Observer and Decorator

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 4448/6448 — Lecture 20 — 11/01/2007

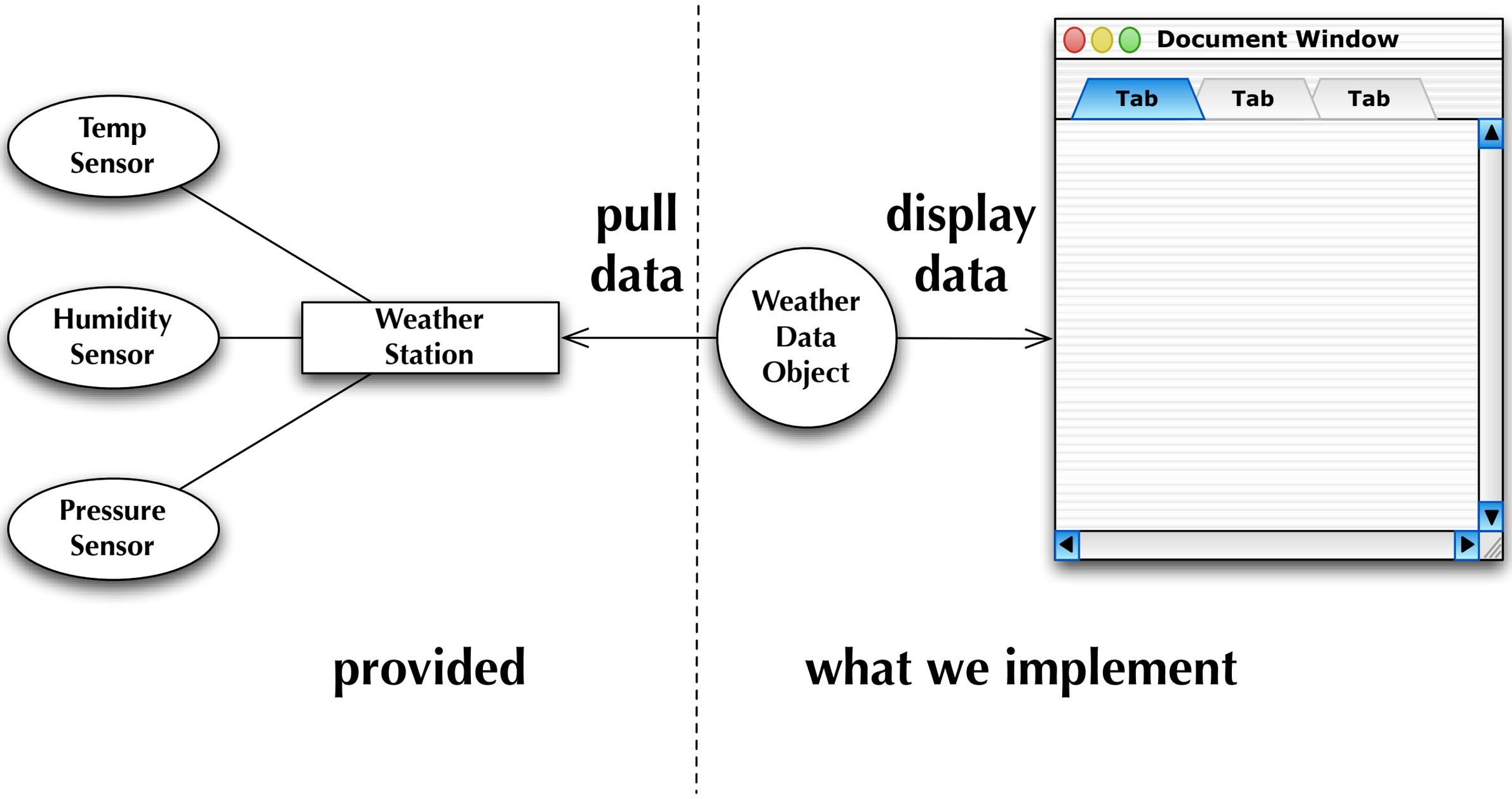
Lecture Goals

- Cover Material from Chapters 2 and 3 of the Design Patterns Textbook
  - Observer Pattern
  - Decorator Pattern

Observer Pattern

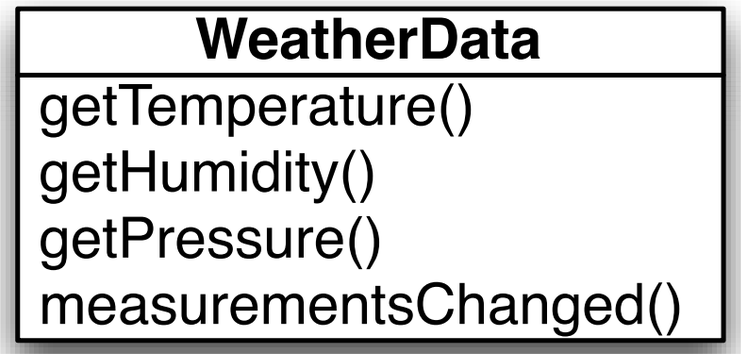
- Don’t miss out when something interesting (in your system) happens!
    - The observer pattern allows objects to keep other objects informed about events occurring within a software system (or across multiple systems)
    - Its dynamic in that an object can choose to receive notifications or not at run-time
    - Observer happens to be one of the most heavily used patterns in the Java Development Kit

Chapter Example: Weather Monitoring



We need to pull information from the station and then generate "current conditions, weather stats, and a weather forecast".

WeatherData Skeleton



We receive a partial implementation of the WeatherData class from our client.

They provide three getter methods for the sensor values and an empty measurementsChanged() method that is guaranteed to be called whenever a sensor provides a new value

We need to pass these values to our three displays… so that's simple!

First pass at measurementsChanged

```
1  ...
2
3  public void measurementsChanged() {
4
5      float temp     = getTemperature();
6      float humidity = getHumidity();
7      float pressure = getPressure();
8
9      currentConditionsDisplay.update(temp, humidity, pressure);
10     statisticsDisplay.update(temp, humidity, pressure);
11     forecastDisplay.update(temp, humidity, pressure);
12
13 }
14
15 ...
16
```

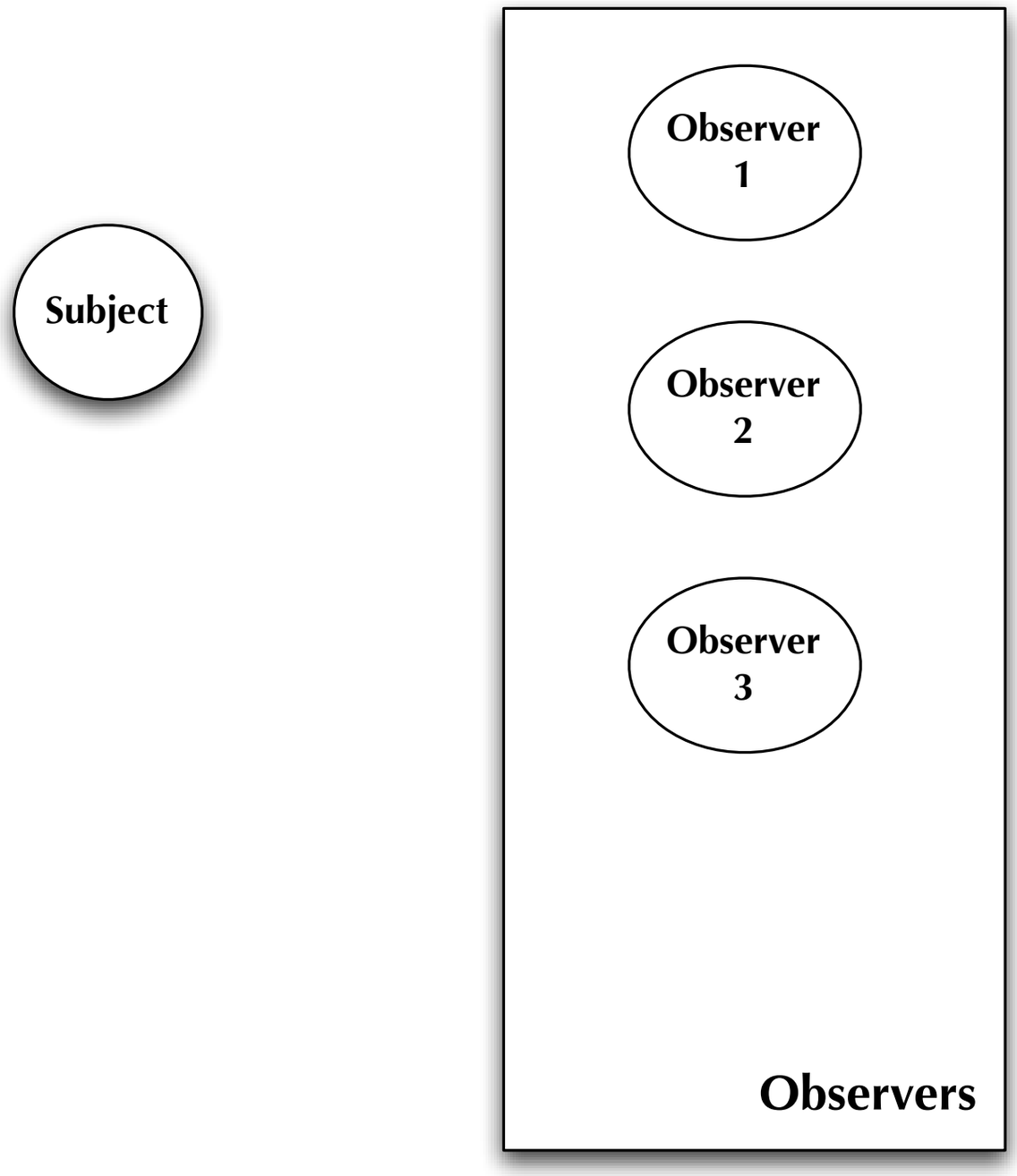
Problems?

1. **The number and type of displays may vary.**
These three displays are hard coded with no easy way to update them.
2. **Coding to implementations, not an interface!**
Although each implementation has adopted the same interface, so this will make translation easy!

Observer Pattern

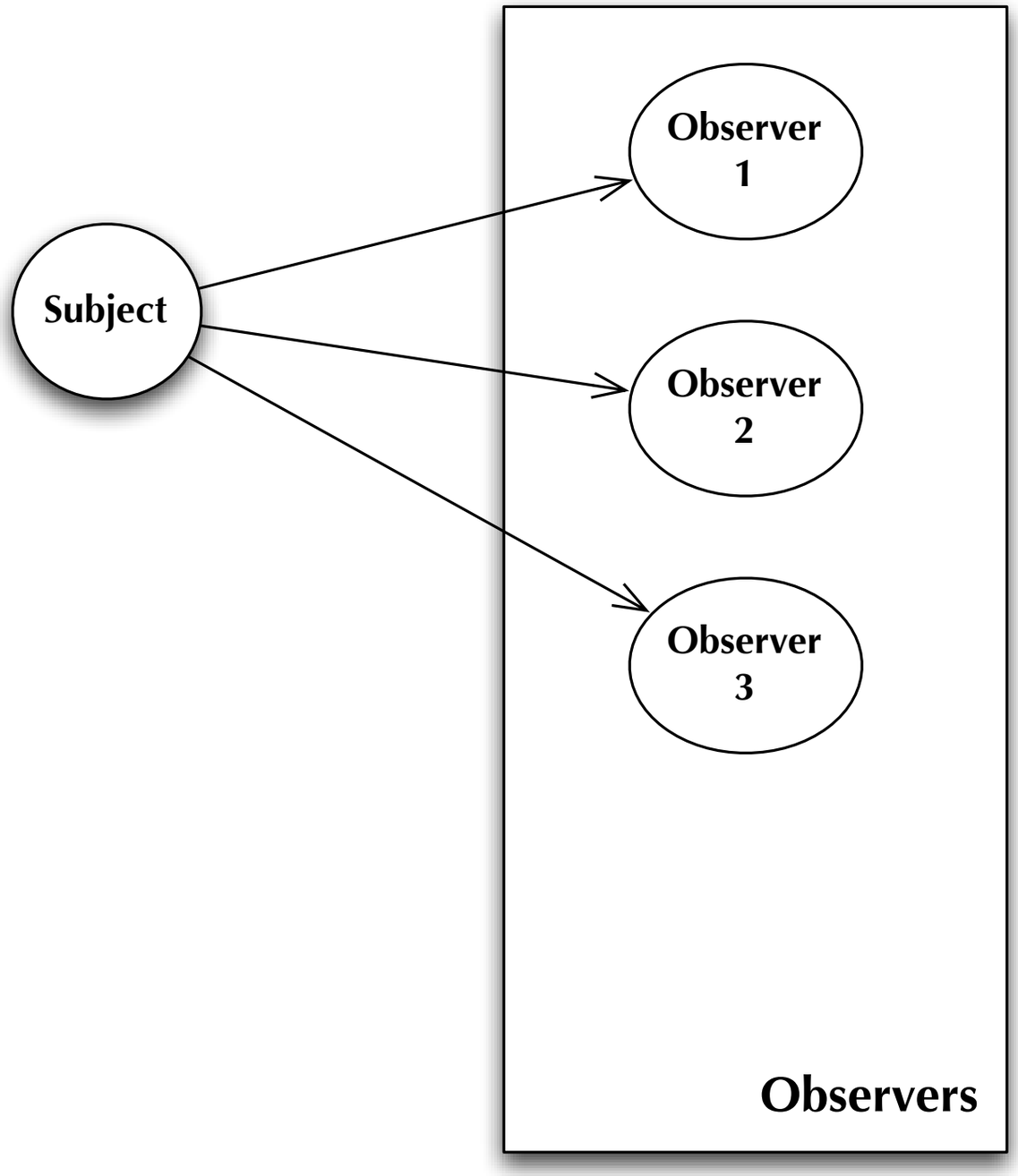
- This situation can benefit from use of the observer pattern
  - This pattern is similar to subscribing to a hard copy newspaper
    - A newspaper comes into existence and starts publishing editions
    - You become interested in the newspaper and subscribe to it
    - Any time an edition becomes available, you are notified (by the fact that it is delivered to you)
    - When you don't want the paper anymore, you unsubscribe
    - The newspaper's current set of subscribers can change at any time
  - Observer is just like this but we call the publisher the "subject" and we refer to subscribers as "observers"

Observer in Action (I)



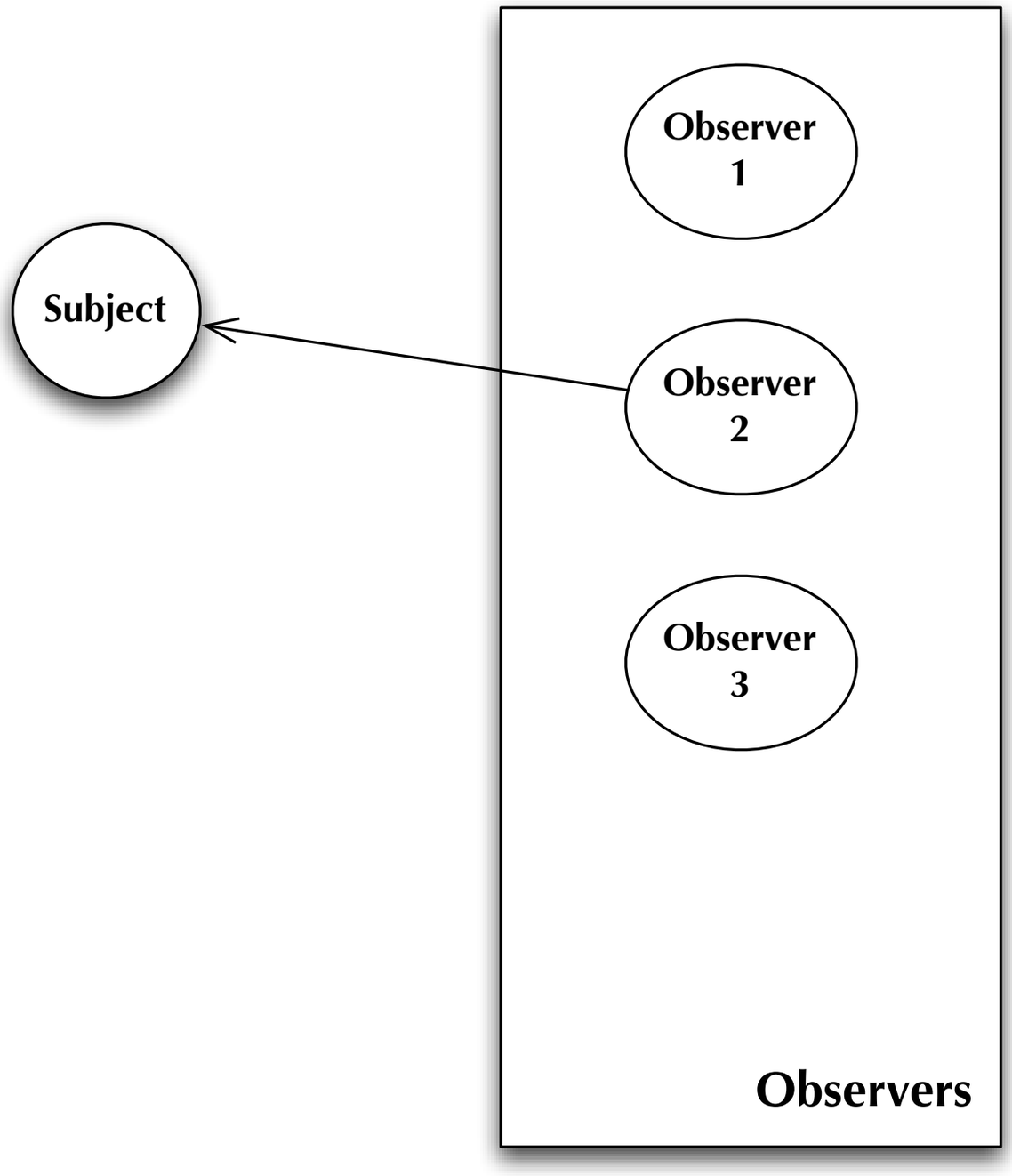
Subject maintains a list of observers

Observer in Action (II)



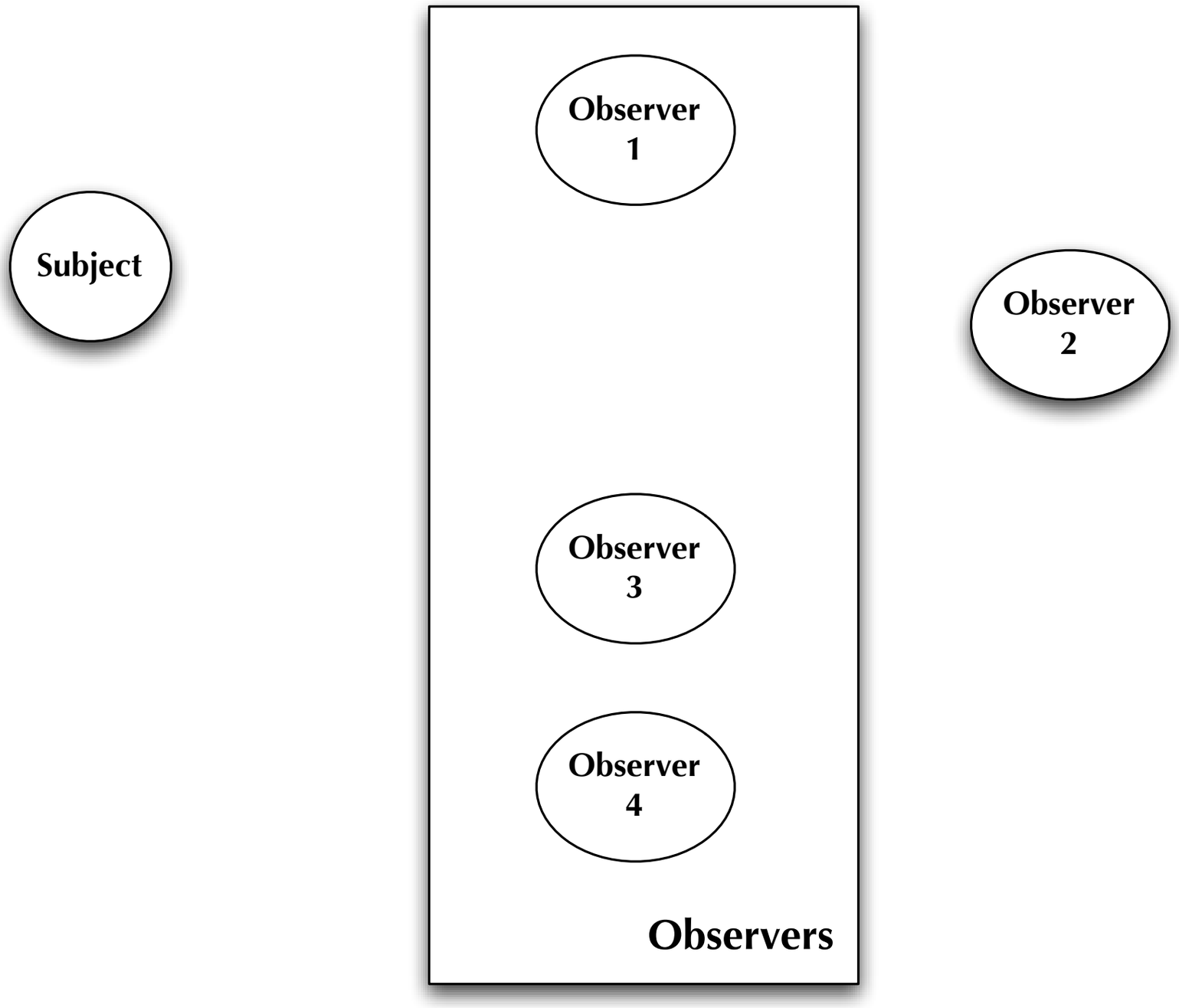
If the Subject changes, it notifies its observers

Observer in Action (III)



If needed, an observer may query its subject for more information

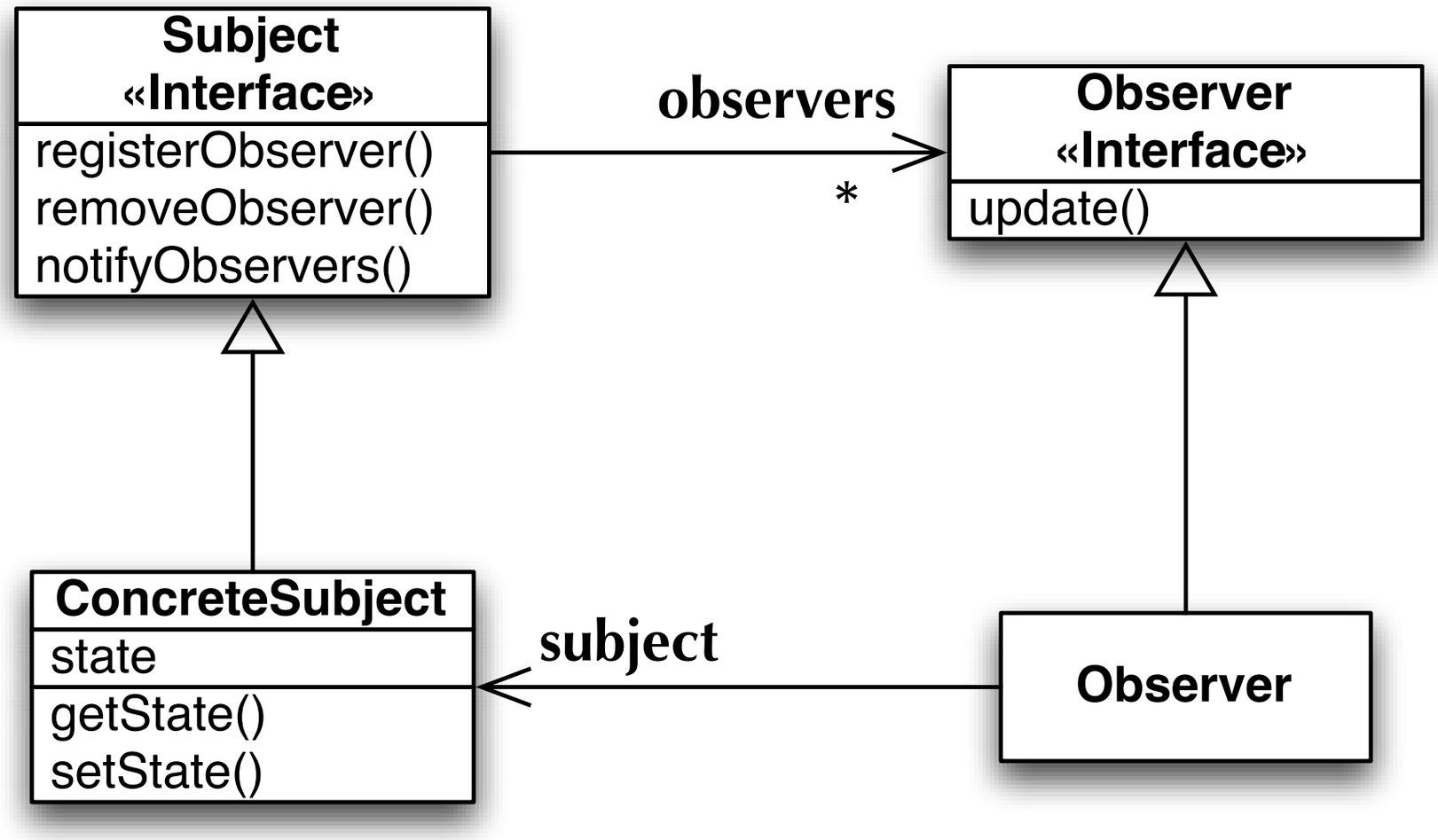
Observer In Action (IV)



At any point, an observer may join or leave the set of observers

Observer Definition and Structure

- The Observer Pattern defines a one-to-many dependency between a set of objects, such that when one object (the subject) changes all of its dependents (observers) are notified and updated automatically



Observer Benefits

- Observer affords a loosely coupled interaction between subject and observer
  - This means they can interact with very little knowledge about each other
- Consider
  - The subject only knows that observers implement the Observer interface
    - We can add/remove observers of any type at any time
    - We never have to modify subject to add a new type of observer
  - We can reuse subjects and observers in other contexts
    - The interfaces plug-and-play where ever observer is used
  - Observers may have to know about the ConcreteSubject class if it provides many different state-related methods
    - Otherwise, data can be passed to observers via the update() method

Demonstration

- Roll Your Own Observer
- Using java.util.Observable and java.util.Observable
    - Observable is a CLASS, a subject has to subclass it to manage observers
    - Observer is an interface with one defined method: update(subject, data)
    - To notify observers: call setChanged(), then notifyObservers(data)
- Observer in Swing
    - Listener framework is just another name for the Observer pattern

Decorator Pattern

- The Decorator Pattern provides a powerful mechanism for adding new behaviors to an object at run-time
    - The mechanism is based on the notion of “wrapping” which is just a fancy way of saying “delegation” but with the added twist that the delegator and the delegate both implement the same interface
        - You start with object A that implements abstract type X
        - You then create object B that also implements abstract type X
        - You pass A into B’s constructor and then pass B to A’s client
        - The client thinks its talking to A but its actually talking to B
        - B’s methods augment A’s methods to provide new behavior

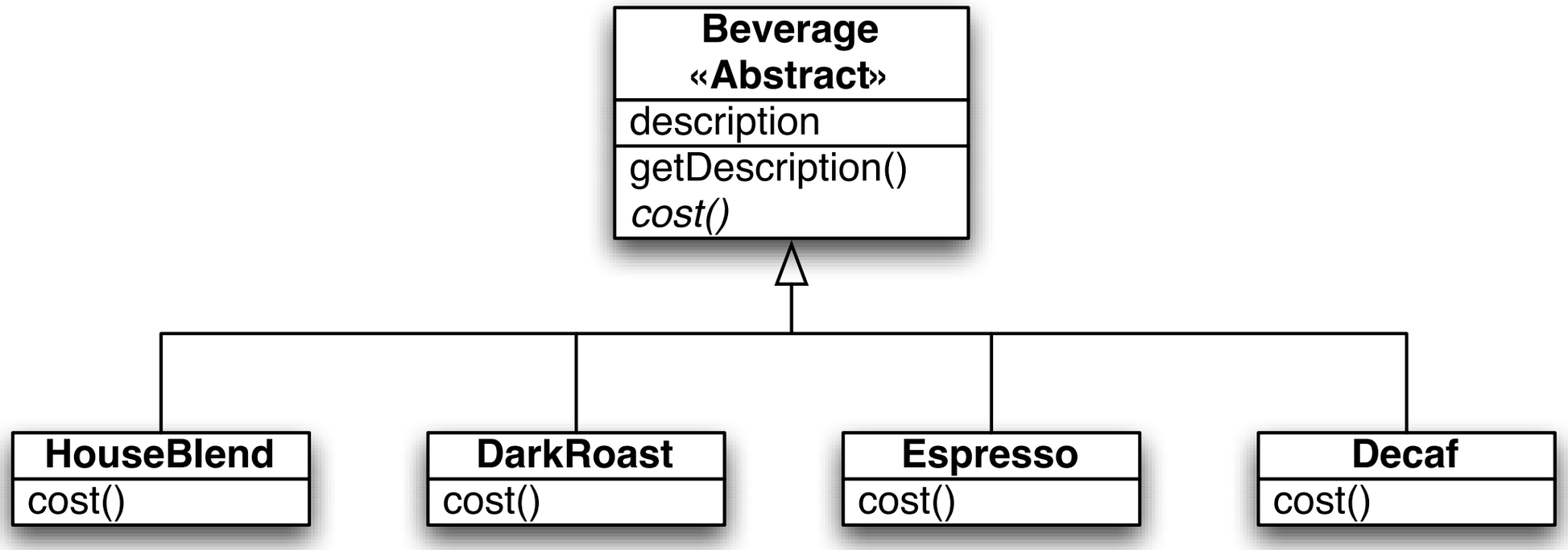
Why? Open-Closed Principle

- The decorator pattern provides yet another way in which a class's runtime behavior can be extended without requiring modification to the class
- This supports the goal of the open-closed principle:
  - Classes should be open for extension but closed to modification
    - Inheritance is one way to do this, but composition and delegation are more flexible (and Decorator takes advantage of delegation)
- Chapter 3's "Starbuzz Coffee" example clearly demonstrates why inheritance can get you into trouble and why delegation/composition provides greater run-time flexibility

Starbuzz Coffee

- Under pressure to update their “point of sale” system to keep up with their expanding set of beverage products
  - Started with a Beverage abstract base class and four implementations: HouseBlend, DarkRoast, Decaf, and Espresso
    - Each beverage can provide a description and compute its cost
- But they also offer a range of condiments including: steamed milk, soy, and mocha
  - These condiments **alter** a beverage’s description and cost
    - “Alter” is a key word here since it provides a hint that we might be able to use the Decorator pattern

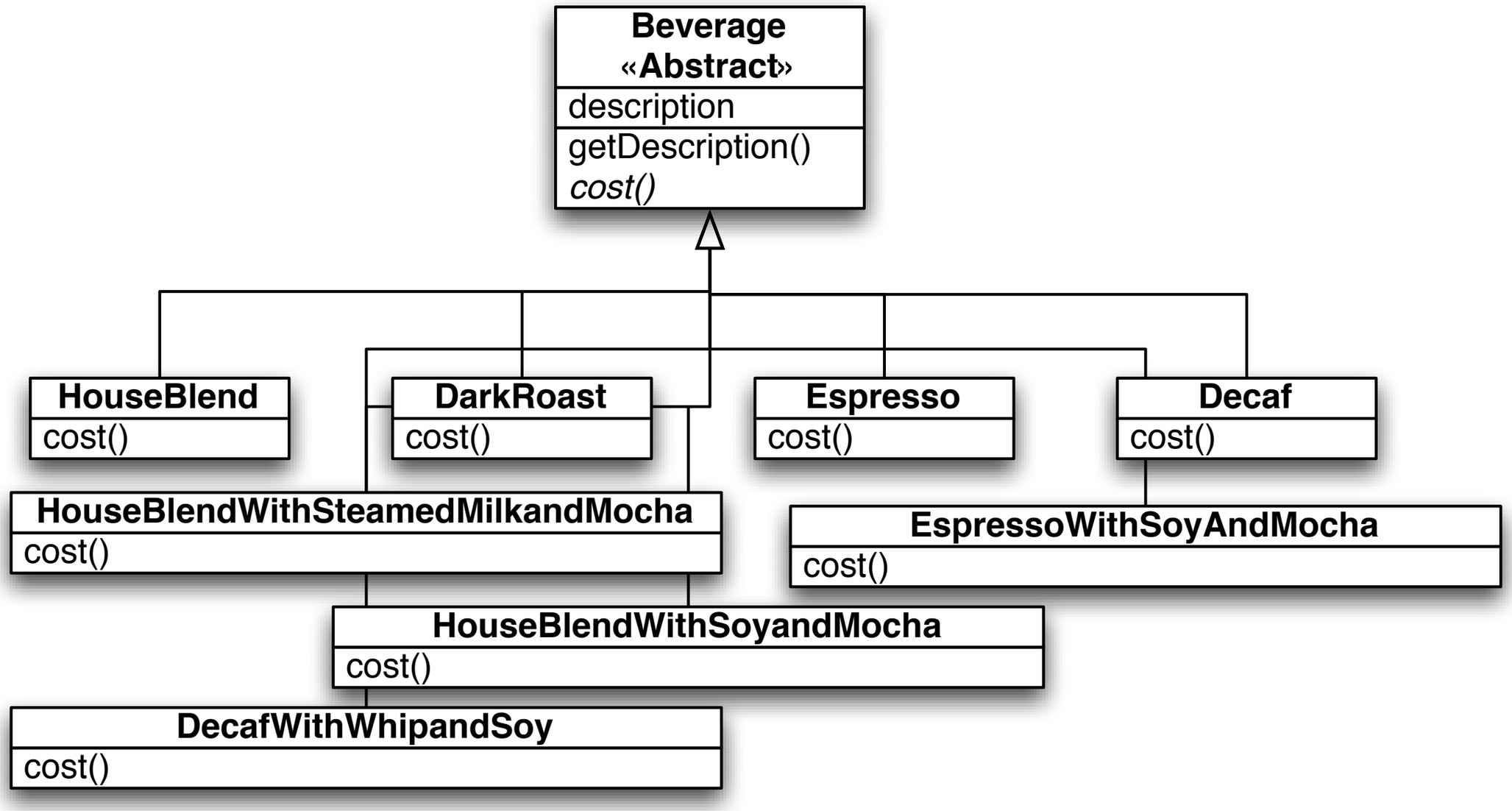
Initial Starbuzz System



With inheritance on your brain, you may add condiments to this design in one of two ways

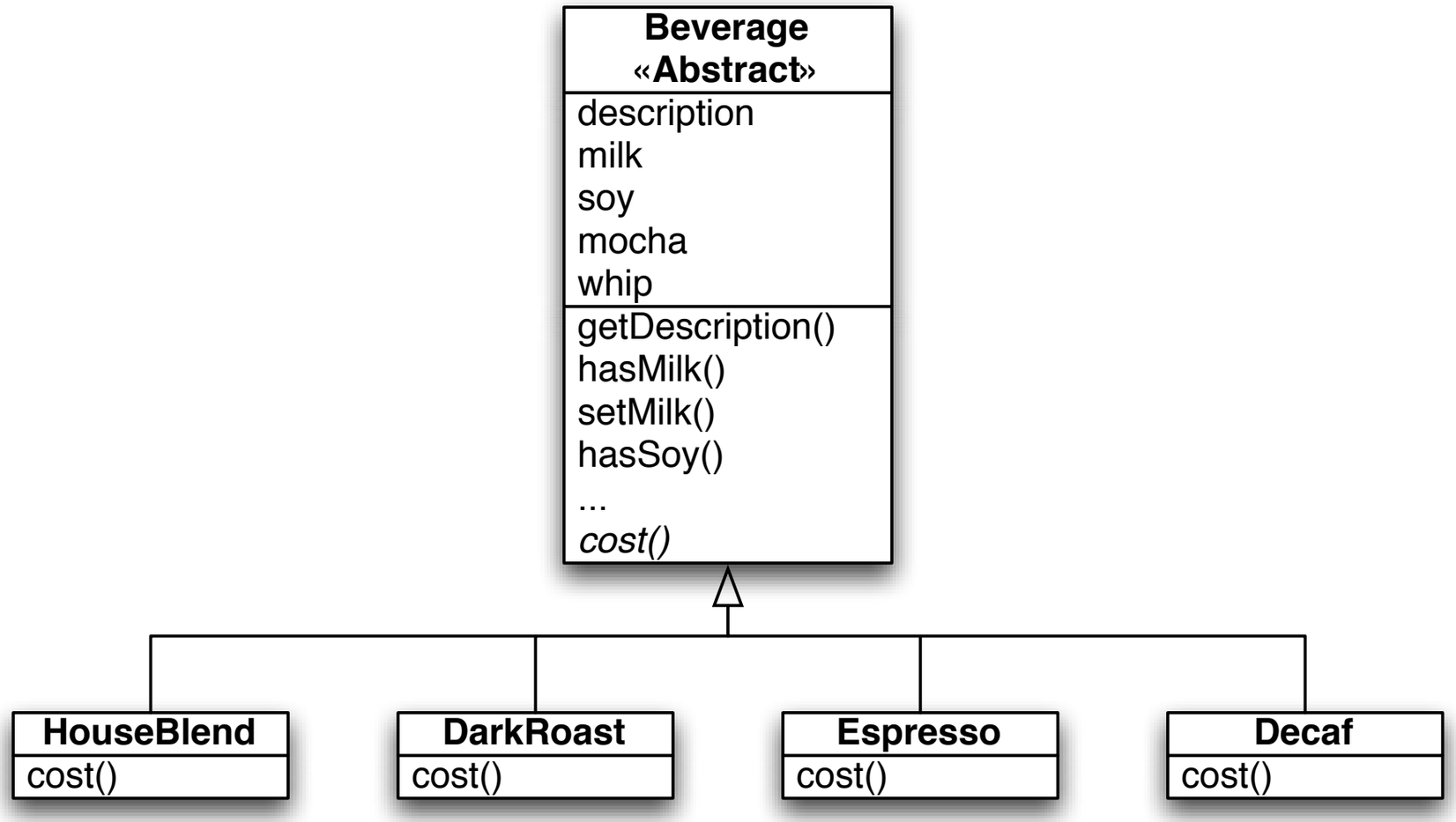
1. One subclass per combination of condiment (wont work in general but especially not in Boulder!)
2. Add condiment handling to the Beverage superclass

One Subclass per Combination



This is incomplete, but you can see the problem…
(see page 81 for a more complete picture)

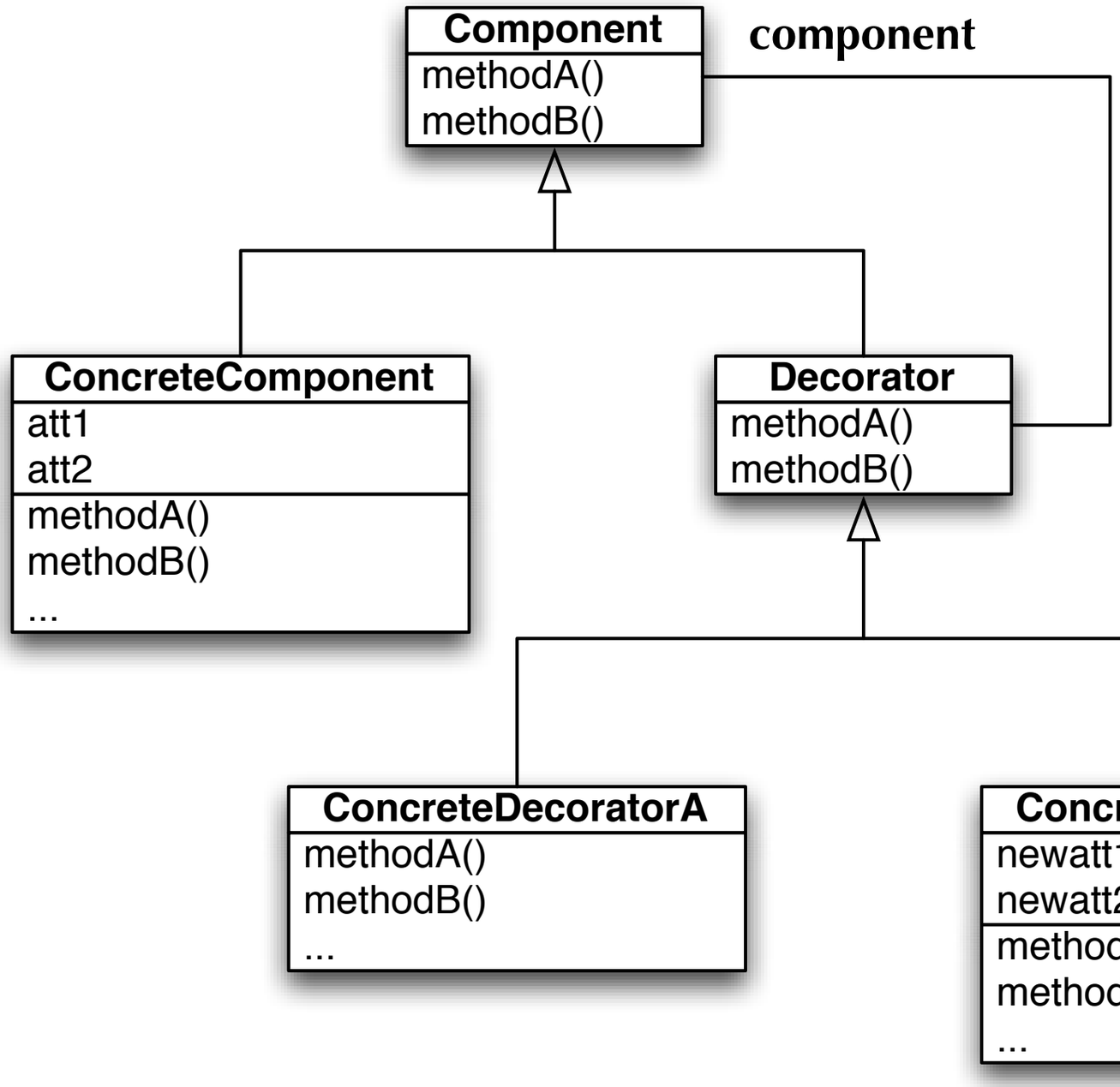
Let Beverage Handle Condiments



Houston, we have a problem…

1. This assumes that all concrete Beverage classes need these condiments
2. Condiments may vary (old ones go, new ones are added, price changes, etc.), shouldn't they be encapsulated some how?
3. How do you handle "double soy" drinks with boolean variables?

Decorator Pattern: Definition and Structure

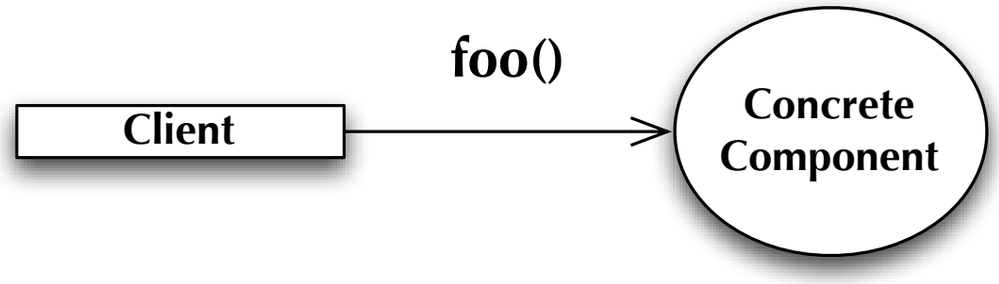


Inheritance is used to make sure that components and decorators **share** the same interface: namely the **public interface of Component** which is either an **abstract class** or an **interface**

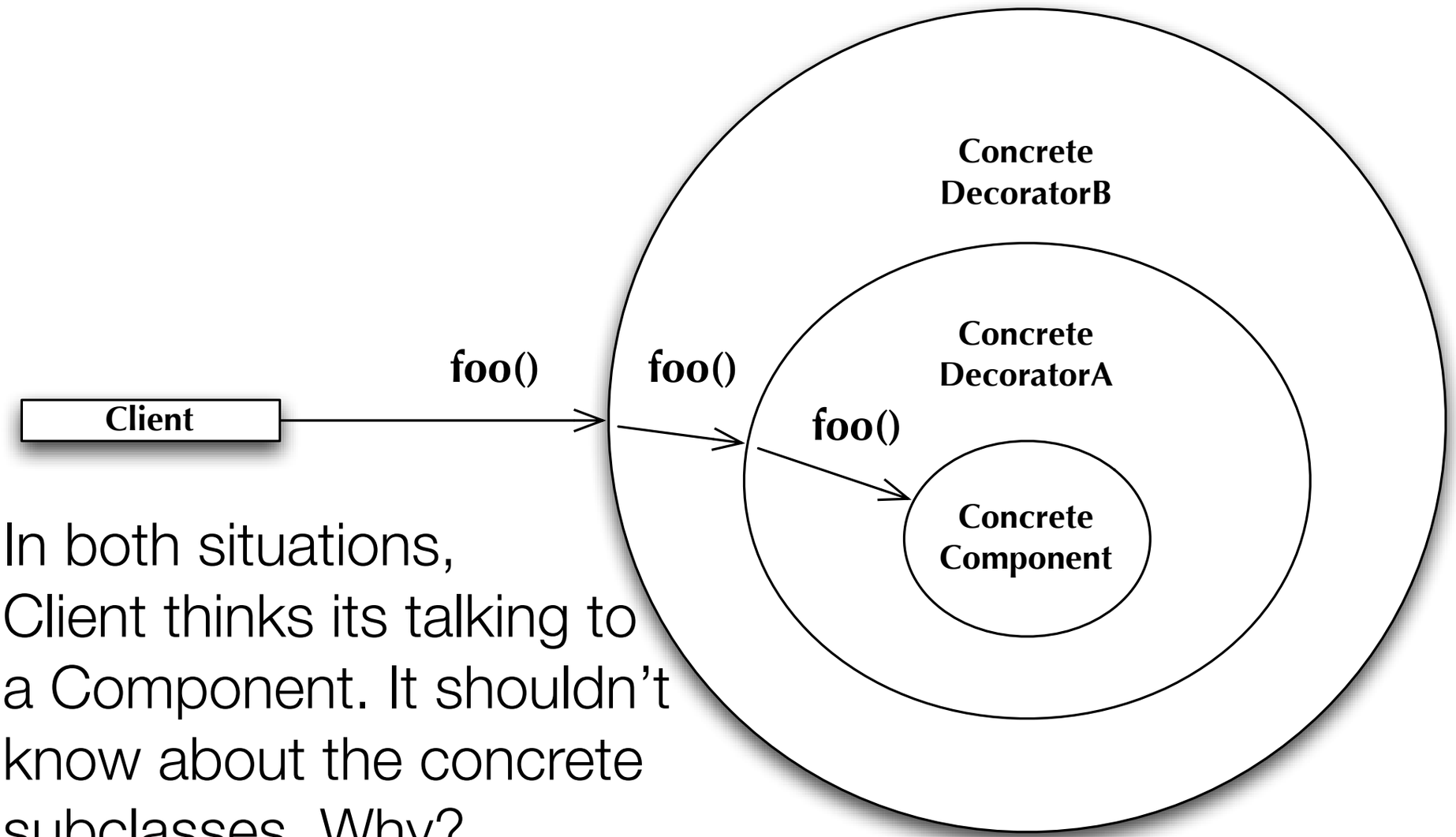
At run-time, concrete decorators **wrap** concrete components and/or other concrete decorators

The object to be wrapped is typically passed in **via the constructor**

Client Perspective



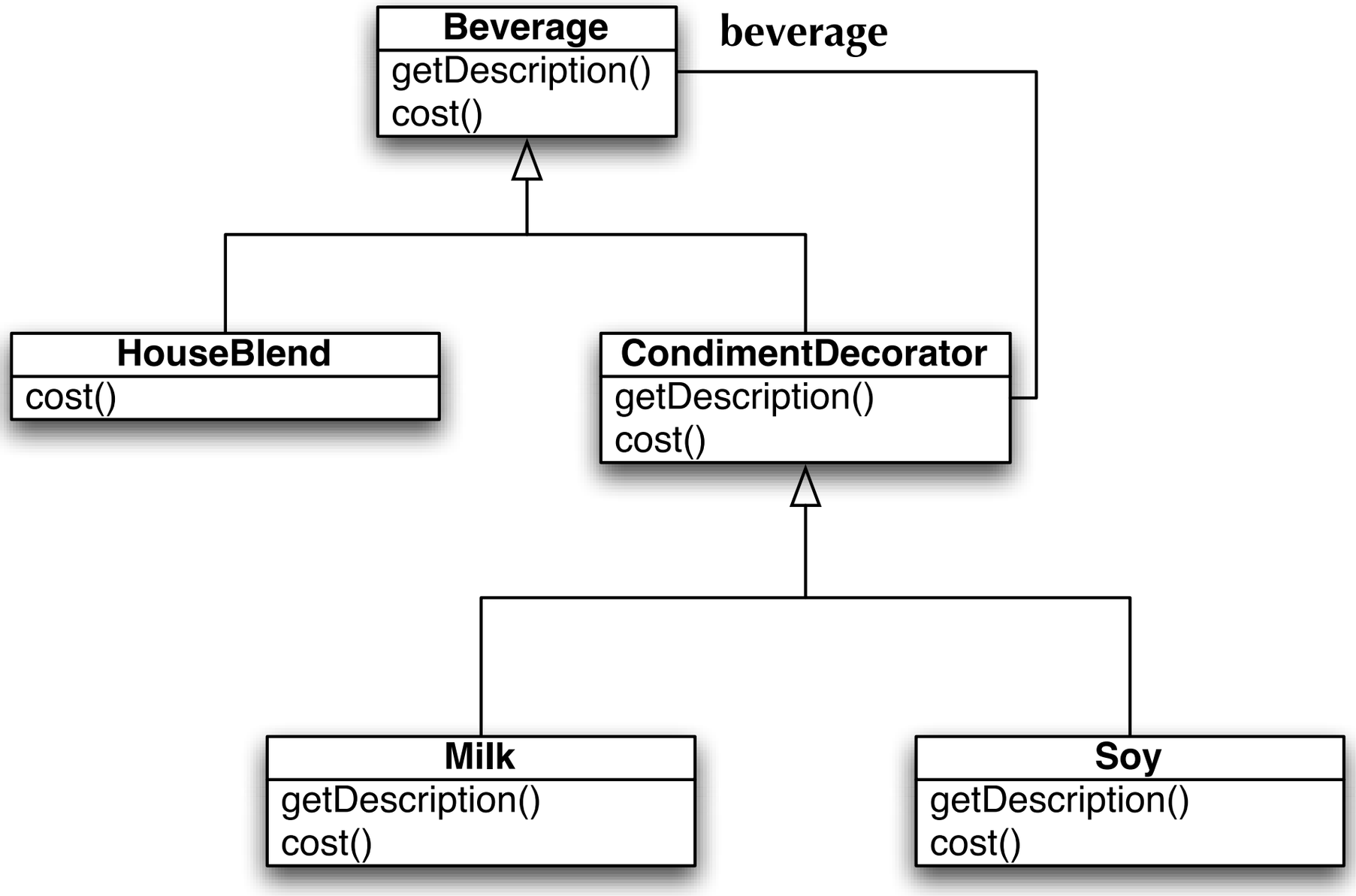
BEFORE



AFTER

In both situations, Client thinks its talking to a Component. It shouldn't know about the concrete subclasses. Why?

StarBuzz Using Decorators (Incomplete)



Demonstration

- Starbuzz Example
- Use of Decorator Pattern in java.io package
    - InputStream == Component
    - FilterInputStream == Decorator
        - FileInputStream, StringBufferInputStream, etc. == ConcreteComponent
        - BufferedInputStream, LineNumberInputStream, etc. == ConcreteDecorator

Wrapping Up

- Observer
  - Loosely coupled state change notifications
  - between a subject and a dynamically changing set of observers
- Decorator
  - Way to implement open-closed principle that
    - makes use of inheritance to share an interface between a set of components and a set of decorators
    - makes use of composition and delegation to dynamically wrap decorators around components at run-time

Coming Up Next

- Lecture 21: Factory Pattern
    - Read Chapter 4 of the Design Patterns Textbook
- Lecture 22: Singleton and Command Patterns
    - Read Chapters 5 and 6 of the Design Patterns Textbook