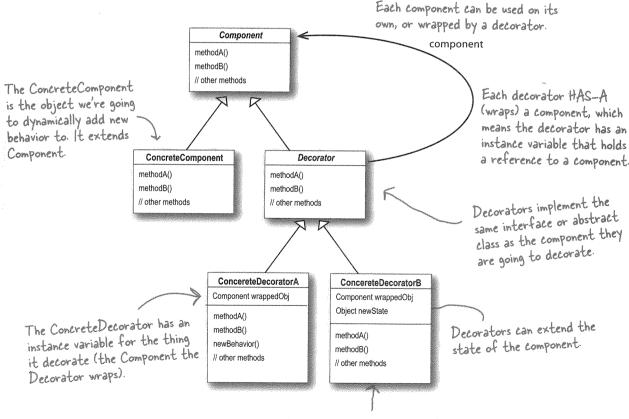# The Decorator Pattern defined

Let's first take a look at the Decorator Pattern description:

> **The Decorator Pattern** attaches additional
> responsibilities to an object dynamically.
> Decorators provide a flexible alternative to
> subclassing for extending functionality.

While that describes the *role* of the Decorator Pattern, it doesn't give us a lot
of insight into how we'd *apply* the pattern to our own implementation. Let's
take a look at the class diagram, which is a little more revealing (on the next
page we'll look at the same structure applied to the beverage problem).

Each component can be used on its
own, or wrapped by a decorator.

component

Component

methodA()
methodB()
// other methods

The ConcreteComponent
is the object we're going
to dynamically add new
behavior to. It extends
Component.

Each decorator HAS-A
(wraps) a component, which
means the decorator has an
instance variable that holds
a reference to a component.

ConcreteComponent

methodA()
methodB()
// other methods

Decorator

methodA()
methodB()
// other methods

Decorators implement the
same interface or abstract
class as the component they
are going to decorate.

ConcereteDecoratorA

Component wrappedObj

methodA()
methodB()
newBehavior()
// other methods

ConcereteDecoratorB

Component wrappedObj
Object newState

methodA()
methodB()
// other methods

The ConcreteDecorator has an
instance variable for the thing
it decorate (the Component the
Decorator wraps).

Decorators can extend the
state of the component.

Decorators can add new methods; however, new
behavior is typically added by doing computation
before or after an existing method in the component.