Patterns 3

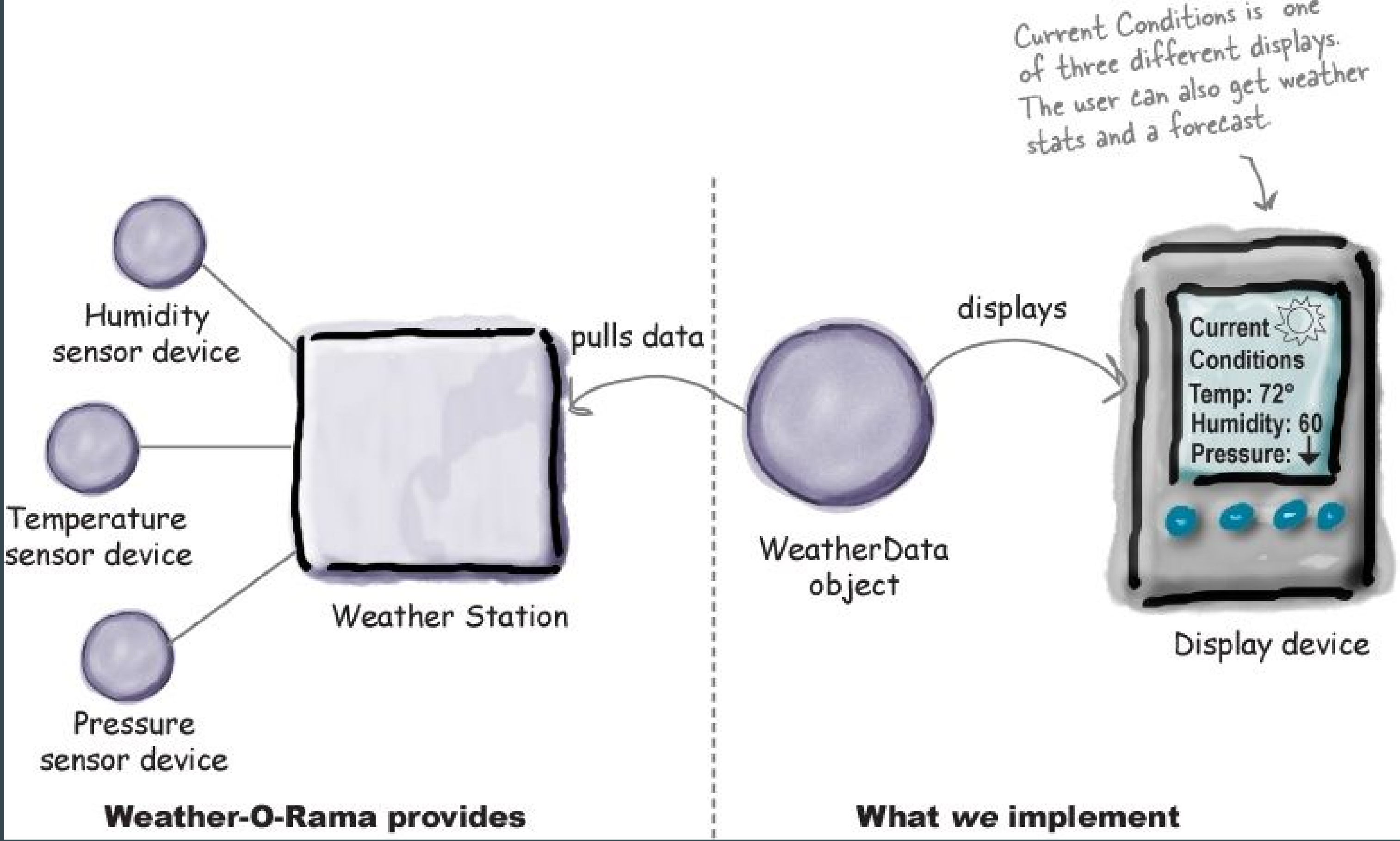
The Observer Pattern, Singleton Pattern, Decorator Pattern

Obeserver Pattern: The Weather Station Application

Congratulations on being selected to build our next-generation, Internet-based Weather Monitoring Station! The weather station will be based on our patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like you to create an application that initially provides three display elements: current conditions, weather statistics, and a simple forecast, all updated in real time as the WeatherData object acquires the most recent measurements. Further, this is an expandable weather station. Weather-ORama wants to release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API! Weather-O-Rama thinks we have a great business model: once the customers are hooked, we intend to charge them for each display they use. Now for the best part: we are going to pay you in stock options. We look forward to seeing your design and alpha application.

Sincerely,Johnny Hurricane, CEOP.S.

Application Overview

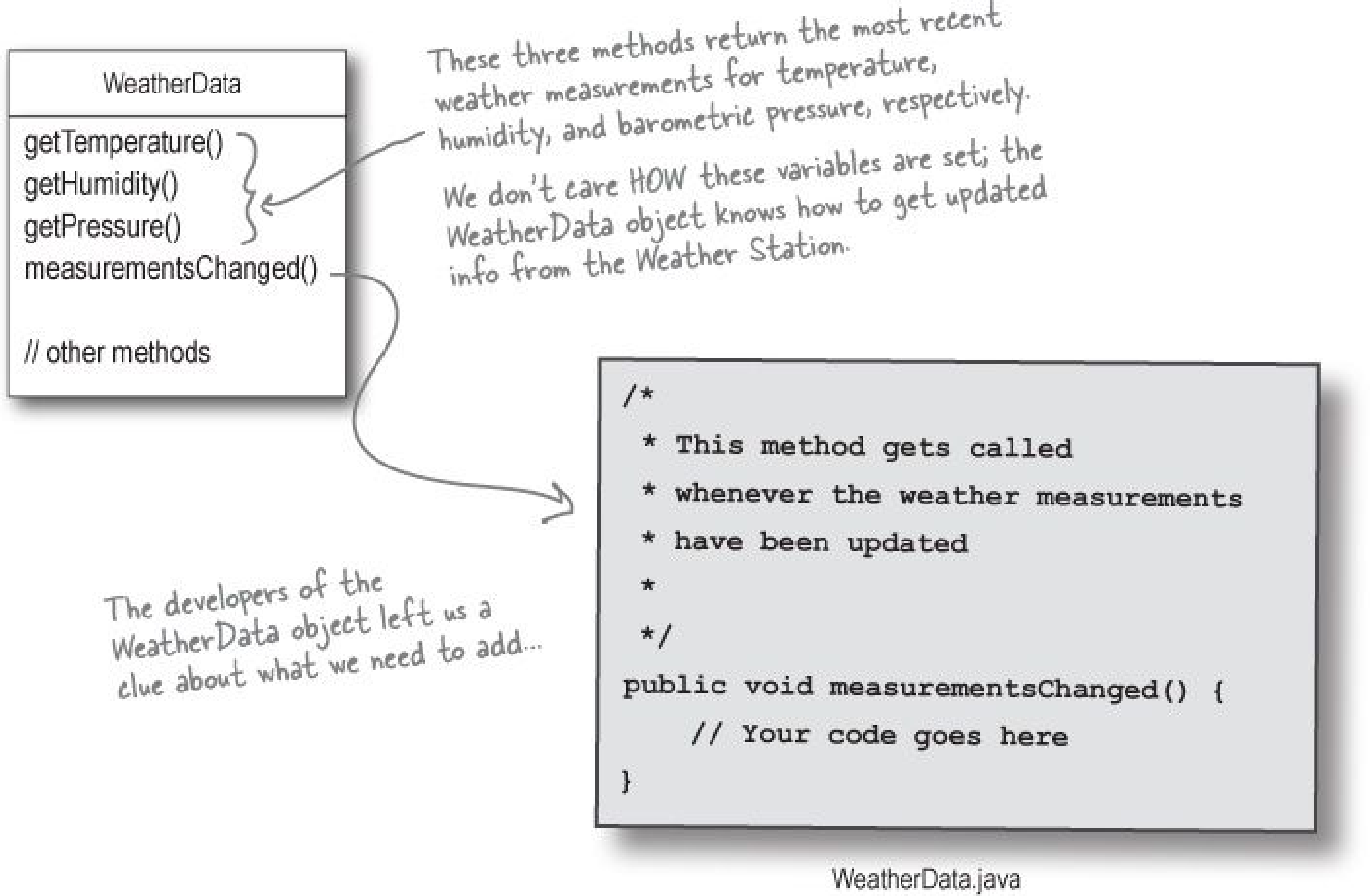


The WeatherData Object

The WeatherData object knows how to get data:

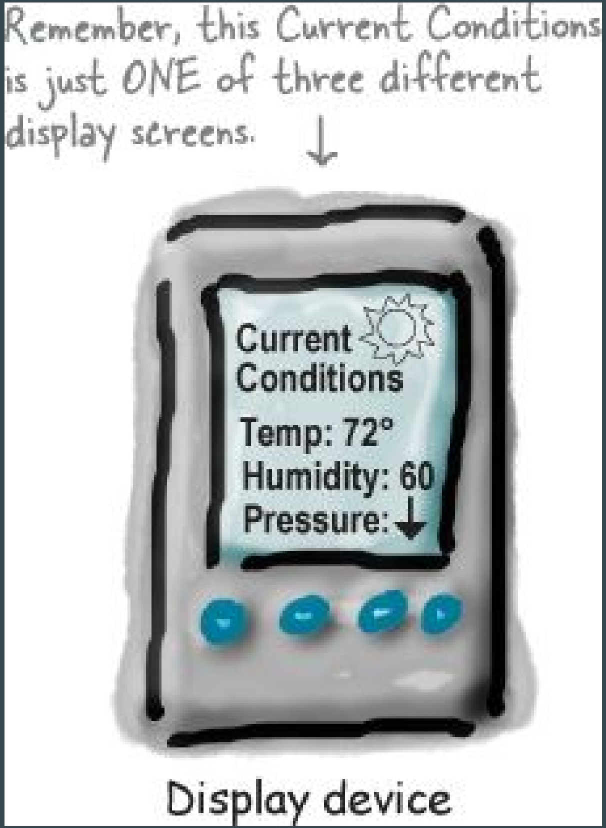
```
getTemperature()
getHumidity()
getPressure()
```

measurementsChanged() is called with every new measurement



The displays

- There are 3 display elements (screens) per display
    - Current conditions
    - Statistics
    - Forecast
- These must be updated every time the station gets new data
- We must be able to add new displays at runtime



A bad solution

This way violates the principles we have discussed

```java
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Area of change. We need to encapsulate this.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method that takes the temp, humidity, and pressure values.

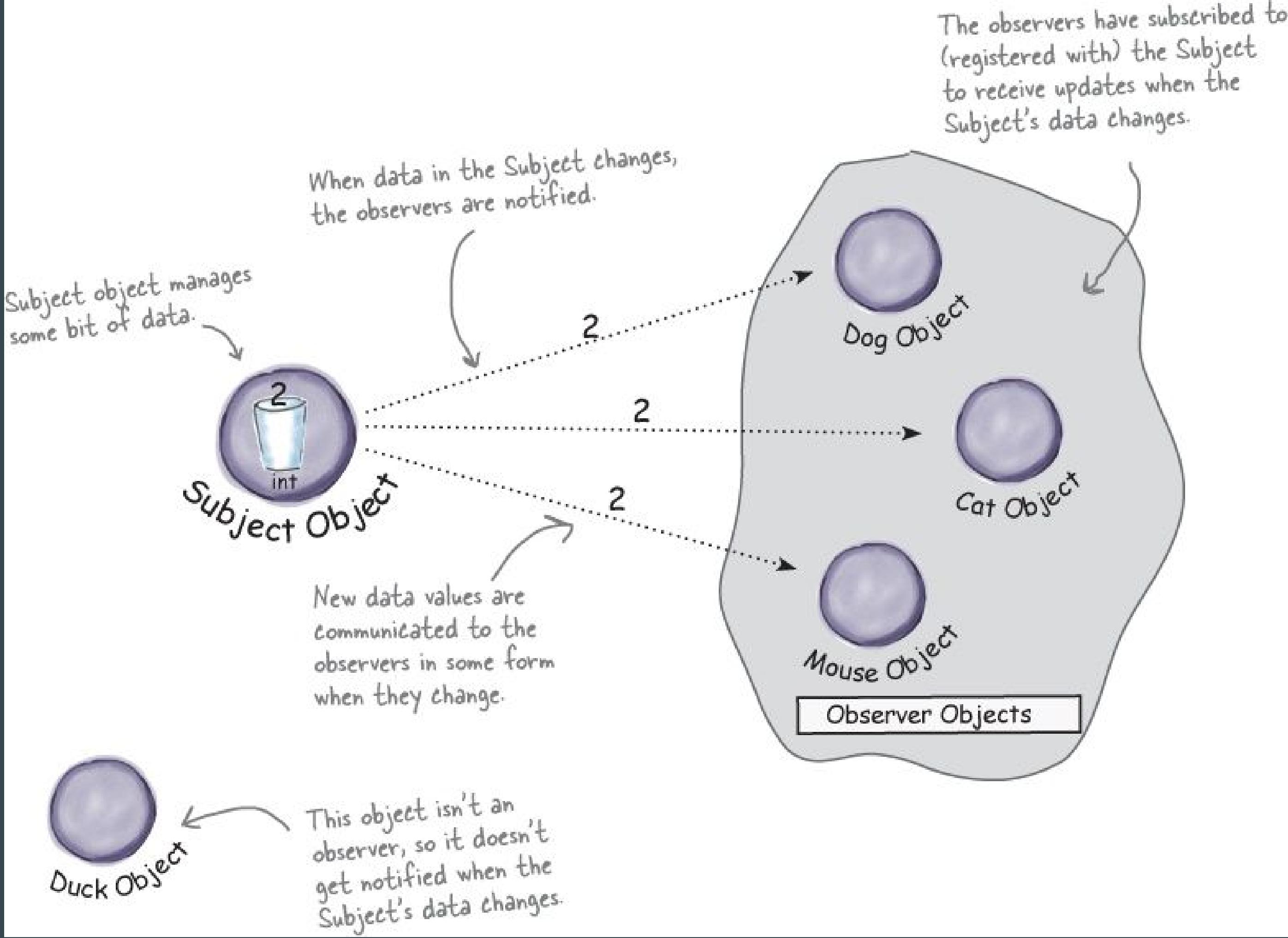
A model: subscribing to a newspaper

- A newspaper publisher goes into business and begins publishing newspapers.
- You subscribe to a particular publisher, and every time there’s a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
- You unsubscribe when you don’t want papers anymore, and they stop being delivered.
- While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.

The pattern schematic

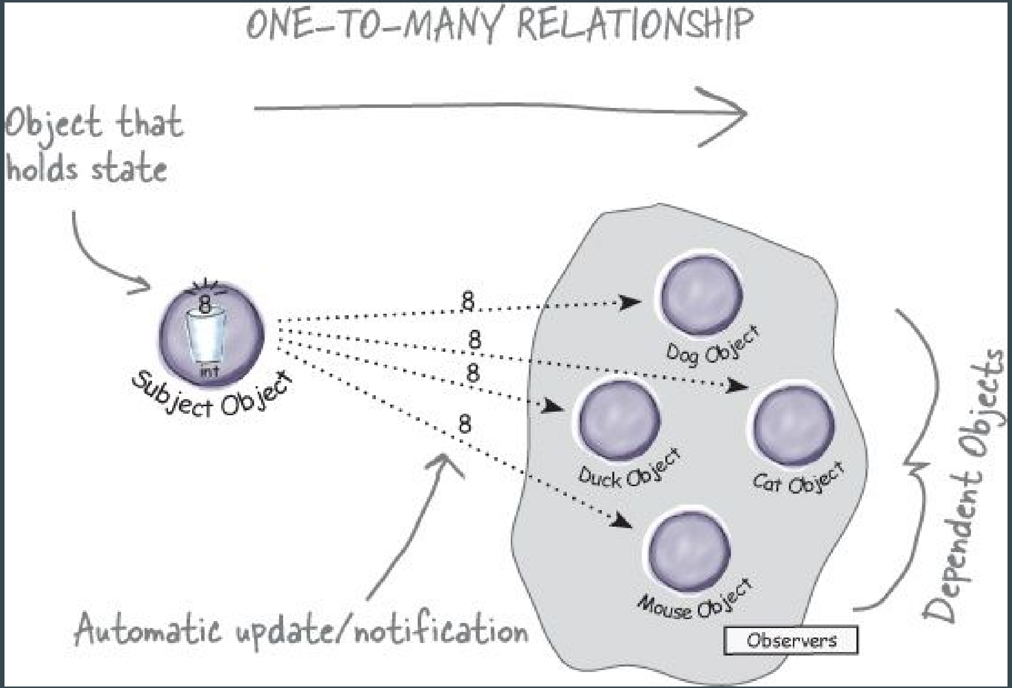
The Dog, Cat, and Mouse objects are subscribers, Duck isn't. Each can decide at any time to subscribe or unsubscribe

Publishers + subscribers = subscriber pattern

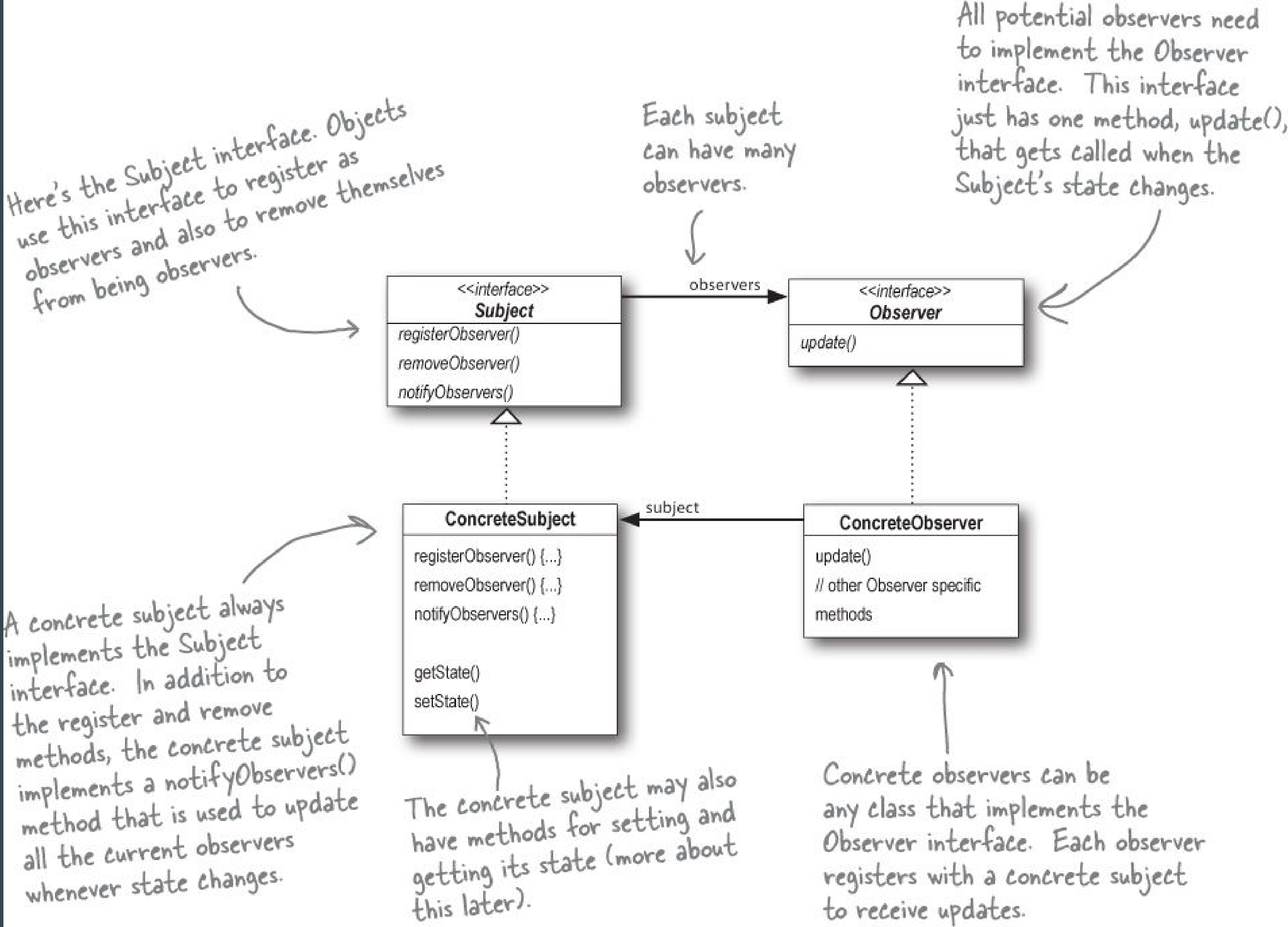


The observer pattern defined

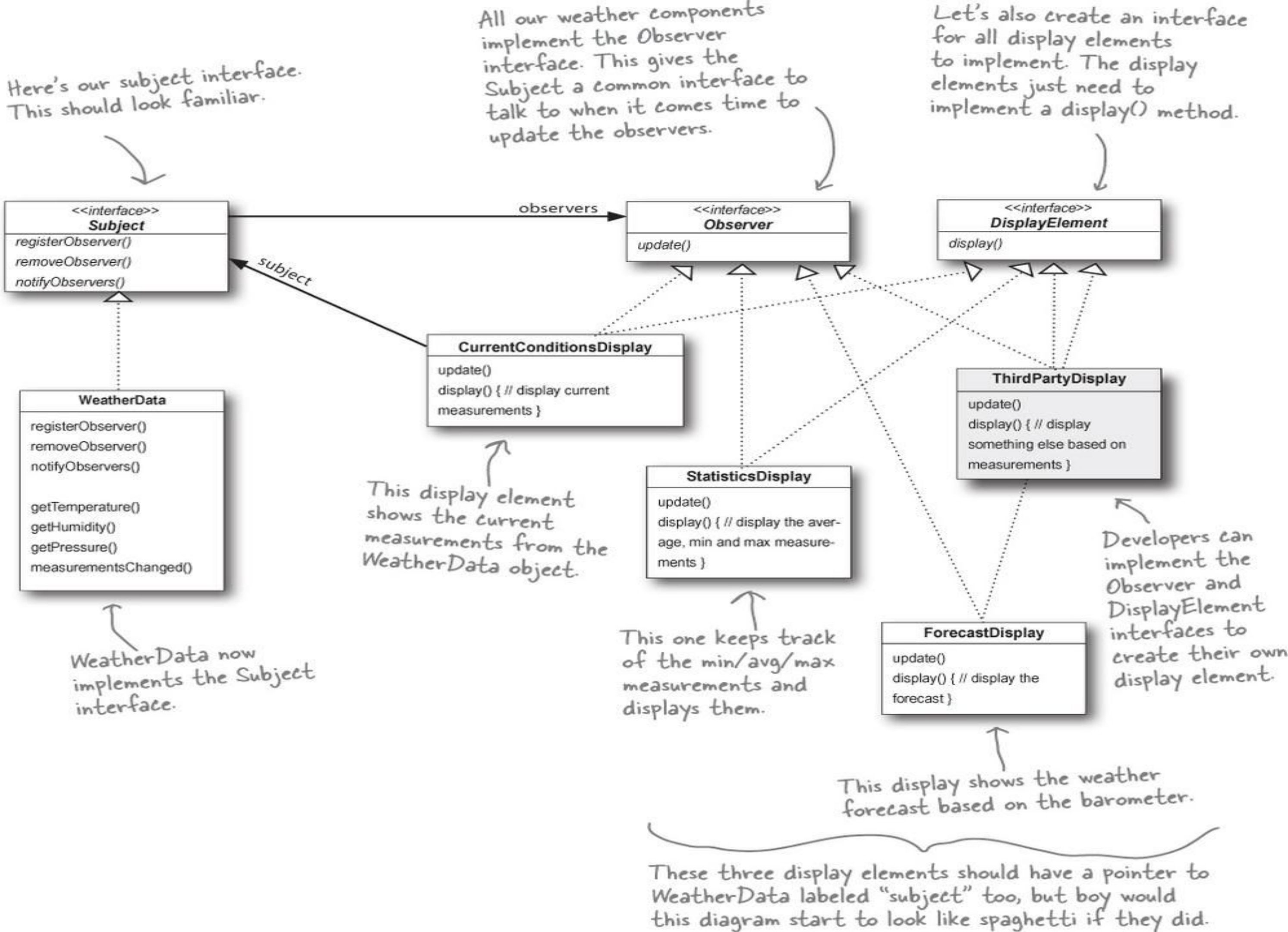
The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



Class diagram



UML for weather station



Code demo

See code

Singleton Pattern

- Classes with only one instance
- Typical singleton objects: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards.
- Having multiple instances of these can cause problems

Implement-ation

Let's rename MyClass to Singleton.

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

We have a static variable to hold our one instance of the class Singleton.

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

Code demo

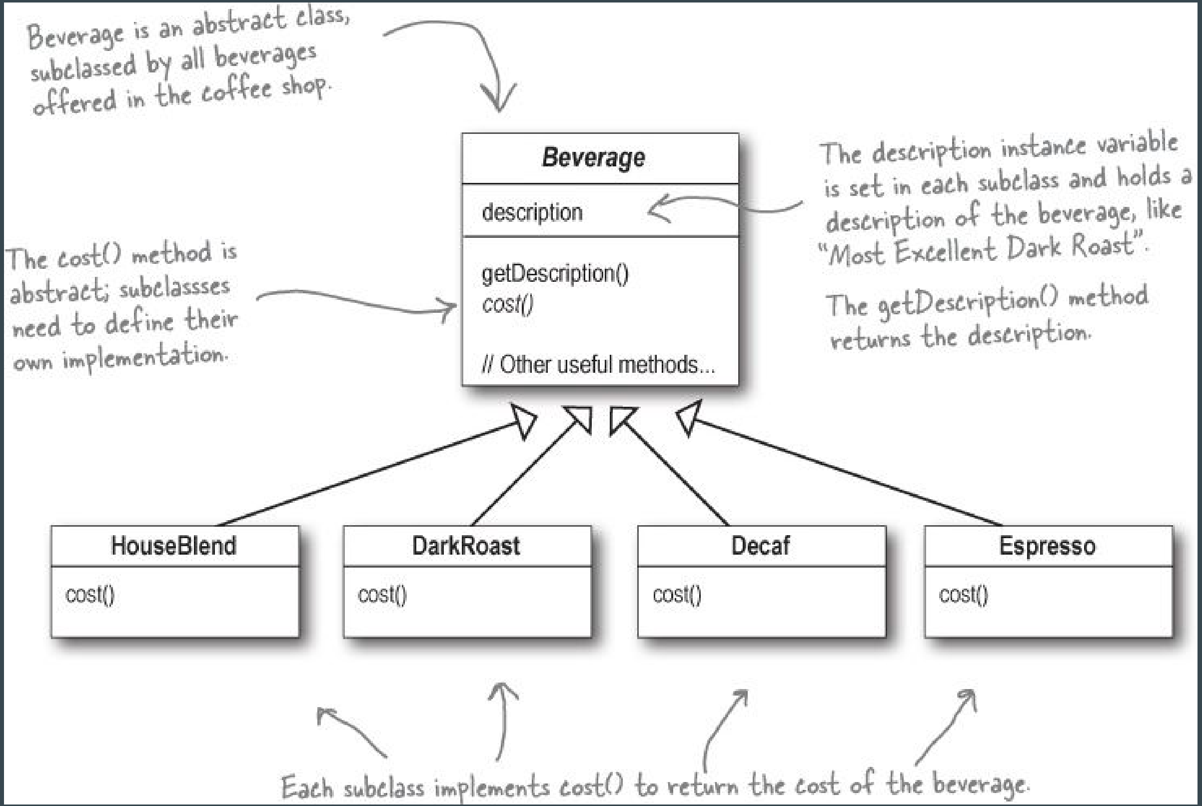
See code

Decorator pattern

Starbuzz coffee is a fast-growing coffee chain

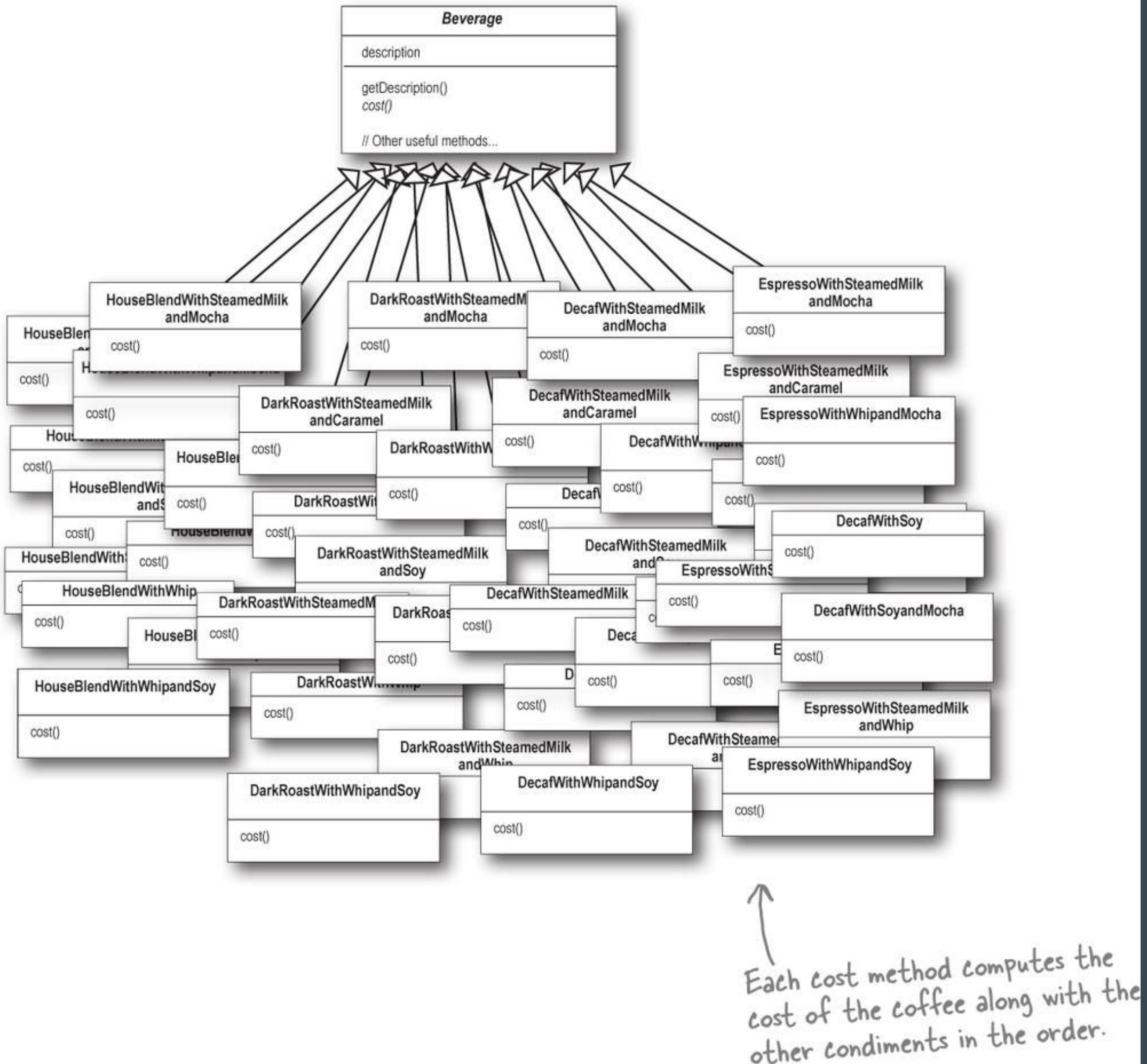
It started out simple enough, with classes like the UML diagram to the right

cost() is implemented by subclasses



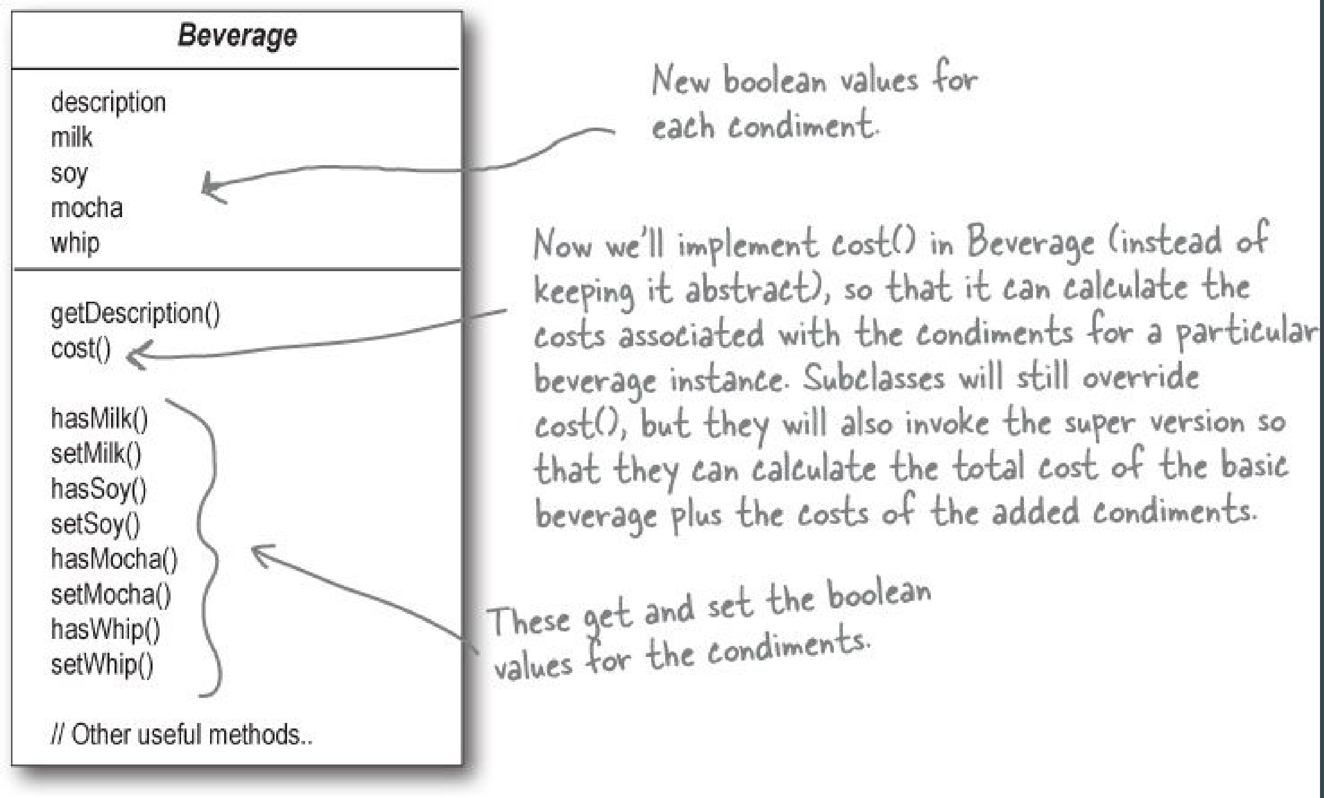
But

They ended up with too many classes and cost() methods to handle all the different condiment combinations



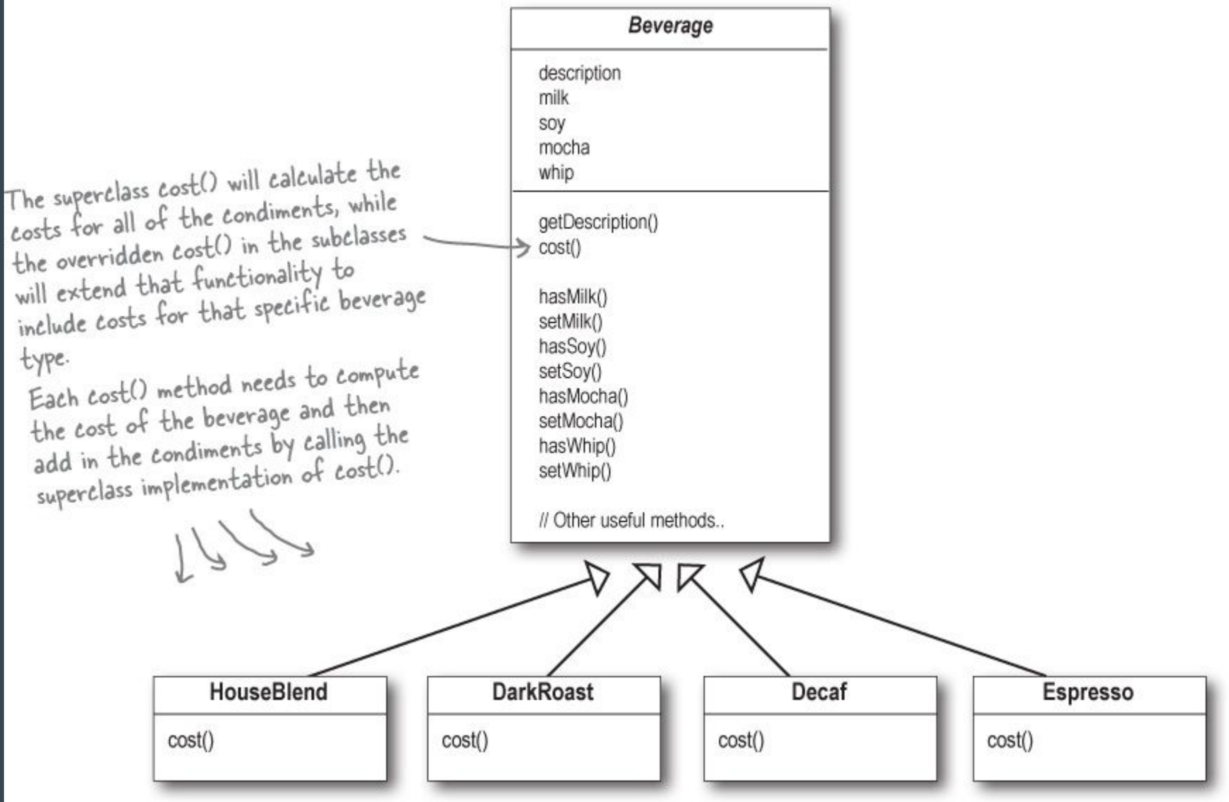
Well,

How about keeping track of condiments in a super class, implementing cost() there?



Then we have each type of coffee inherit from the superclass

But the functionality isn't appropriate for all the subclasses



Enter Decorator Pattern

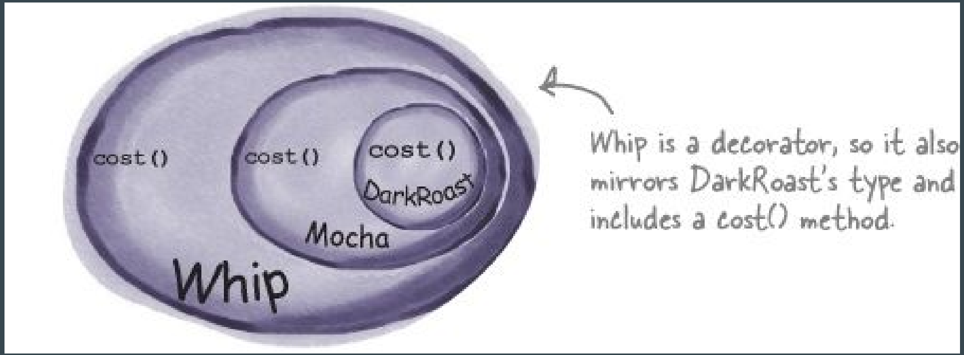
① Take a DarkRoast object

② Decorate it with a Mocha object

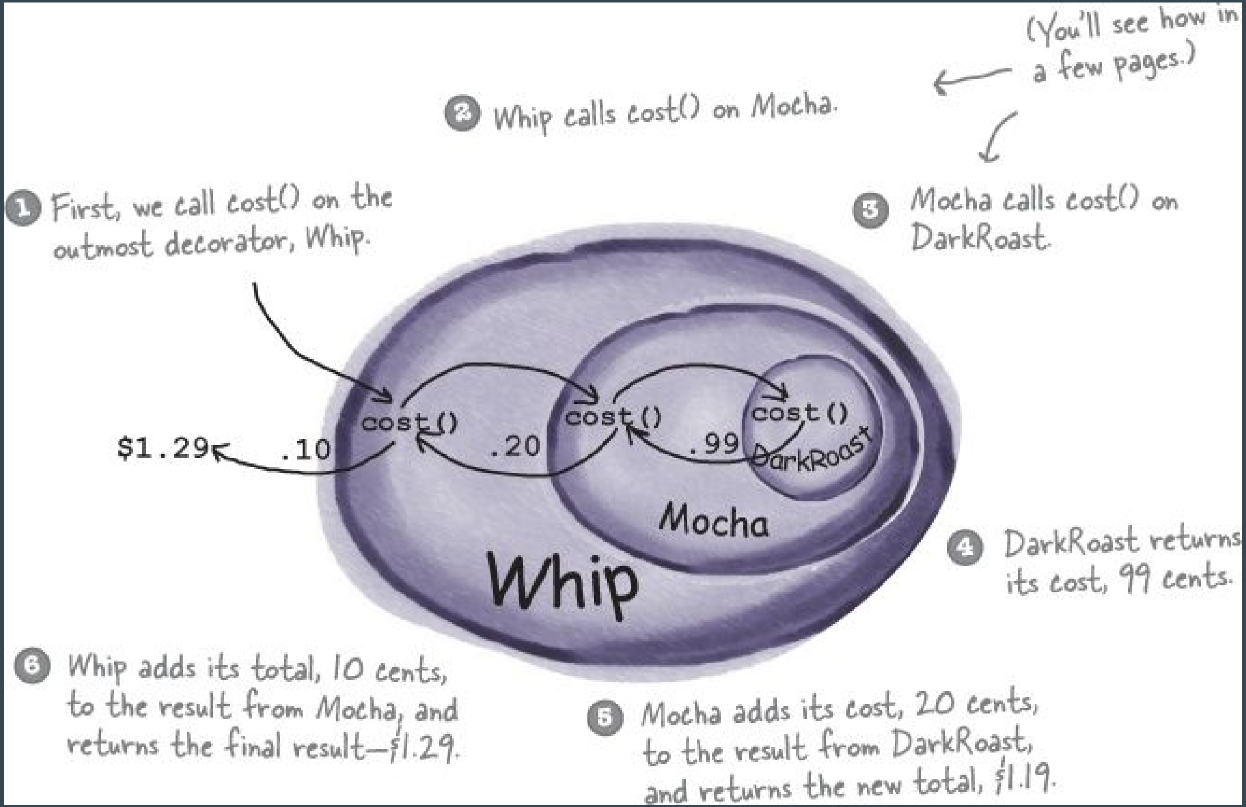
③ Decorate it with a Whip object

④ Call the cost() method and rely on delegation to add on the condiment costs

What's decorate mean: look to the right: we wrap DarkRoast with Mocha, and wrap Mocha with Whip



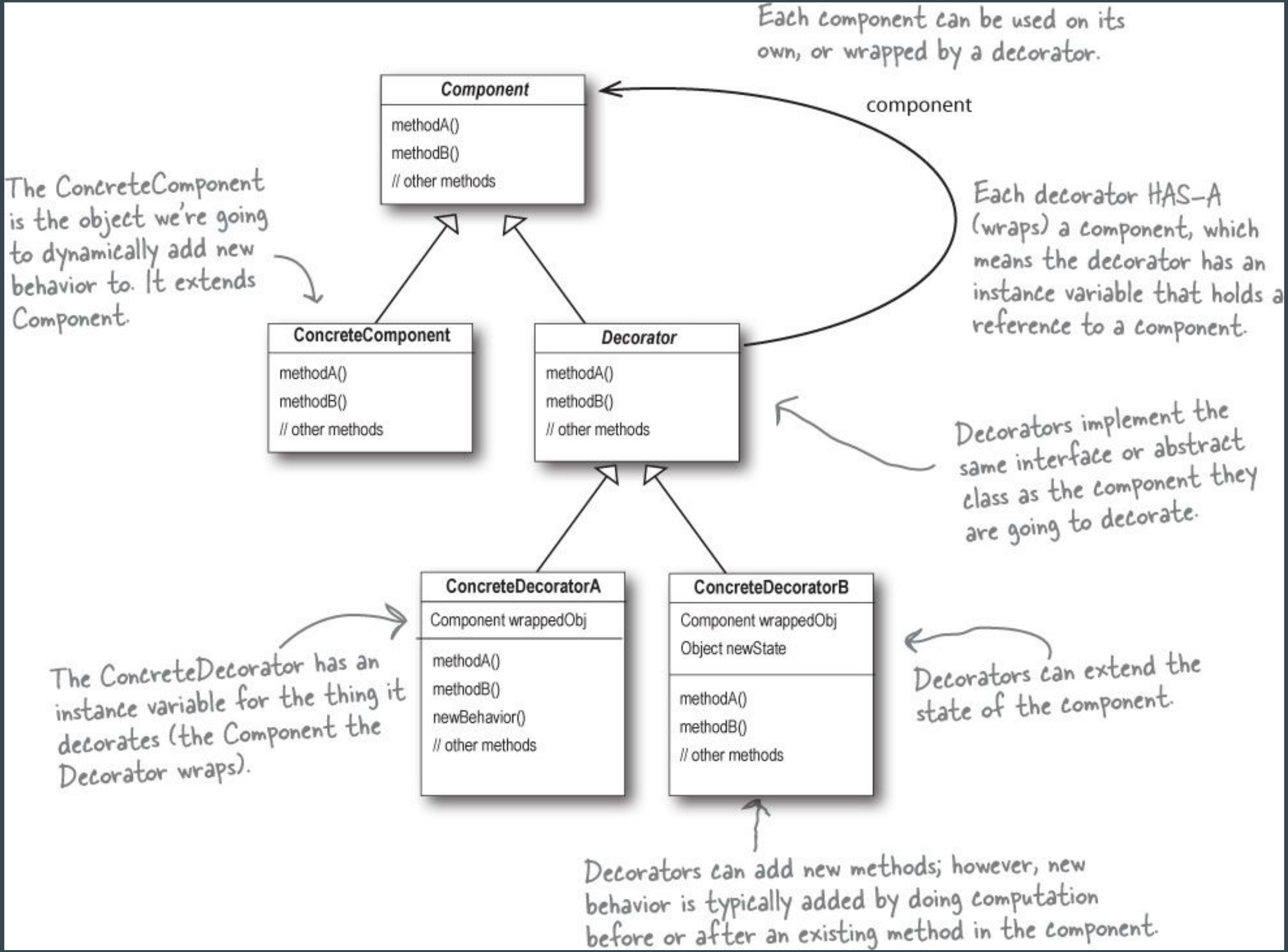
Now we can compute the cost as in the diagram to the right



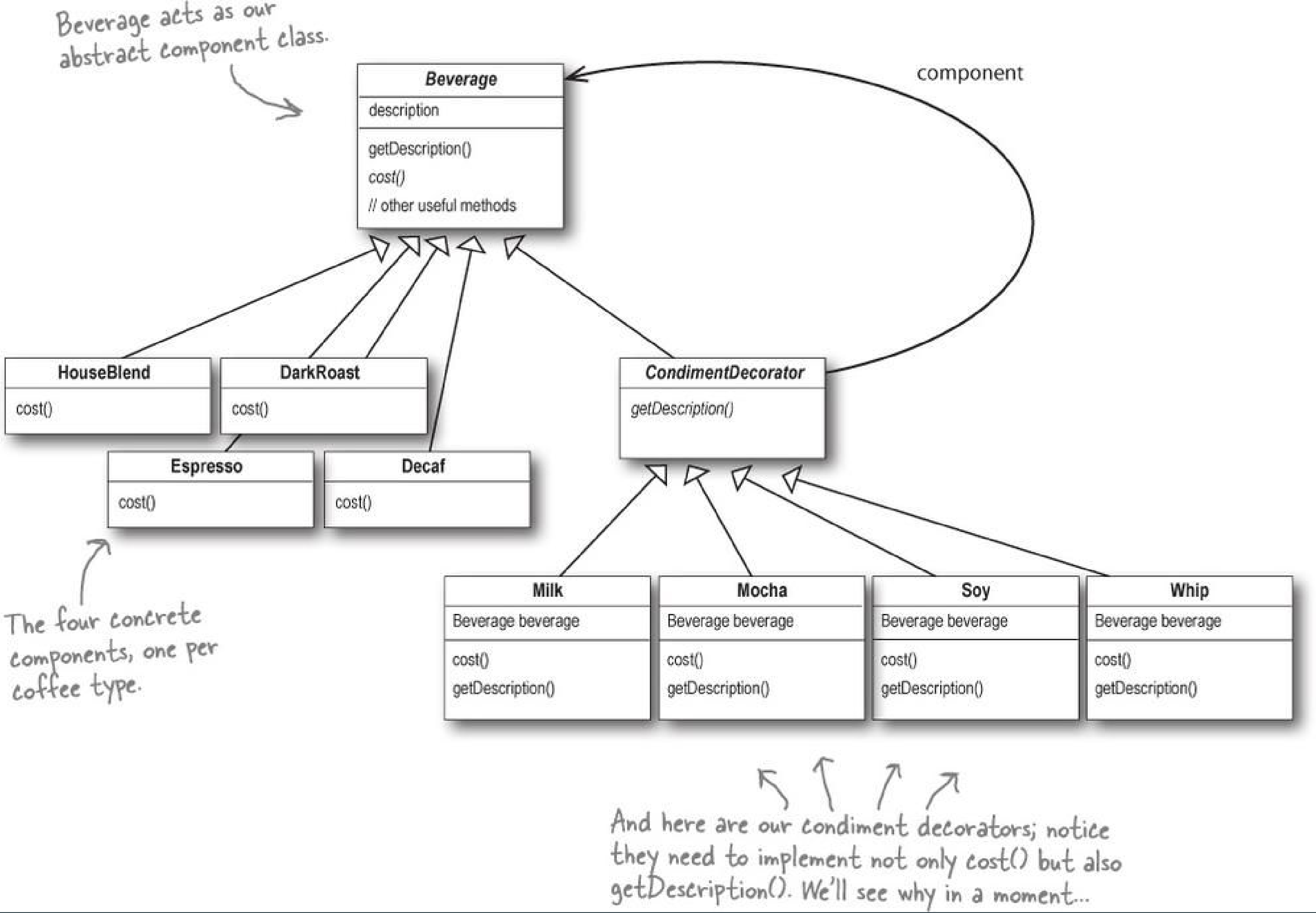
The Decorator Pattern defined

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

How it's implemented



Applied to beverages



Code Demo

See code

Real world decorators: Java I/O