Decorator

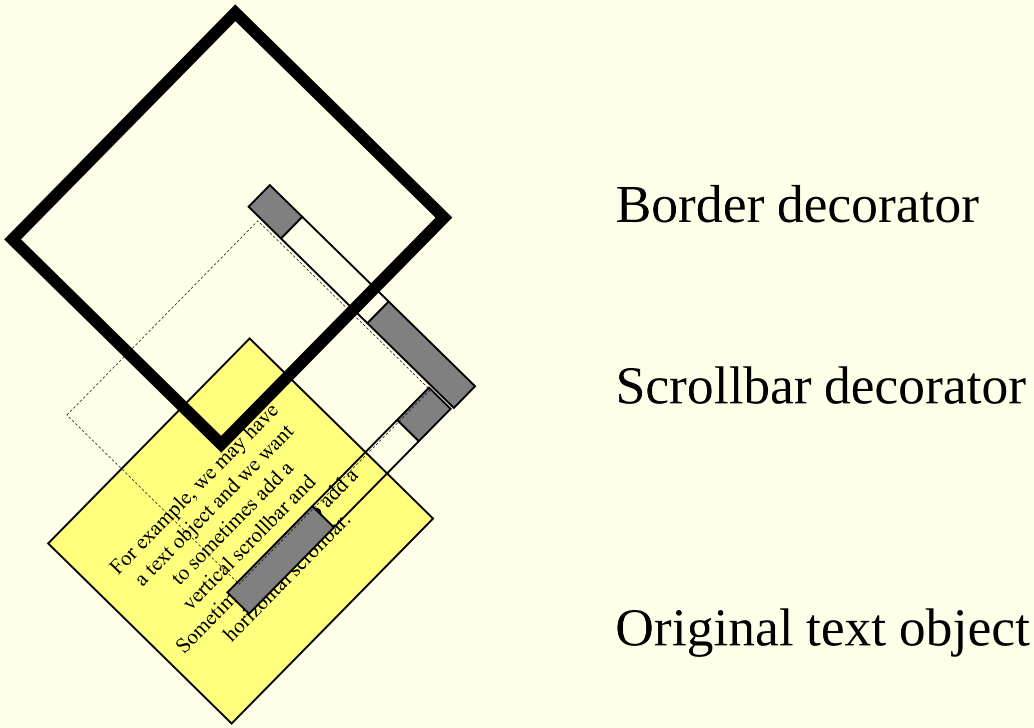
Sometimes we need a way to add responsibilities to an object dynamically and transparently.

The Decorator pattern gives a mechanism without using inheritance.

The Decorator pattern allows one to add and remove layers to a base object.

Decorator

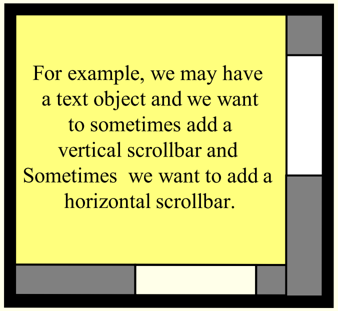
For example, we may have a text object and we want to sometimes add a scrollbar and sometimes we want to add a border.



Border decorator

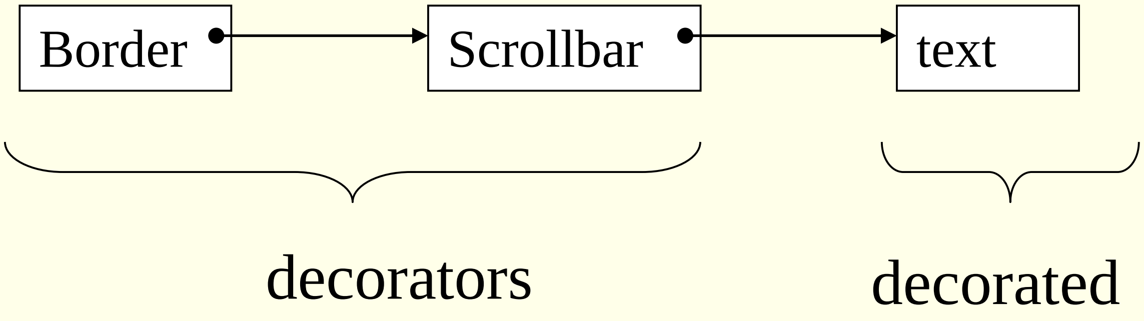
Scrollbar decorator

Original text object



For example, we may have a text object and we want to sometimes add a vertical scrollbar and Sometimes we want to add a horizontal scrollbar.

Decorator

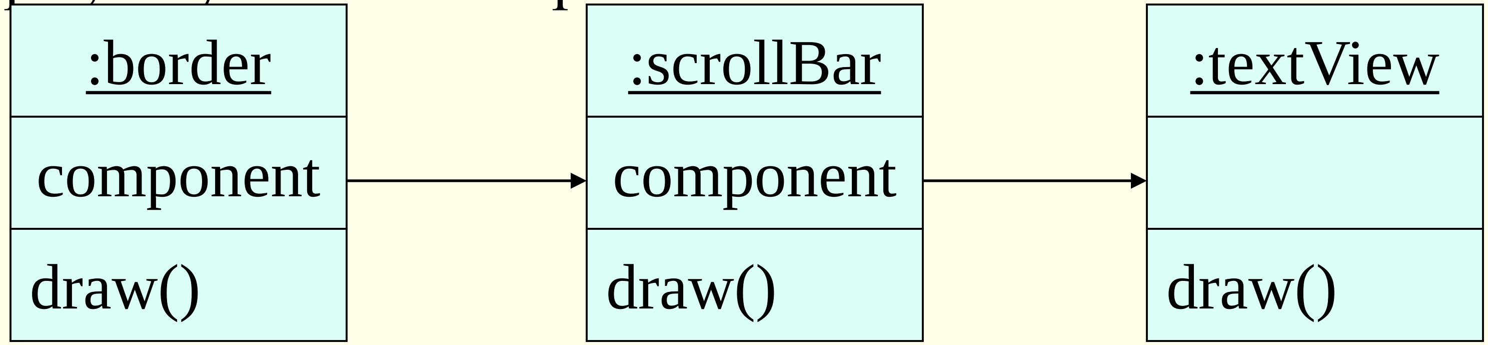


The objects are linked (a linked list or chain of objects).

The last in the list is the decorated object.

Decorator

e.g. In a windowing environment, scrolling bars, borders, etc. could be *decorators* on top of the text view of a document. In this example, they are all “components”



:border

component

draw()

:scrollBar

component

draw()

:textView

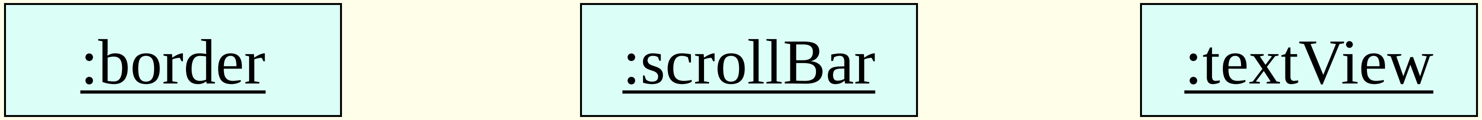
draw()

When it’s necessary for the document to appear (to draw itself), the draw message would be sent to :border and then:

- :border would draw itself;
- :border would send the draw message to :scrollBar which would draw itself;
- :scrollBar would send the draw message to :textView which would draw itself

Decorator

If draw is sent to :border , as discussed on previous slide, what is the sequence diagram?

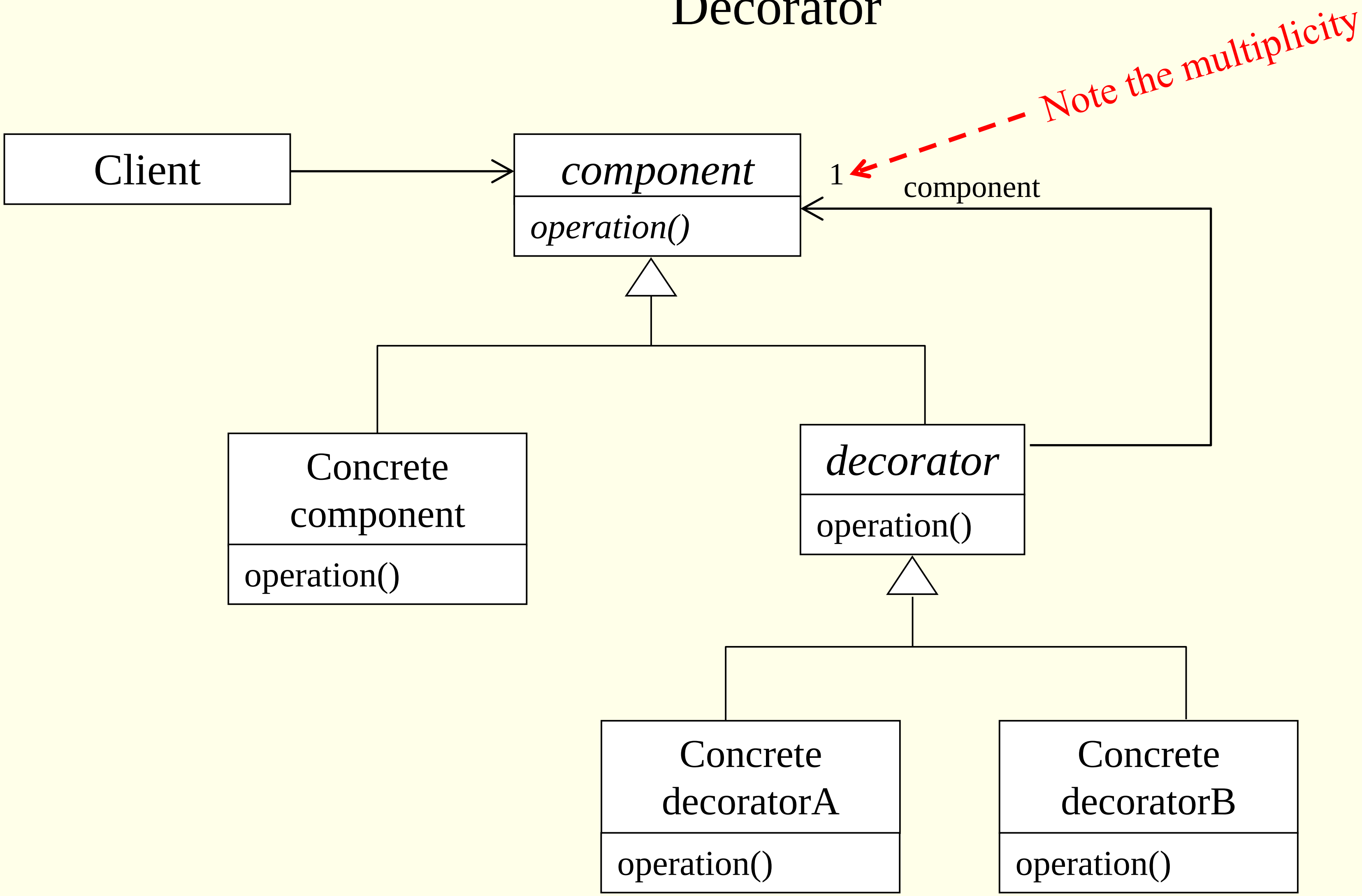


:border

:scrollBar

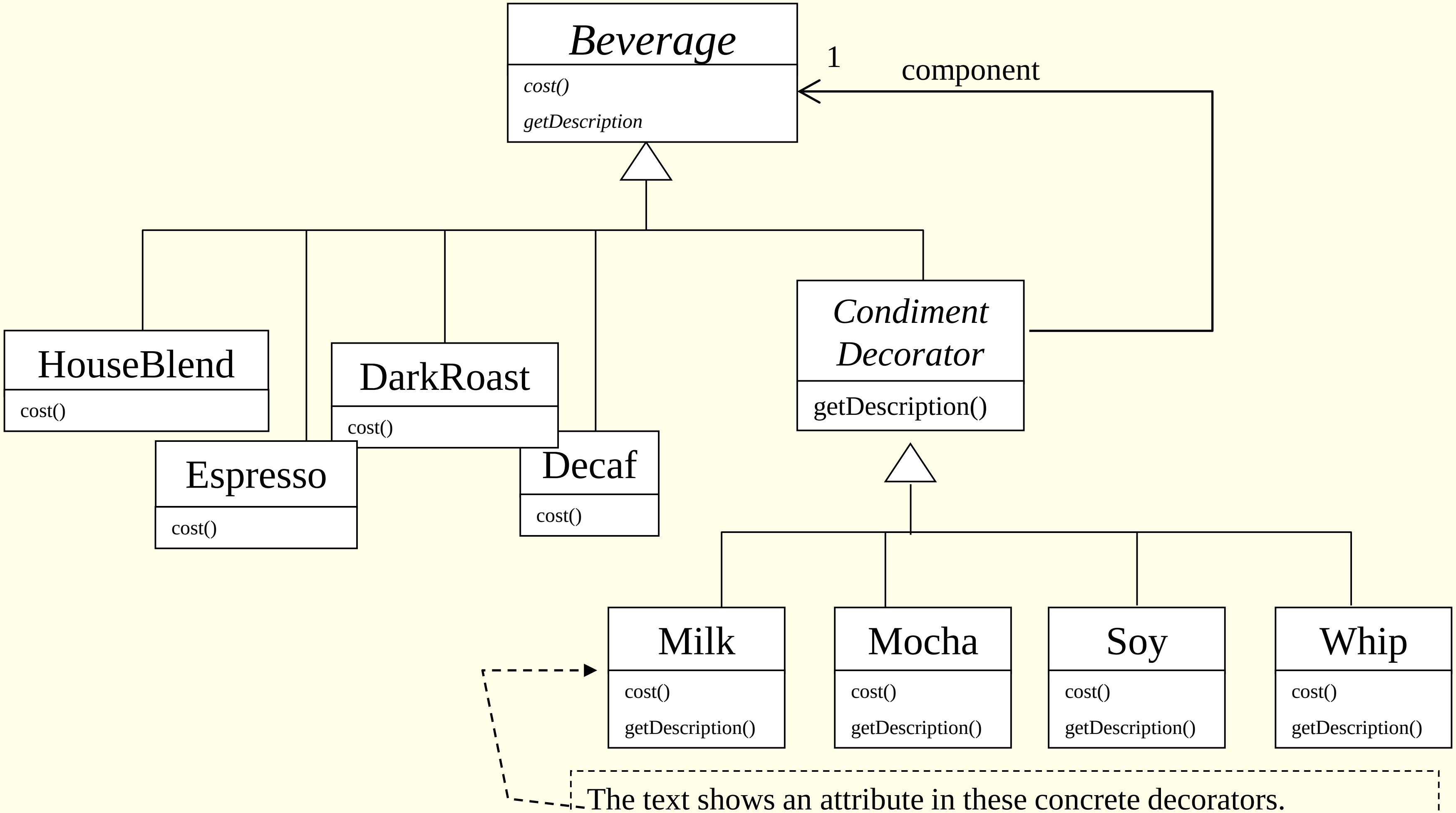
:textView

Decorator



See page 93

Decorator



The text shows an attribute in these concrete decorators.

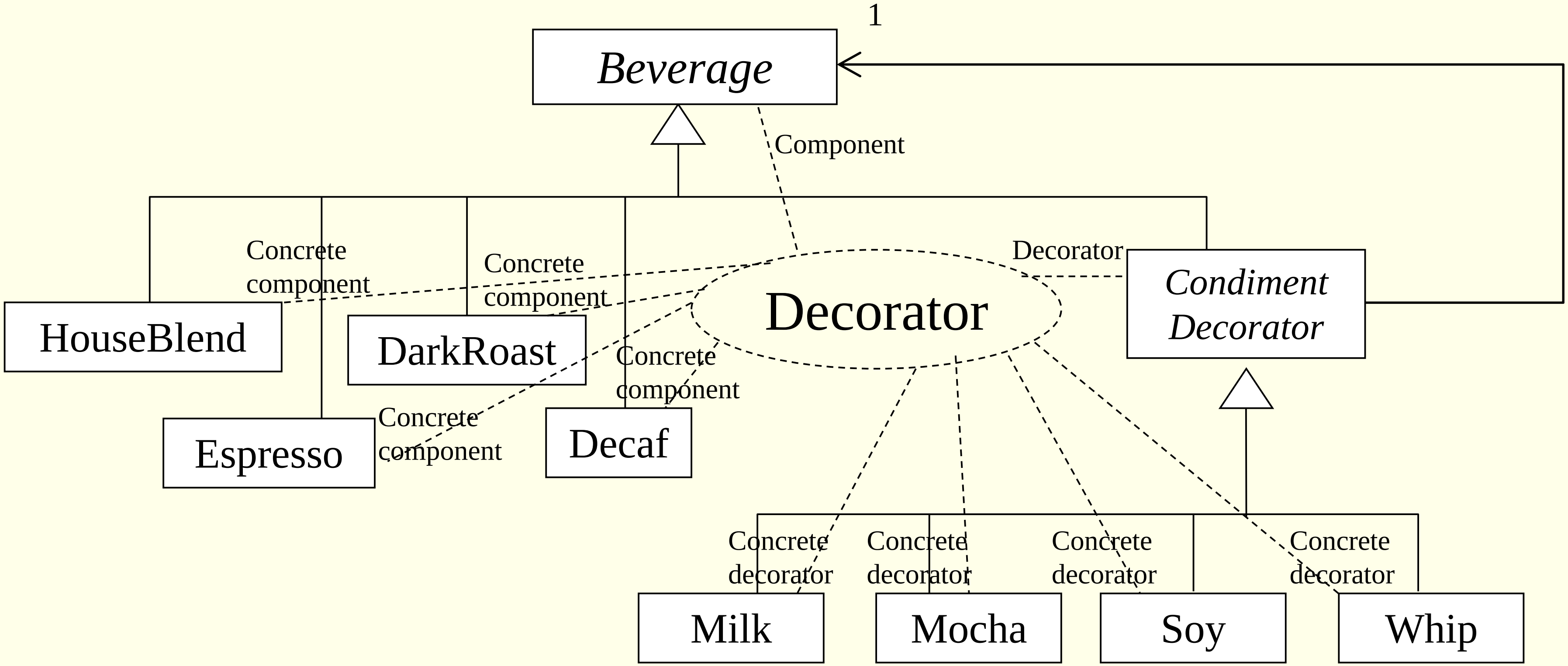
An ***implementation*** needs a reference to the next component.

This is implied by the reflexive association.

See page 94

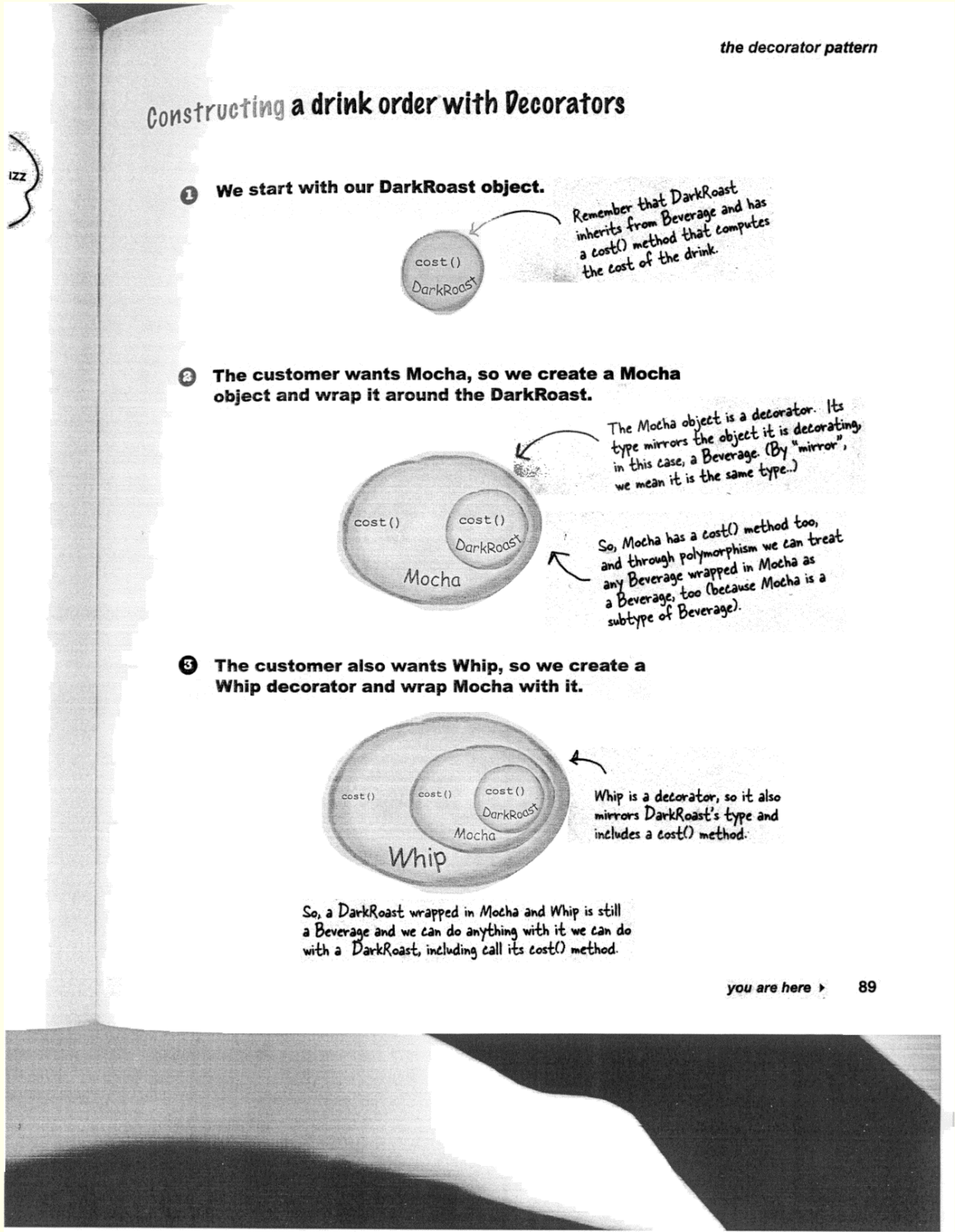
Question: What is the object diagram for a whipped mocha decaf?

Decorator



The class diagram augmented to show the roles the classes/objects play in the decorator collaboration

Constructing a drink



Constructing a drink order with Decorators

1. **We start with our DarkRoast object.**

cost()
DarkRoast

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

2. **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**

cost()
cost()
DarkRoast
Mocha

The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

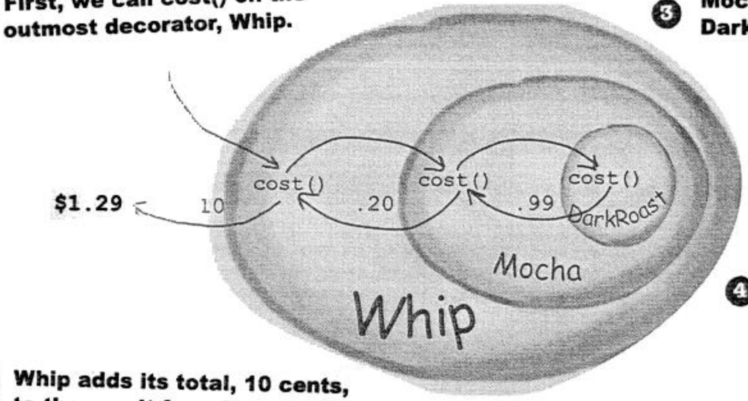
So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

3. **The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.**

Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

4 **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



1 **First, we call cost() on the outmost decorator, Whip.**

2 **Whip calls cost() on Mocha.**

(You'll see how in a few pages.)

3 **Mocha calls cost() on DarkRoast.**

4 **DarkRoast returns its cost, 99 cents.**

5 **Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.**

5 **Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.**

Okay, here's what we know so far...

- Decorators have the same supertype as the objects they decorate.
- You can use one or more decorators to wrap an object.
- Given that the decorator has the same supertype as the object it decorates, we can pass around a decorated object in place of the original (wrapped) object.
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.
- Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Key Point!

Now let's see how this all really works by looking at the Decorator Pattern definition and writing some code.

Decorator Pattern - example

Consider a POS system. Suppose this system must produce a sales receipt. A sales receipt will have a header and a footer, and perhaps more than one header … and more than one footer. Let’s assume the print() method of Receipt results in the receipt’s lines being printed

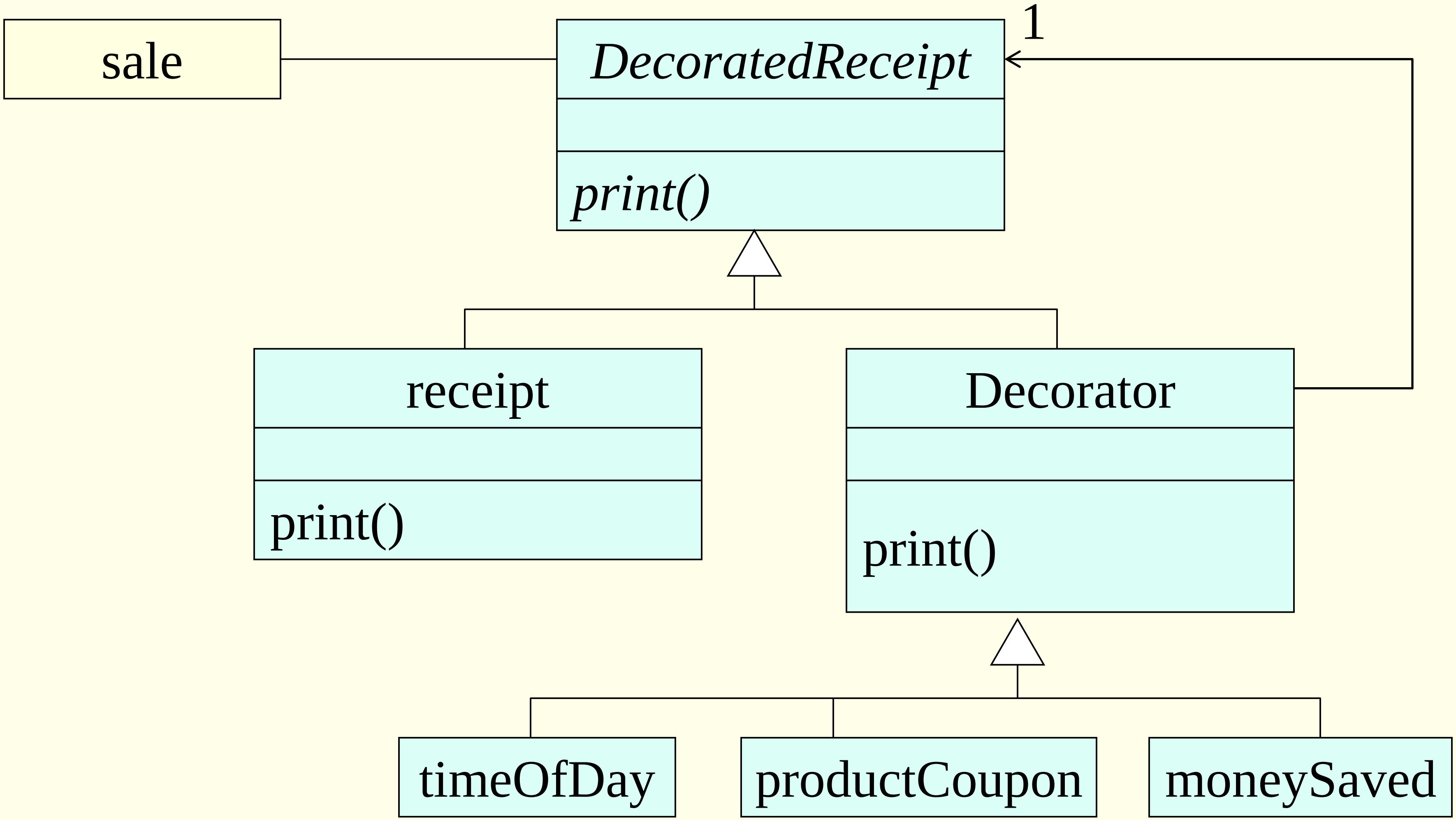
Suppose we add coupons to the sales receipt … perhaps based on the products purchased / the season / information about the customer / etc.

Time of day header
Product2 coupon header

Line item 1
Line item 2
Line item 3
…
Money saved footer

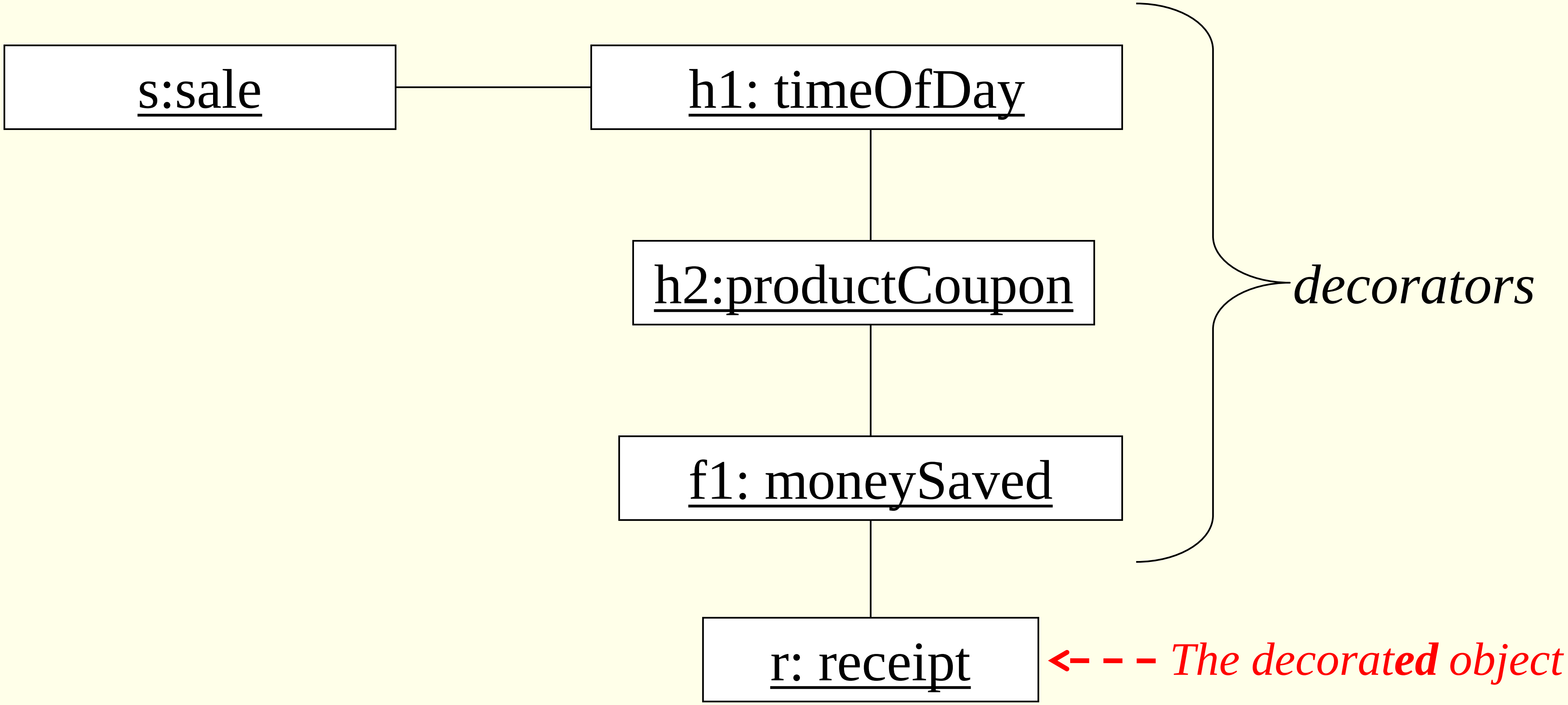
Decorator Pattern - example

UML class diagram

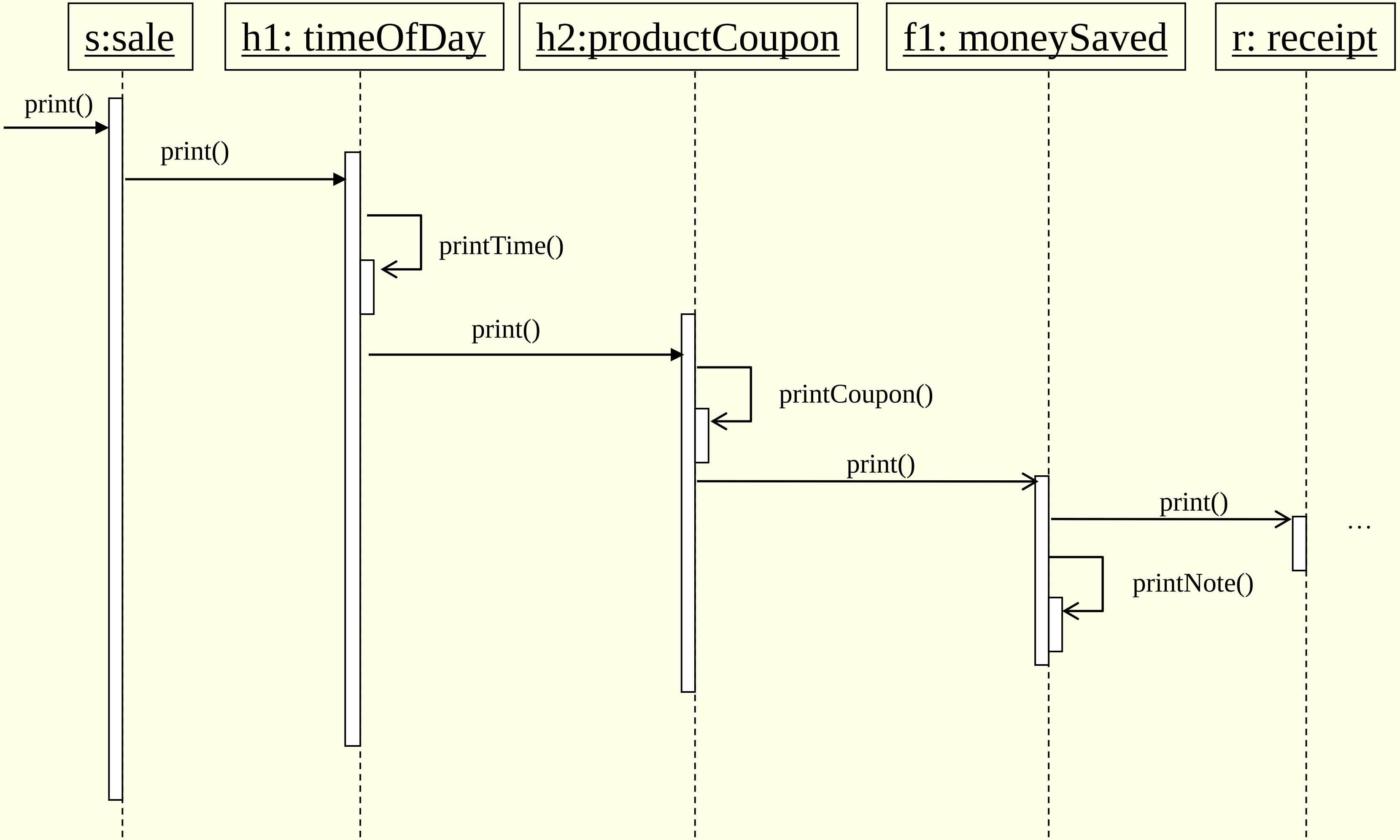


Decorator Pattern – example object diagram

a sale object is related to a receipt, but the receipt is decorated with headers and footers (as a particular receipt requires)



Decorator Pattern - example
Printing the receipt


s:sale
h1: timeOfDay
h2:productCoupon
f1: moneySaved
r: receipt
print()
print()
printTime()
print()
printCoupon()
print()
print()
…
printNote()