Decorator Pattern

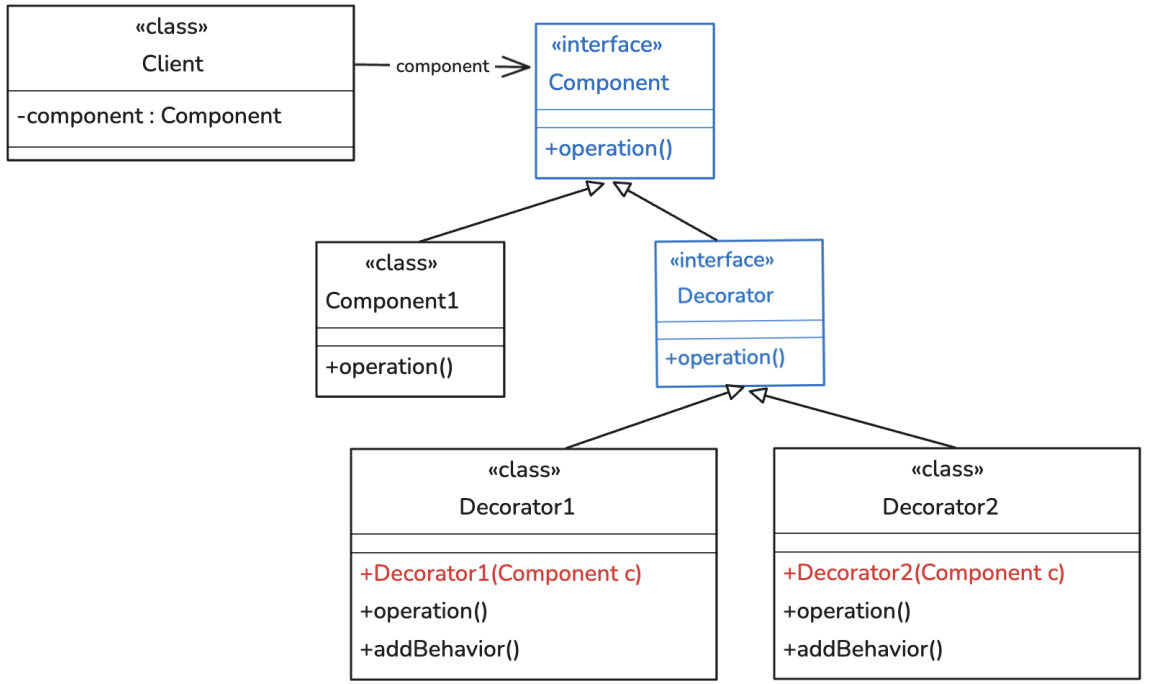
UNSW
SYDNEY

Decorator Pattern: Intent

- "Attach additional responsibilities to an object dynamically.
  Decorators provide a flexible alternative to sub-classing for extending functionality."
  [GoF]
- Decorator design patterns allow us to selectively add functionality to an object (not the class) at runtime, based on the requirements.
- Original class is not changed (Open-Closed Principle).
- Inheritance extends behaviors at compile time, additional functionality is bound to all the instances of that class for their life time.
- The decorator design pattern prefers a composition over an inheritance.
  Its a structural pattern, which provides a wrapper to the existing class.
- Objects can be decorated multiple times, in different order, due to the recursion involved with this design pattern. See the example in the Demo.
- Do not need to implement all possible functionality in a single (complex) class.

Decorator Pattern: Structure

- *Client* : refers to the Component interface.
- *Component*: defines a common interface for *Component1* and *Decorator* objects
- *Component1* : defines objects that get decorated.
- *Decorator*: maintains a reference to a *Component* object, and forwards requests to this component object (*component.operation()*)
- *Decorator1, Decorator2, ...* : Implement additional functionality (*addBehavior()*) to be performed before and/or after forwarding a request.

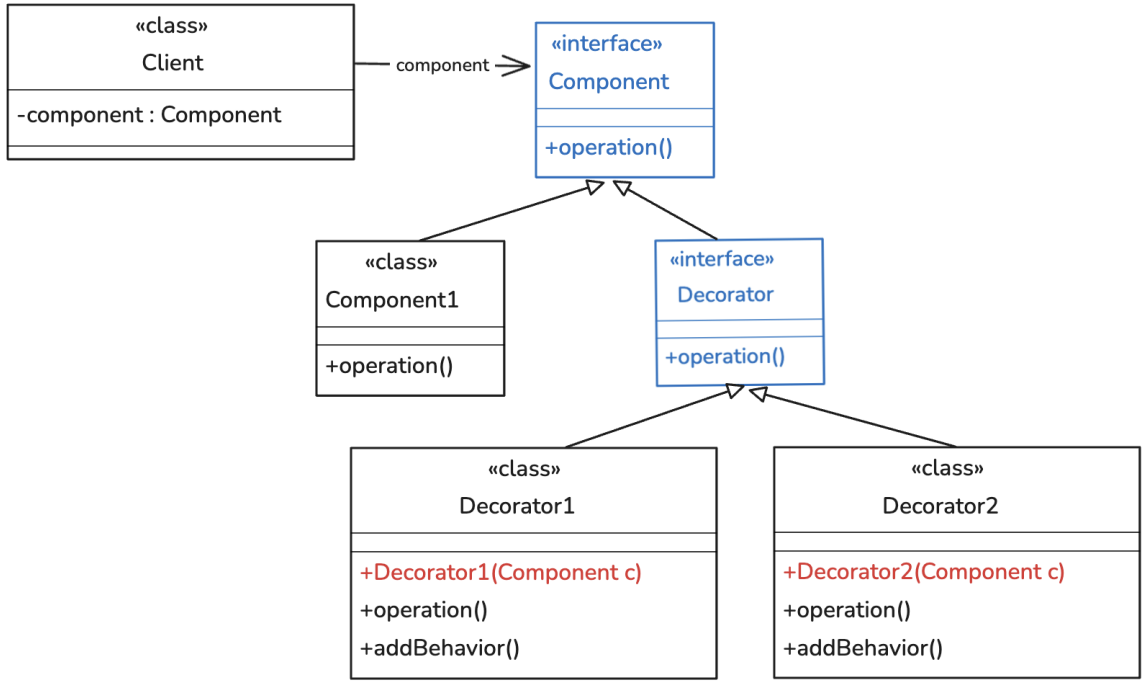


Decorator Pattern: Structure

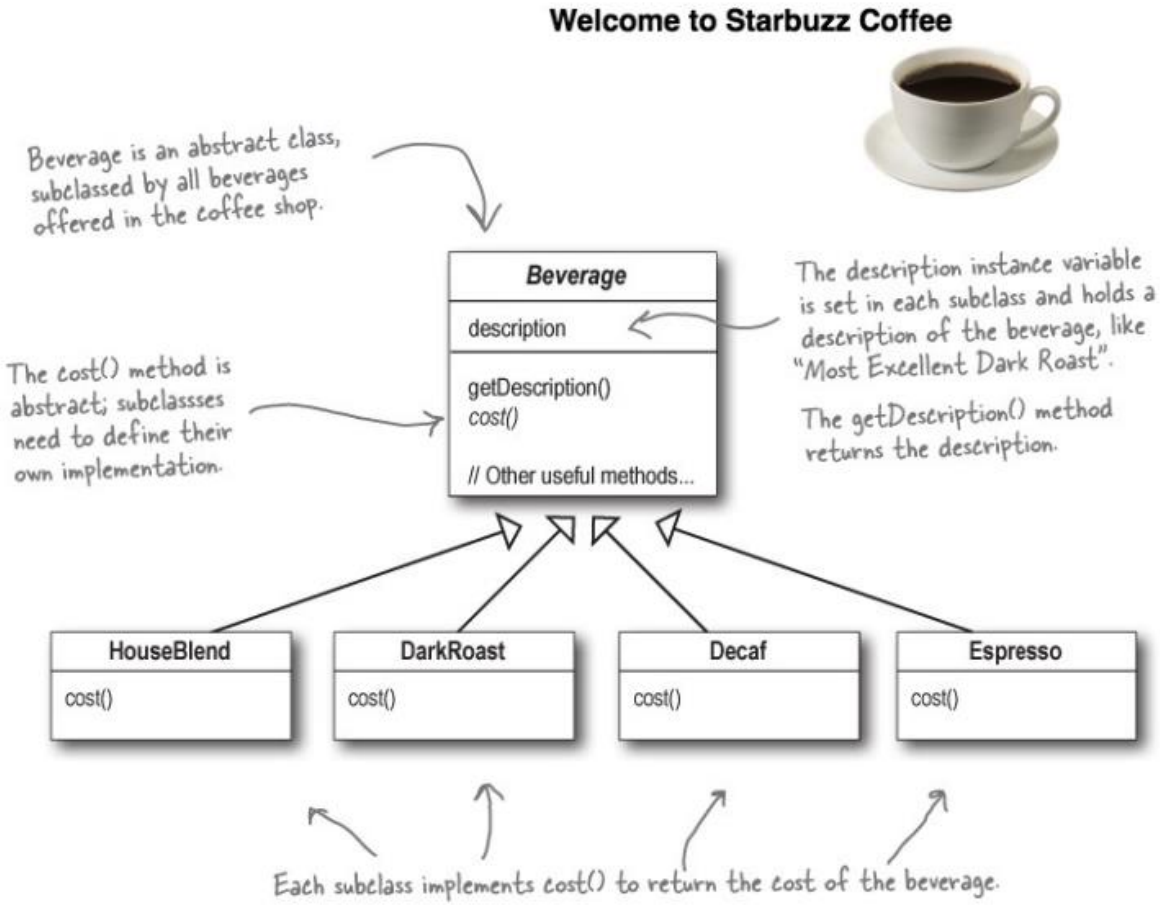
- Given that the decorator has the same supertype as the object it decorates,

  we can pass around a decorated object in place of the original (wrapped) object.

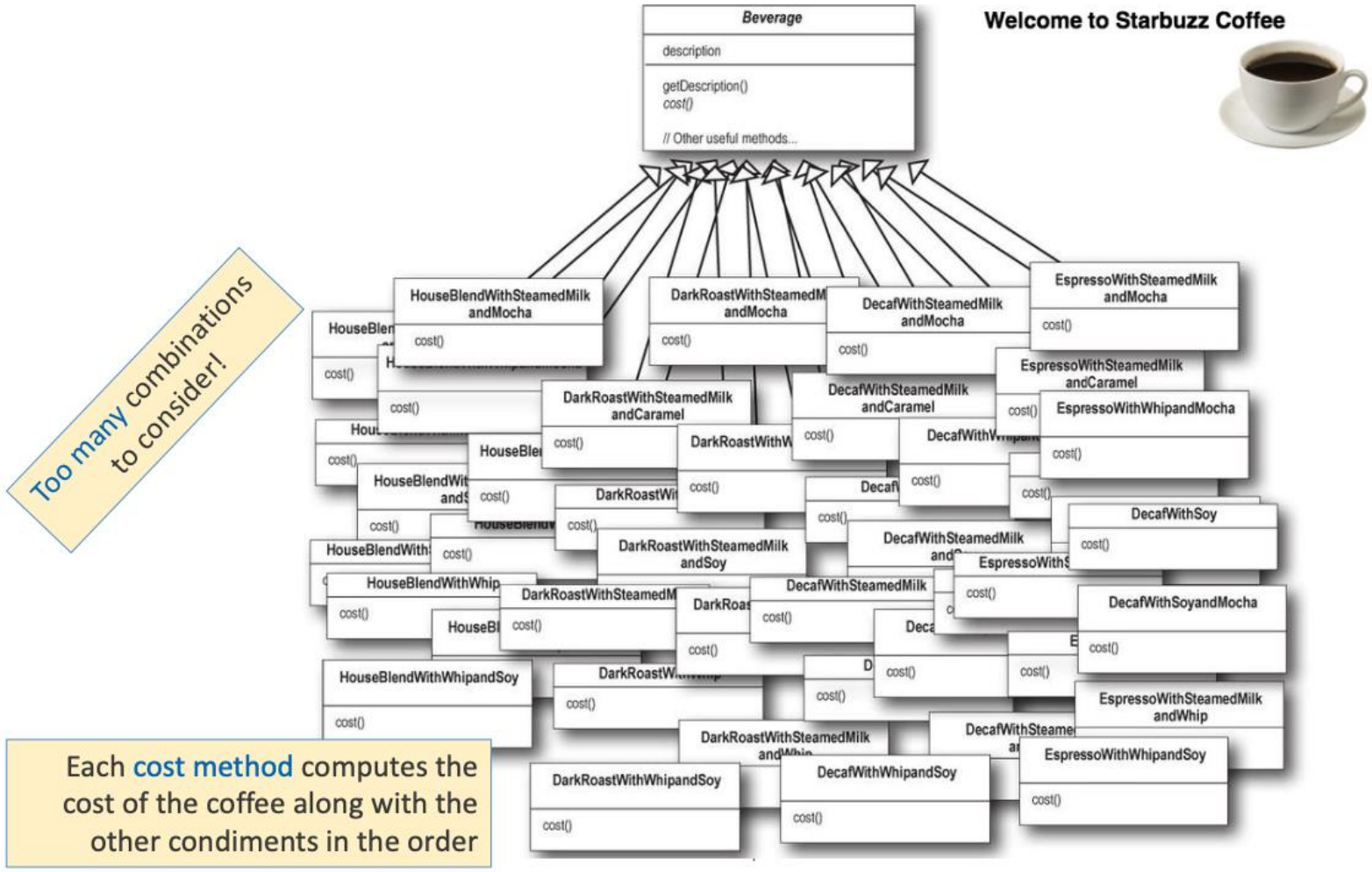
- The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.



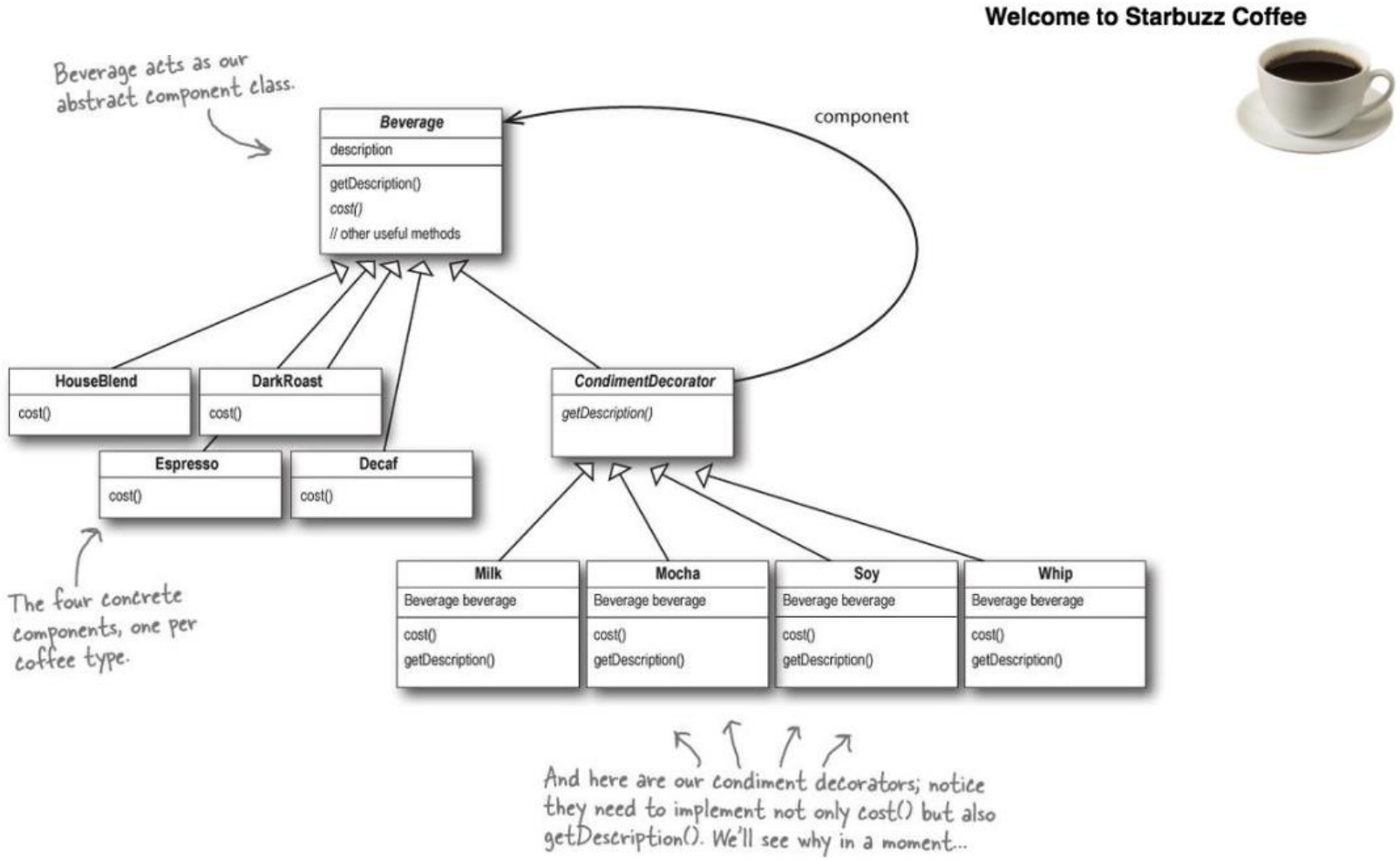
Decorator Pattern: Example



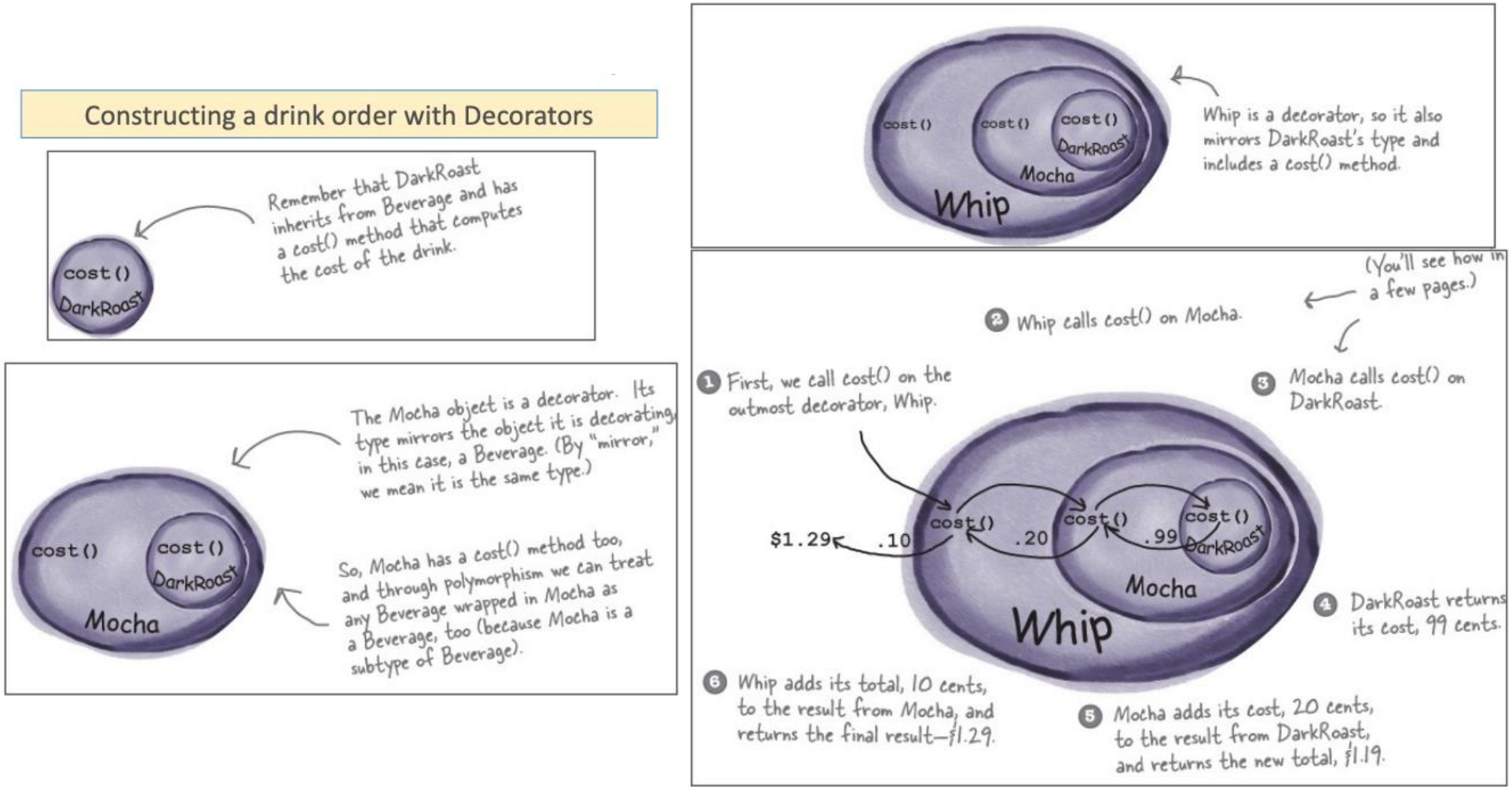
Decorator Pattern: Example



Decorator Pattern: Example



Decorator Pattern: Example



Decorator Pattern: Code

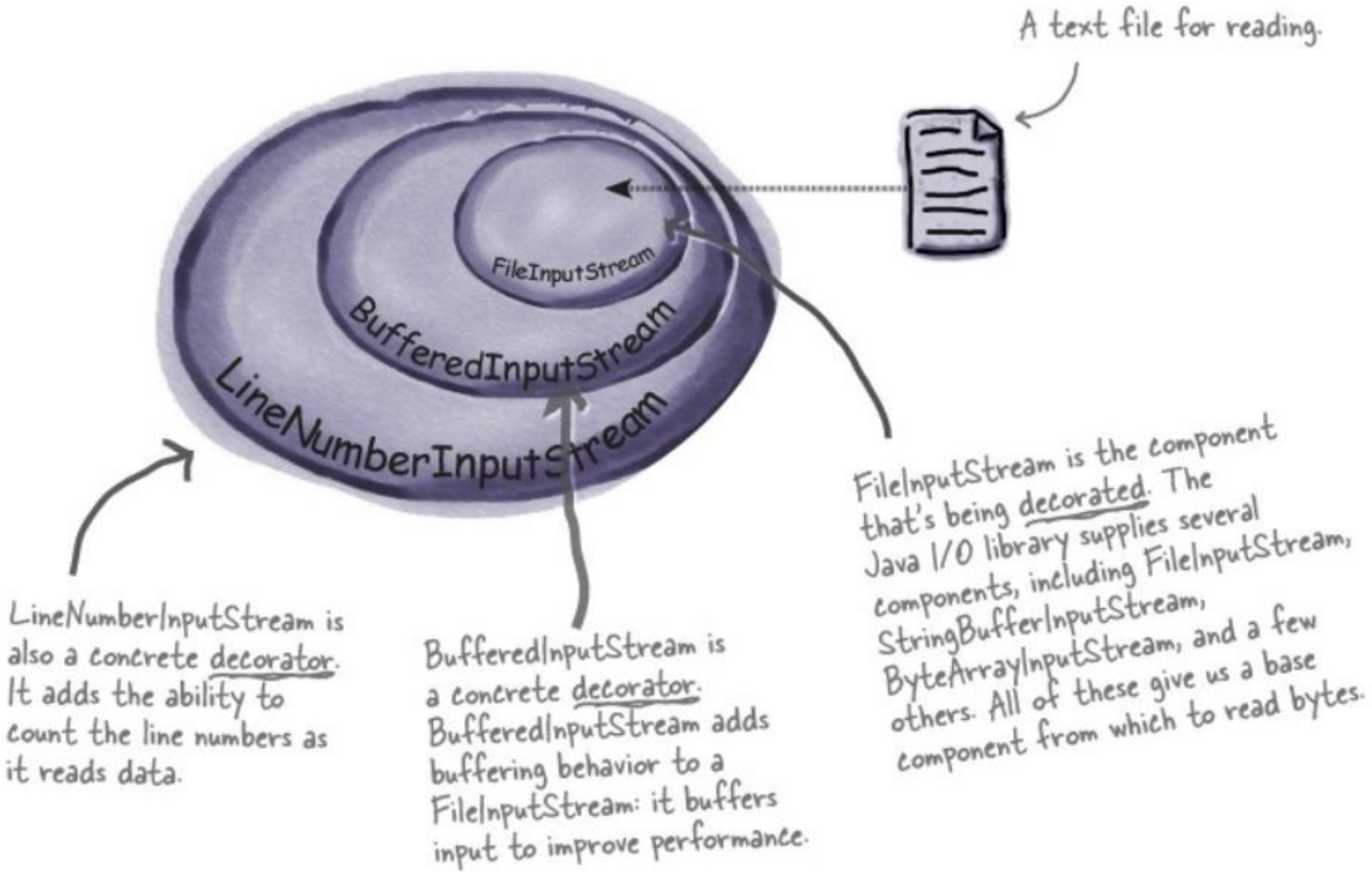
```
Beverage beverage = new Espresso();
System.out.println(beverage.getDescription()
        + " $" + beverage.cost());
System.out.println("-------------------------
Beverage beverage2 = new DarkRoast();
beverage2 = new Mocha(beverage2);
beverage2 = new Mocha(beverage2);
beverage2 = new Whip(beverage2);
System.out.println(beverage2.getDescription()
        + " $" + beverage2.cost());

System.out.println("------------------------- ");

Beverage beverage3 = new HouseBlend();
beverage3 = new Soy(beverage3);
beverage3 = new Mocha(beverage3);
beverage3 = new Whip(beverage3);
System.out.println(beverage3.getDescription()
        + " $" + beverage3.cost());
System.out.println("-------------------------
```

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Whipe: beverage.cost() is: " + beverage_cost);
    System.out.println("  - adding One Whip cost of 0.10c ");
    System.out.println("  - new cost is: " + (0.10 + beverage_cost ) );

    return 0.10 + beverage_cost ;
}
```

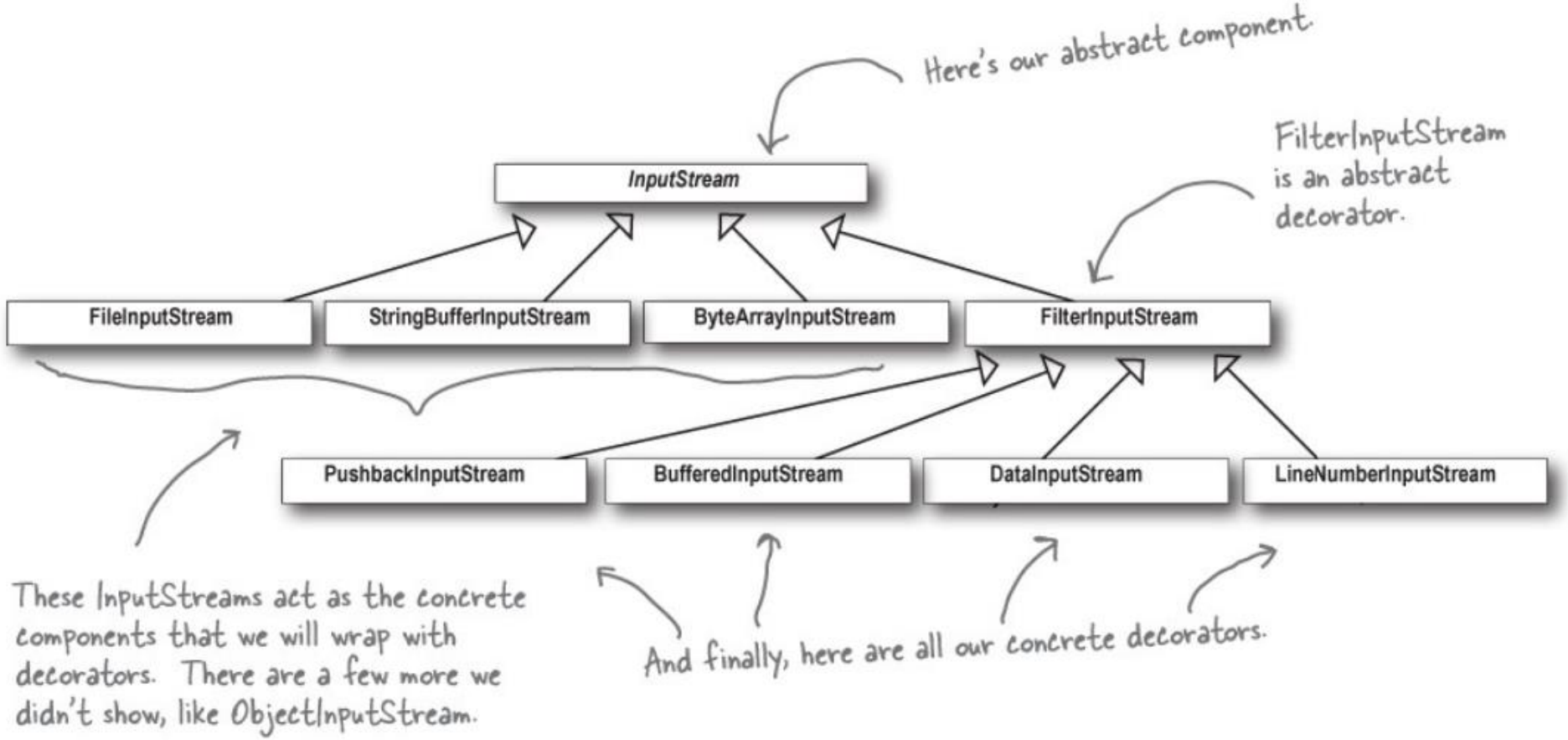
Read the example code
discussed/developed in the lectures,
and also provided for this week

```
public double cost() {
    double beverage_cost = beverage.cost();
    System.out.println("Mocha: beverage.cost() is: " + beverage_cost );
    System.out.println("  - adding One Mocha cost of 0.20c ");
    System.out.println("  - new cost is: " + (0.20 + beverage_cost ) );

    return 0.20 + beverage_cost ;
}
```

Decorator Pattern: Java I/O Example



Decorator Pattern: Java I/O Example



Decorator Pattern: Code

```
InputStream f1 = new FileInputStream(filename);
InputStream b1 = new BufferedInputStream(f1);
InputStream lCase1 = new LowerCaseInputStream(b1);
InputStream rot13 = new Rot13(b1);

while ((c = rot13.read()) >= 0) {
    System.out.print((char) c);
}
```

Read the example code discussed/developed in the lectures, and also provided for this week

Decorator Pattern:

…… Demo ……

End