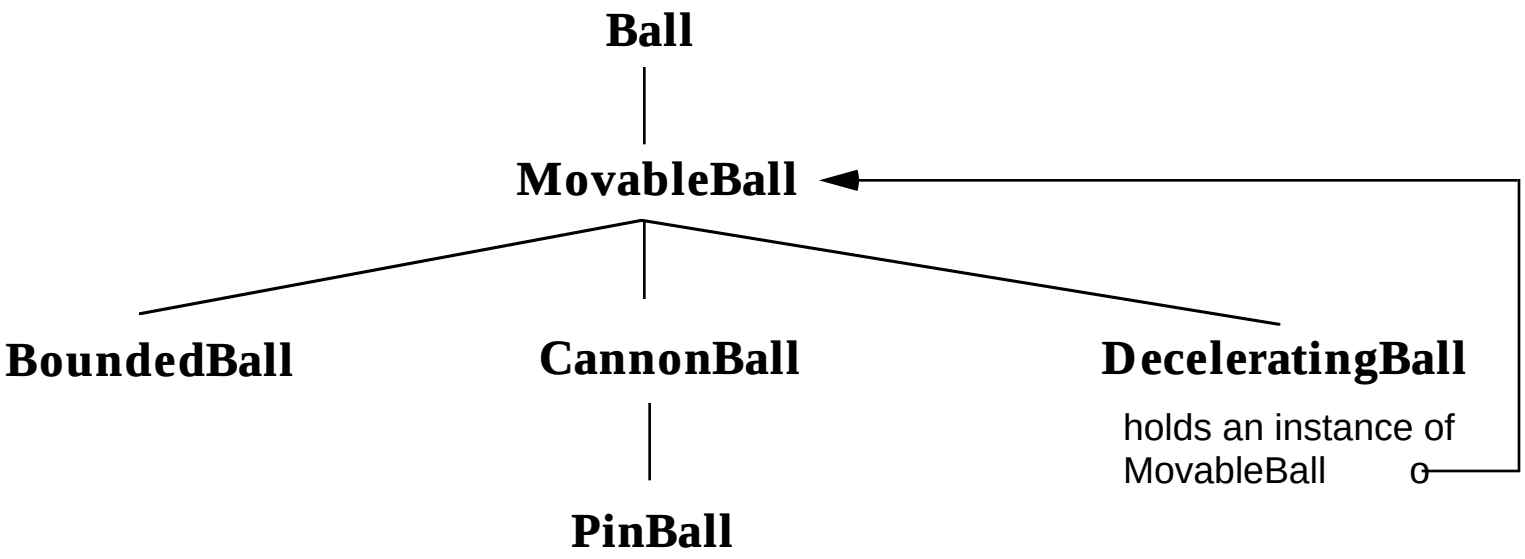
- We can add deceleration behavior to any `MovableBall` with this same class!

- We can add deceleration behavior to any *future* subclass of `MovableBall`—**with no new code**!!

- The tedious task of writing the delegation methods can be done automatically within many OO programming tools. In any case, writing them once seems more palatable than writing multiple subclasses for deceleration throughout the hierarchy.

As is often the case, we sometimes choose a somewhat more complex solution when it offers *extensibility* and *flexibility* as benefits.

The New `Ball` Hierarchy



An Exercise

Add an `ExpandingBall` class to the `MovableBall` hierarchy.
An `ExpandingBall` becomes a little bit larger every time it moves.

Make an `ExpandingBall` class that works like our `DeceleratingBall` class.

```
public class ExpandingBall extends MovableBall
{
   private MovableBall workerBall;

   public ExpandingBall( MovableBall aBall )
   {
      super();
      workerBall = aBall;
   }

   public void move()
   {
      workerBall.move();
      workerBall.region().height =
         ( workerBall.region().height * 11 ) / 10;
      workerBall.region().width  =
         ( workerBall.region().width  * 11 ) / 10;
   }

   // *** MESSAGES DELEGATED ENTIRELY TO THE INSTANCE VAR

   public void paint( Graphics g )
   {
      workerBall.paint( g );
   }

   ... // all the rest of the messages in MovableBall, too
}
```

Using An ExpandingBall

Here's how we might use an ExpandingBall in the MultiBallWorld:

```
protected void initializeArrayOfBalls( Color ballColor )
{
   ballArray = new MovableBall [ BallArraySize ];
   for (int i = 0; i < BallArraySize; i++)
   {
      ballArray[i] =
         new ExpandingBall(
             new BoundedBall(
                    10, 15, 5, 3.0+i, 6.0-i, this ) );
      ballArray[i].setColor( ballColor );
   }
}
```

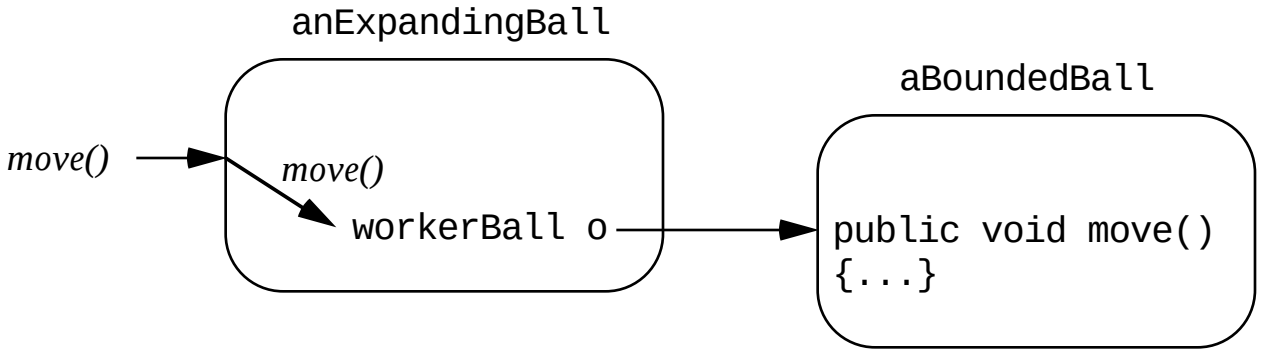
But here's a beautiful example of what using a decorator can do for you:

```
protected void initializeArrayOfBalls( Color ballColor )
{
   ballArray = new MovableBall [ BallArraySize ];
   for (int i = 0; i < BallArraySize; i++)
   {
      ballArray[i] =
         new ExpandingBall(
             new DeceleratingBall(
                 new BoundedBall(
                     10, 15, 5, 3.0+i, 6.0-i, this ) ) );
      ballArray[i].setColor( ballColor );
   }
}
```

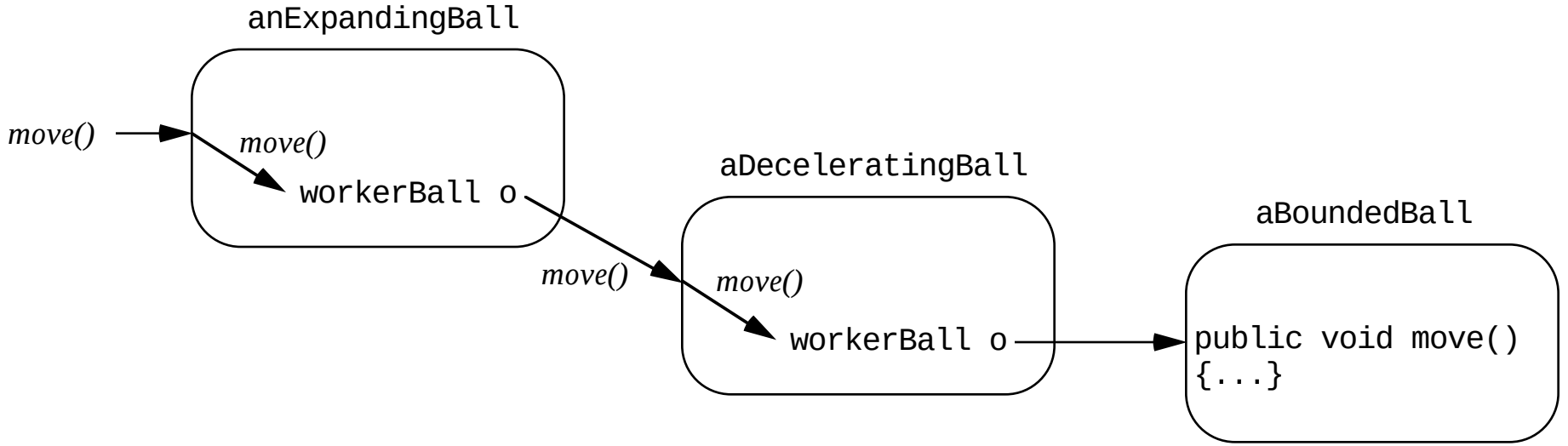
Since a decorator is substitutable for instances of its base class, you can decorate a decorator!

How a Decorator Works...

```
MovableBall b = new ExpandingBall( new BoundedBall( x, y, r, dx, dy, this ) );
```




```
MovableBall b = new ExpandingBall(
                    new DeceleratingBall(
                        new BoundedBall( x, y, r, dx, dy, this ) ) );
```



Do You Recognize a Pattern?

We added flexibility and extensibility to our system using the same ideas we used in our previous two class sessions: combining **composition** and **inheritance**.

- substitution
- delegation
- recursion

The new twist in this solution is that `DeceleratingBall` and `ExpandingBall` use substitution on a class in their own class hierarchy!

This new twist is so common that it has its own name: **decorator**. (*)

The Problem

> We would like to add a behavior to a *set* of classes that share a common interface.

A Tempting Solution that Fails

> Use inheritance to create a new class of objects that has the behavior. Use instances of this class when you need the behavior, and use instances of the superclass otherwise.
>
> This solution is impractical. Why?
>
> We will need to create multiple subclasses and replicate the behavior in each.
>
> What if we would like to add *more* behavior to the extended object? We have to make (many!) more subclasses!

The Problem Solved by the Decorator Pattern

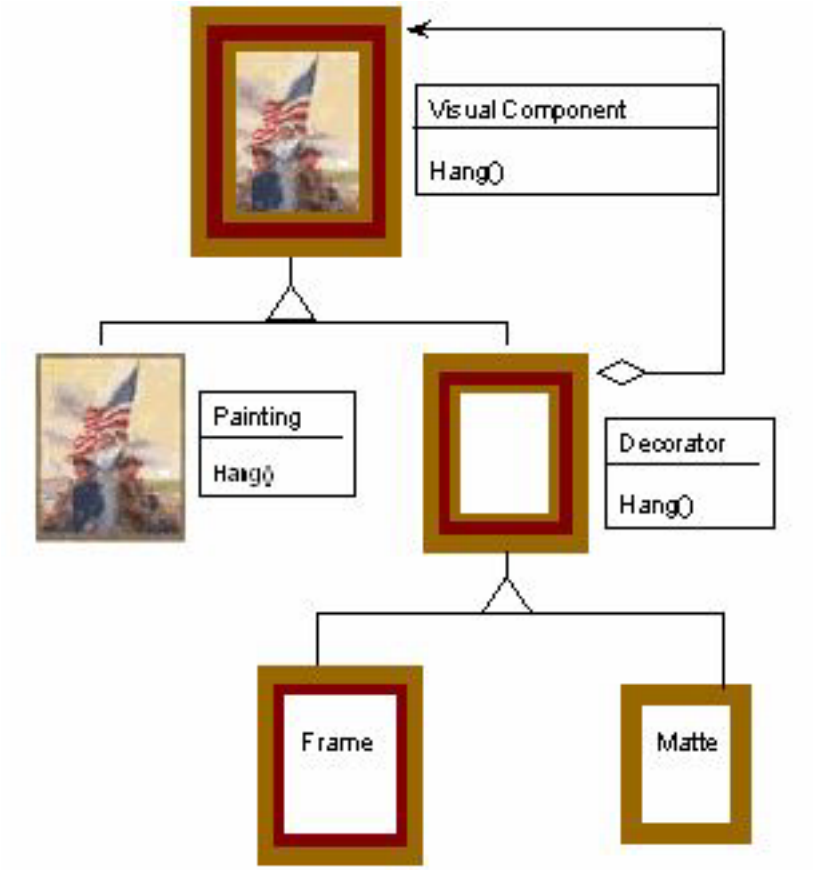
Why Does This Problem Matter?

It occurs in many domains and in many applications:

- We want to add features to individual balls in our ball games or to individual card piles in our card games.
- We want to add features to individual streams in the Java library, such as buffering the input we read from a stream.
- We want to add windowing features to individual objects in a word processor or drawing program.

GoF figure bottom of page 175

Decorators Do Not Occur Only in Programs...



The Decorator Pattern

The Solution

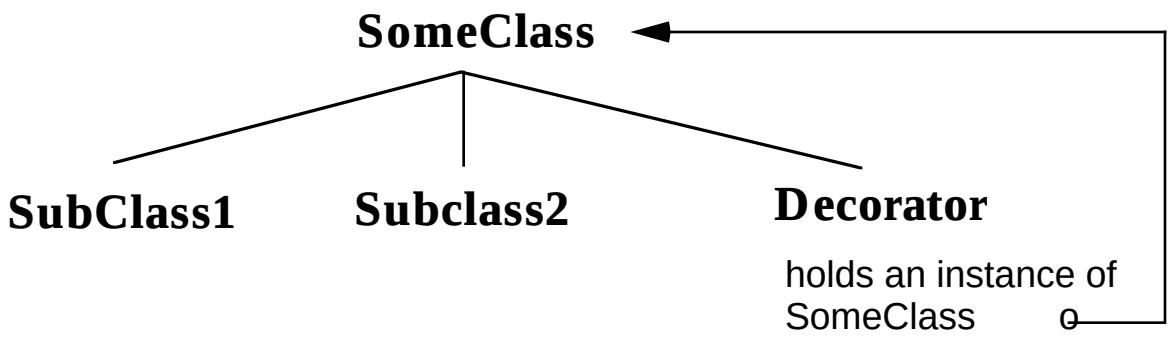
Create a decorator class.

1. *Encapsulate* an instance of the base class as an instance variable of the decorator.

   Implement the new behavior in the decorator. Delegate as much of the new behavior to the instance variable as possible.

   Send all other messages recursively to the encapsulated object.

2. Use *inheritance* to extend the decorator class from the contained class.



How Does the Decorator Pattern Work?

This is a second example of *using inheritance and composition together*:

Inheritance creates substitutable classes. This allows decorated object to be used in all the same places as the encapsulated object!

The superclass acts as an interface for all of its subclasses.

An application won't know—or need to know—what sort of `MovableBall` it is using; the ball responds to all the same messages.

Composition uses substitution to reuse the code in the base class, but in a way that is controlled by the decorator.

This allows us to add the same behavior to **all** of the classes in the hierarchy!

In a way, the decorator pattern allows us to add new behavior to a single instance, rather than to the whole class.

Using a decorator makes sense any time there are two varieties of some object, but the variations are not directly related.

- balls that *bounce* of the walls and balls that *decelerate*