Decorator Design Pattern

Learning outcomes

- Understand when to use decorators
- Distinguish between decorators and strategy design pattern
- Understand how decorators are used in java.io.* package.

Creating chaos with Inheritance

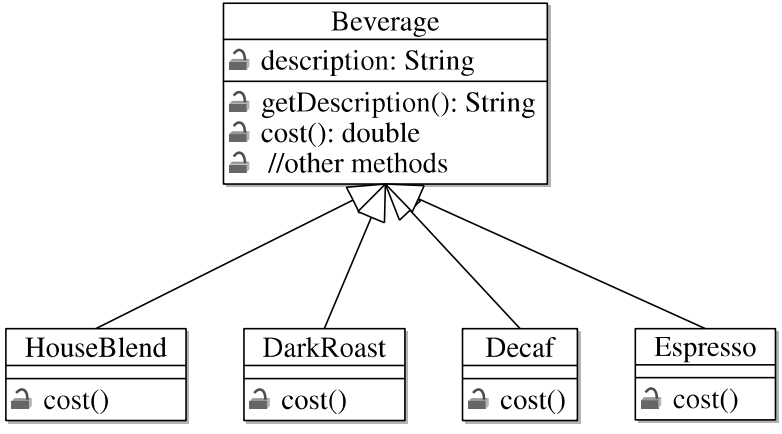
Example: Coffee shop description
Coffee shop serves and takes payments for beverages

House Blend	£1.99
Dark Roast	£2.30
Decaf	£2.50
Espresso	£4.00

you can add a number of condiments to the beverages like soy, whip, milk, mocha, etc. Each condiment has a small cost in addition to the cost, of the coffee.

Creating chaos with Inheritance

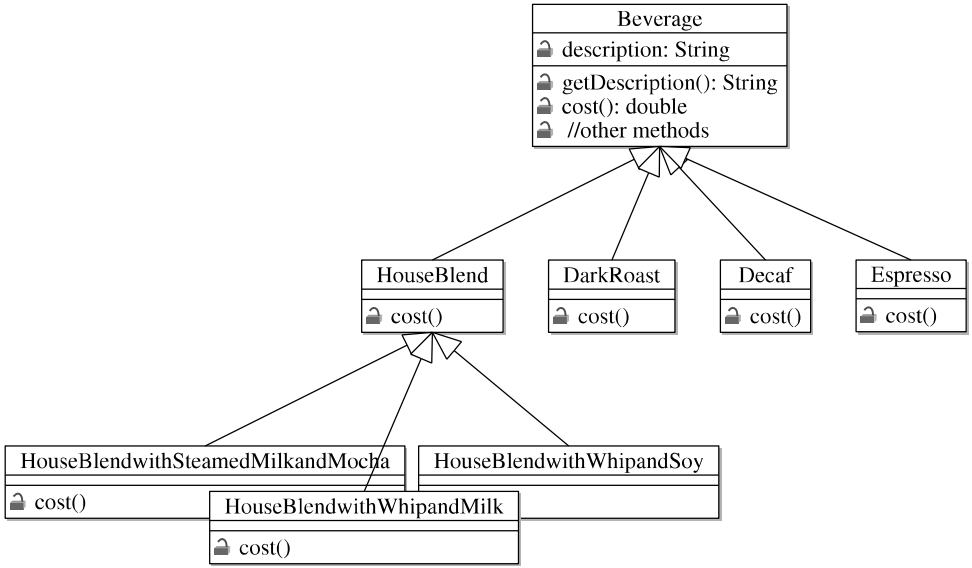
Example: Coffee shop design



- Each subclass overrides the cost method to calculate the cost of a particular coffee.

Creating chaos with Inheritance

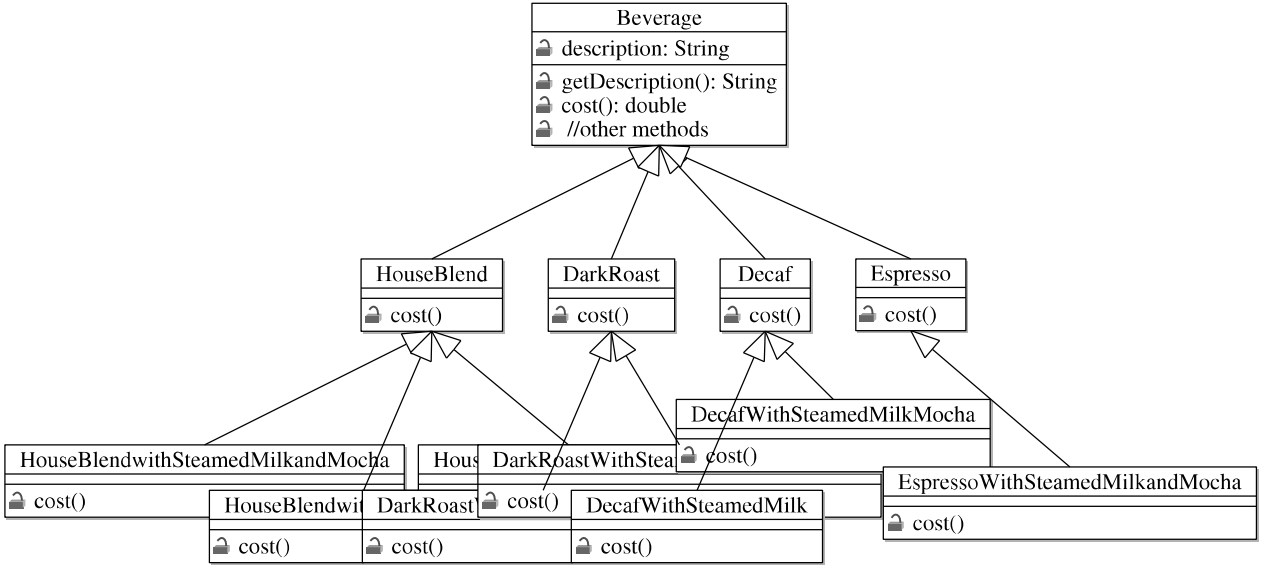
Example: Coffee shop design



- But there are many variants of these beverages, like a decaf with soy, etc. Thus, for such variance we can add a few more beverages.

Creating chaos with Inheritance

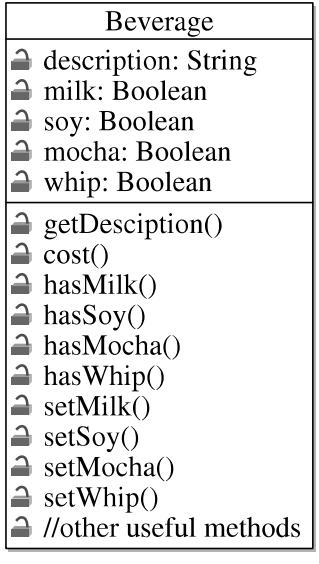
Example: Coffee shop design



- The variance may be quite exhaustive and the inheritance hierarchy quickly become unmanageable

Creating chaos with Inheritance

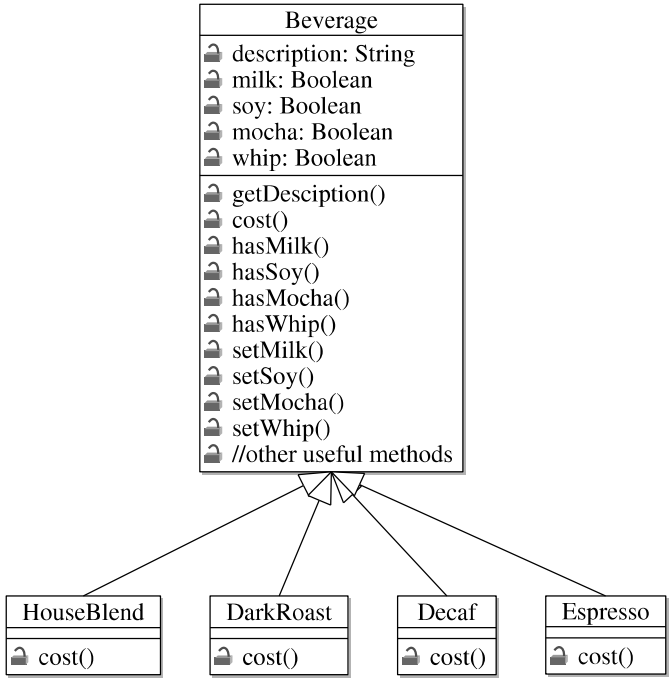
Coffee shop redesign



- Object properties are used to keep track of the beverages and the option
  - fields in the superclass tracks condiments in the drink

Creating chaos with Inheritance

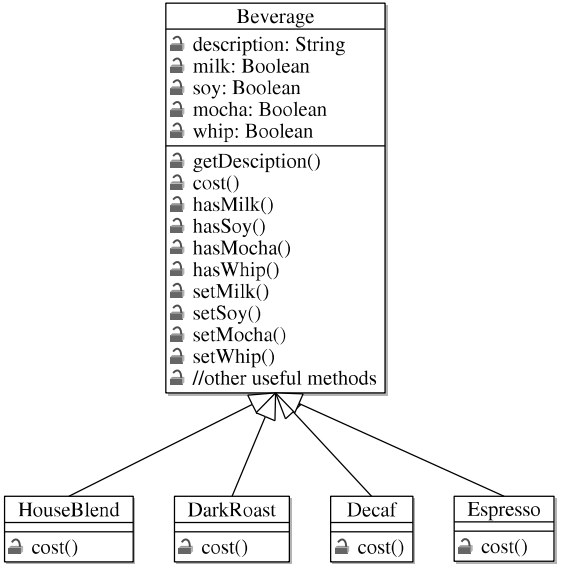
Coffee shop redesign



- We then subclass the beverage superclass again with each type

Creating chaos with Inheritance

Coffee shop redesign

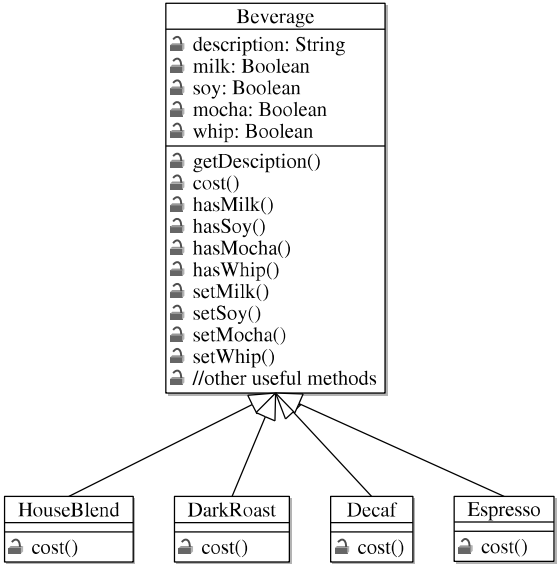



```
HouseBlend drink = new HouseBlend();
drink.setSoy();
drink.setWhip();
double cost = drink.cost();
```

- Costing for HouseBlend with soy and whip

Creating chaos with Inheritance

Coffee shop redesign

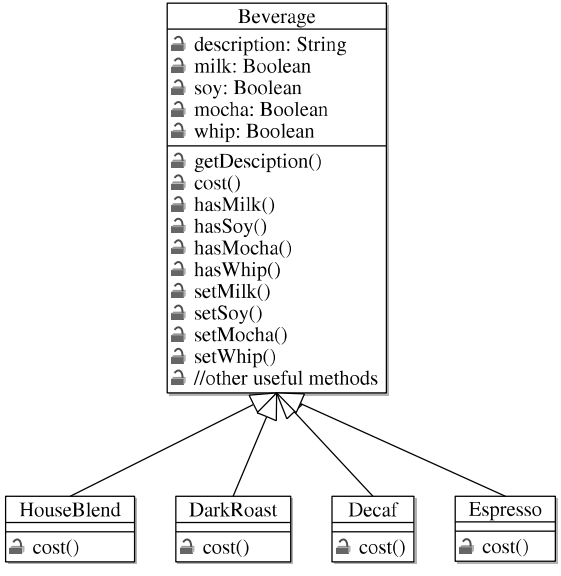



```
if(hasMilk()){
    cost += .10;
}
if(hasSoy()){
    cost += .20;
}
if(hasMocha()){
    cost += .15;
}
...
```

- This approach depends on the cost method in each Beverage subclass (looks simpler)

Creating chaos with Inheritance

Coffee shop redesign



But:

- Price changes will alter potentially all existing code
- New condiments require changing the superclass
- Condiments are not appropriate for some beverages
- We cannot handle all orders (like a double mocha)

- Not quite flexible or maintainable design

Creating chaos with Inheritance

Coffee shop redesign
Problem:

- We know in the future, we'll have to support new beverage types.
- That means we'll have to keep modifying existing code.
- But, that's exactly what we don't want.
  - We want to leave our design **open** for different beverage types, but **closed** in the sense we don't want to touch existing code.

Object Oriented Design Principles

Design Principle #5:

Classes should be open for extension, but closed for modification.

Recap from Strategy Pattern

Inheritance

- Inheritance is powerful but it can lead to inflexible design.
- When we subclass, we are forced to make static, compile time decisions.
- Also, all classes inherit the same behaviour.

Recap from Strategy Pattern

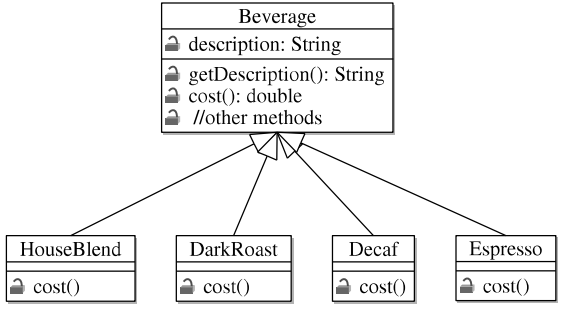
Composition

- We get more flexible and adaptable design.
- We can still take on new behaviour, but we achieve it by composing objects together.
- We can make dynamic runtime decisions.
  - add behavior without altering existing code.
  - We can even go as far as adding new behavior that the creator of a class never considered.

Recap from Strategy Pattern

Decorator applies composition in a new way that's different from strategy pattern.

Extending behaviour with composition

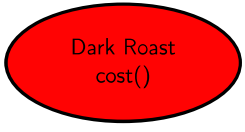


Our first take on the beverage hierarchy was not too bad until when we started thinking of condiments

Extending behaviour with composition

Redesigning the Coffee Shop

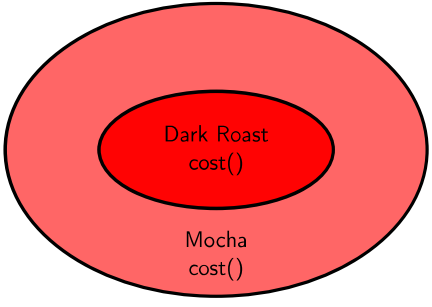
- Say a customer wants a dark roast with mocha and whip.



- Rather than using a specific class for the entire beverage (like we did before), we do object composition. Thus:

Extending behaviour with composition

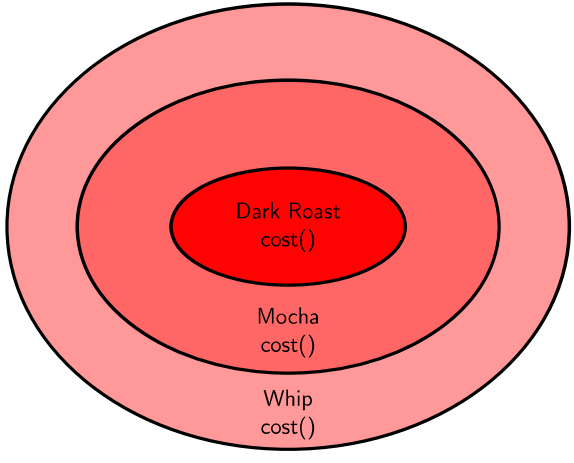
Redesigning the Coffee Shop



1. First, we create a mocha object and wrap it around or compose it with, the dark roast object.

Extending behaviour with composition

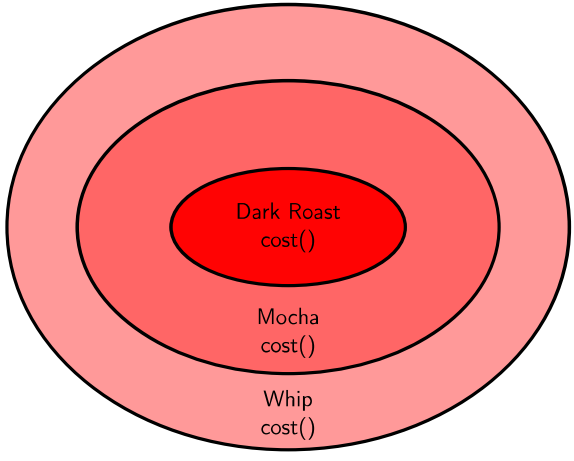
Redesigning the Coffee Shop



2. Then we create a whip object and wrap it around or compose it with the mocha object.

Extending behaviour with composition

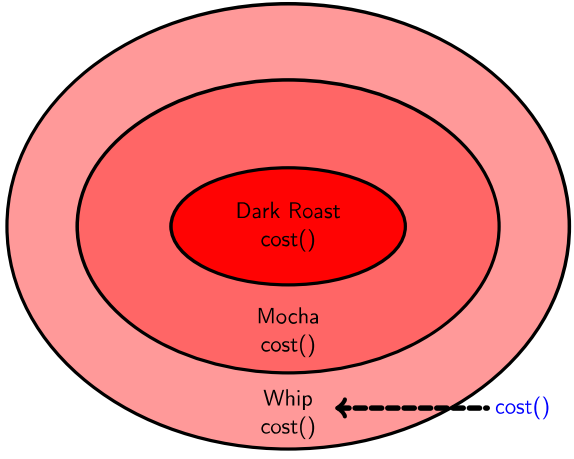
Redesigning the Coffee Shop



- Each of the objects look alike (they all have cost method) and they are all responsible for their own part of the cost.

Extending behaviour with composition

Redesigning the Coffee Shop

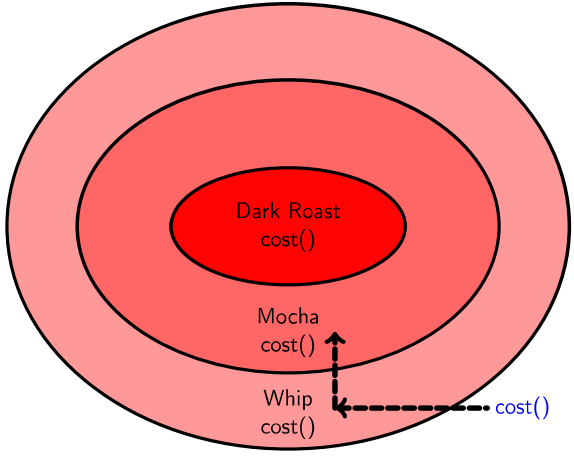


To determine the cost of coffee:

- First we call cost on the outermost object

Extending behaviour with composition

Redesigning the Coffee Shop

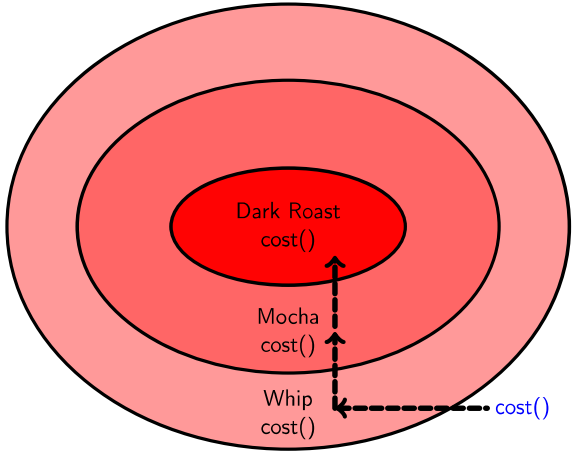


To determine the cost of coffee:

- which then delegates the cost to the next object

Extending behaviour with composition

Redesigning the Coffee Shop

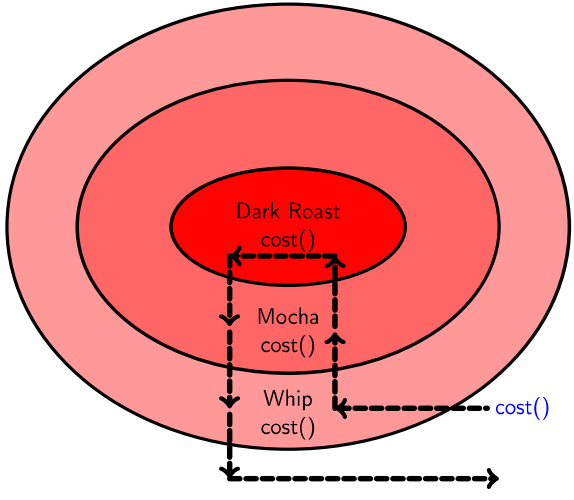


To determine the cost of coffee:

- which then delegates the cost to the next object, when then delegates to the next object etc.

Extending behaviour with composition

Redesigning the Coffee Shop



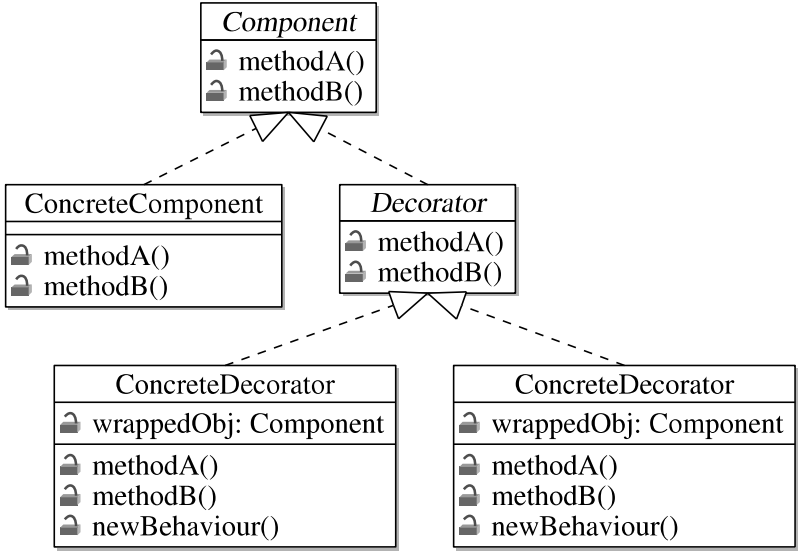
To determine the cost of coffee:

- Each delegation returns a value which is added to the calling object cost. The final returned cost is the sum of all cost.

The Decorator Pattern

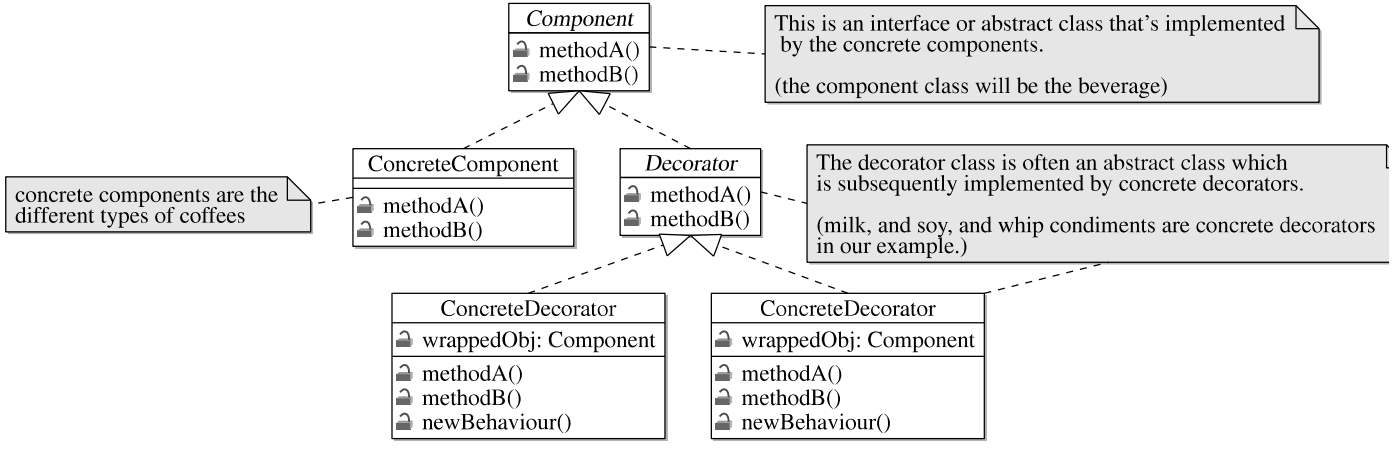
Definition
The *decorator pattern* attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The Decorator Class Diagram

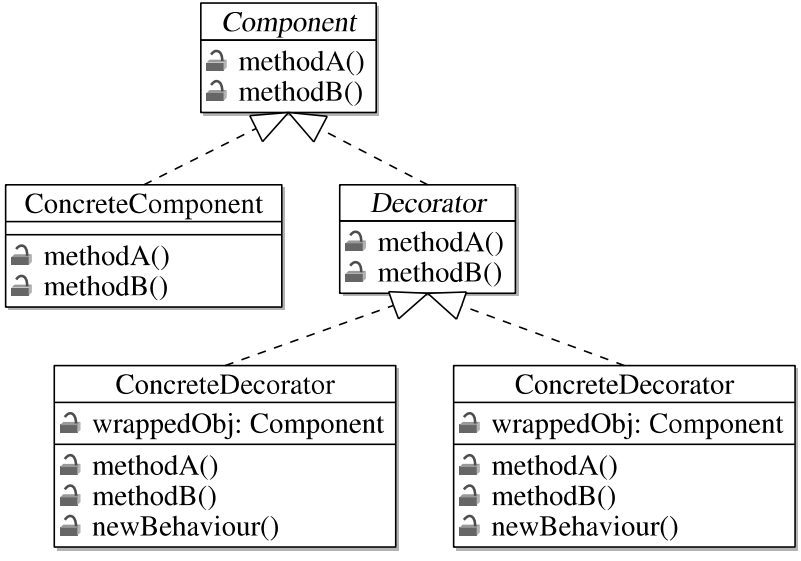


- The two important parts to the decorator pattern:
  1. *Components* (represented by beverages in our example), and
  2. *Decorators* (represented by condiments in our example)

The Decorator Class Diagram

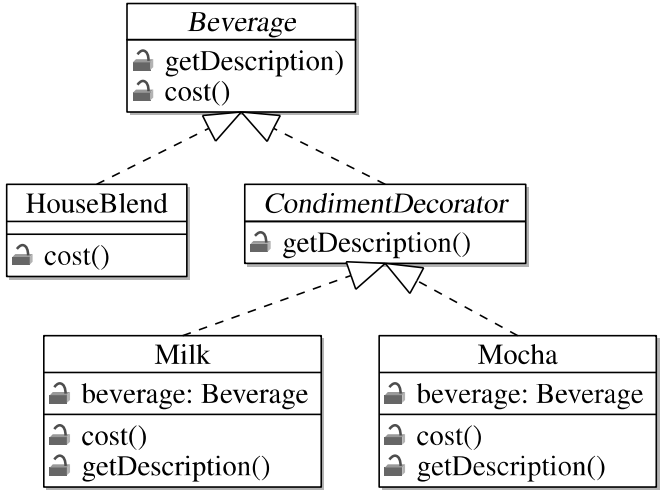


The Decorator Class Diagram



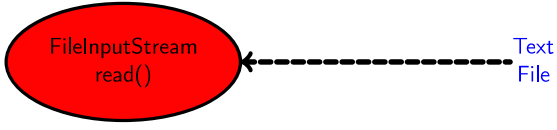
- Note that concrete components and the decorators implement the same component superclass.
  1. This is because we want to be able to wrap any decorator around any of the components.

The Decorator Class Diagram



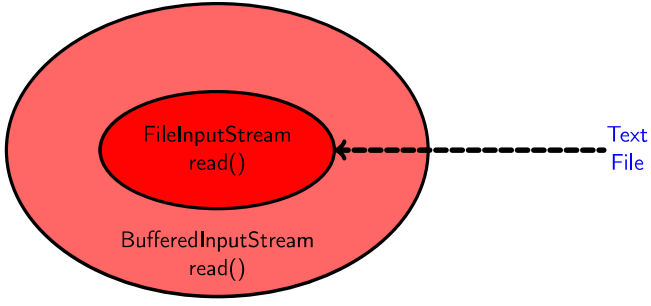
- Note that concrete components and the decorators implement the same component superclass.
  2. example, we want to be able to wrap any of the condiments around any of the coffees, and then cull the cost and get description methods on any of these wrapped objects.

Decorator in java libraries: java.io.*



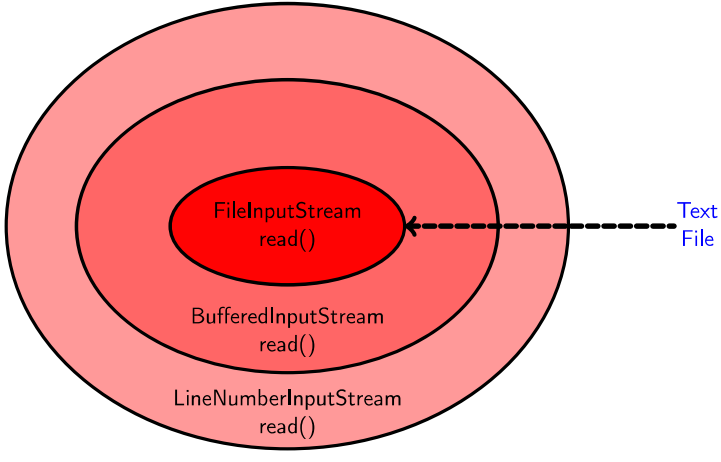
- For instance, the *FileInputStream* class is a component that you can use to read data from a file.

Decorator in java libraries: java.io.*



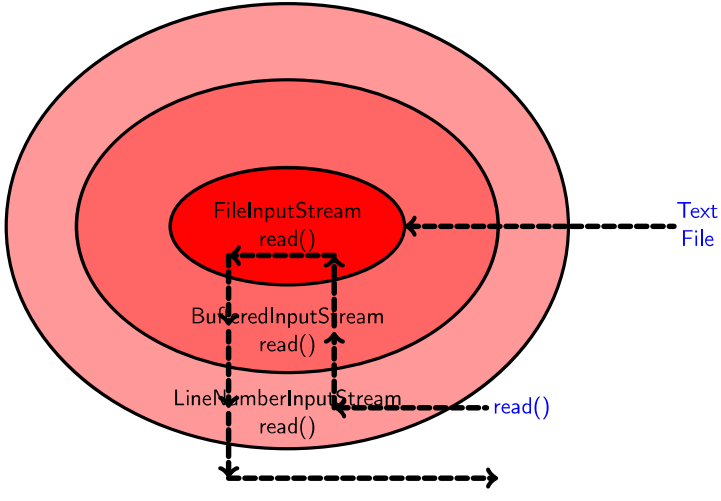
- You can decorate this component, with decorators such as *BufferedInputStream*, which buffers input to improve performance.

Decorator in java libraries: java.io.*



- and *LineNumberInputStream*, which counts line numbers as it reads data.

Decorator in java libraries: java.io.*



- The call to *read* is passed through the decoration layers to the main component, *FileInputStream*, with each decorator adding on its own functionality.

Class Diagram java.io.* package