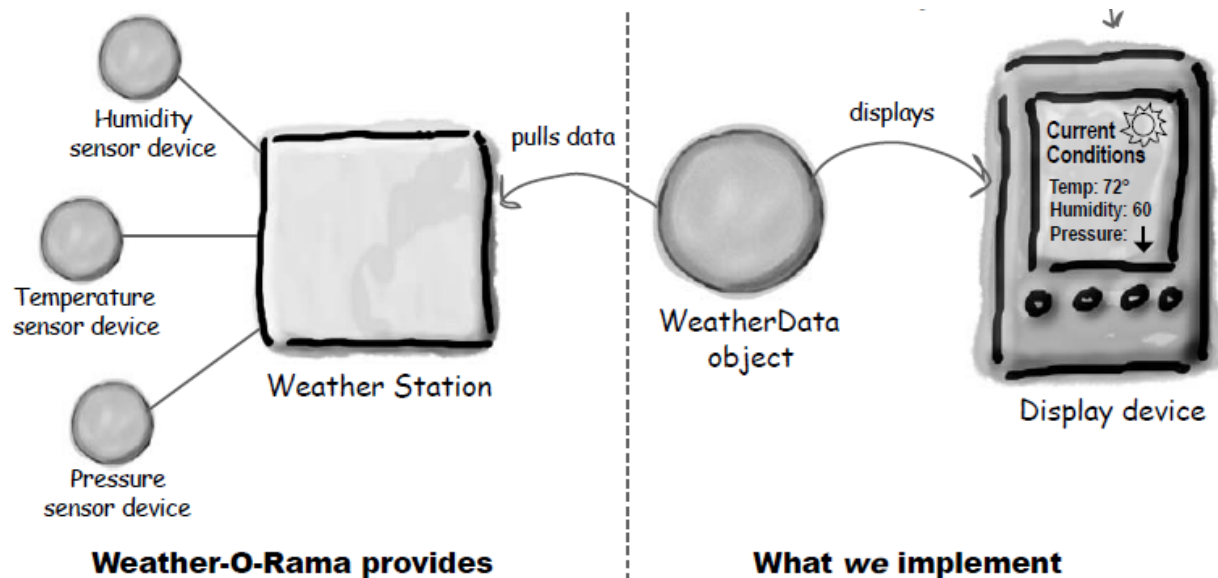# Design Patterns

Observer and Decorator

Some slides are adapted from Jon Simon's (jonathan_simon@yahoo.com)

Observer Pattern

- Weather Monitoring Application

Weather Monitoring Application

- Weather Station which monitors humidity, temperature, and pressure.

- There are three different displays for showing the information from the Weather Station
  - Current Conditions
  - Weather Stats
  - Forecast

- Goal:
  - Changes in the Weather Station must update all three displays.
  - *In addition, need capability to add additional displays in the future.*

WeatherData Class

WeatherData

getTemperature()
getHumidity()
getPressure()
measurementsChanged()
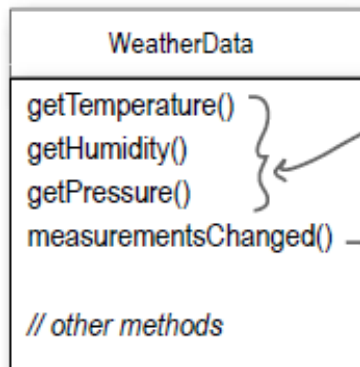
// other methods

These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

The developers of the WeatherData object left us a clue about what we need to add...

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

WeatherData Class Implementation

```
public class WeatherData {

    // instance variable declarations          기상 상황이 바뀌었다.!

    public void measurementsChanged() {
                                                Grab the most recent measuremets
        float temp = getTemperature();          by calling the WeatherData's getter
        float humidity = getHumidity();         methods (already implemented).
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);   Now update
        statisticsDisplay.update(temp, humidity, pressure);          the displays...
        forecastDisplay.update(temp, humidity, pressure);
    }
                                                Call each display element to
    // other WeatherData methods here           update its display, passing it the
}                                               most recent measurements.
```
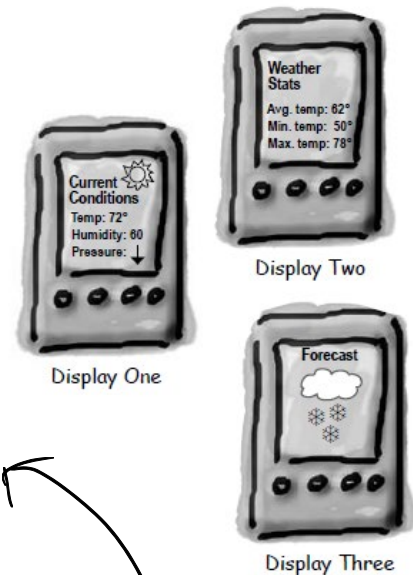
What's Wrong with the Implementation

```
public void measurementsChanged() {

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

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

}
```

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 takes the temp, humidity, and pressure values.

Display One

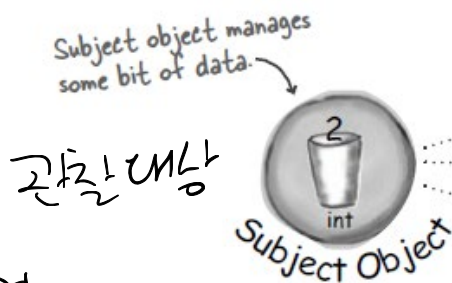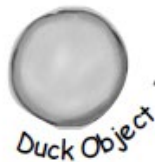Display Two

Display Three

Weather Stats
Avg. temp: 62°
Min. temp: 50°
Max. temp: 78°

Current Conditions
Temp: 72°
Humidity: 60
Pressure: ↓

Forecast

Future displays
?

어디에 추가시켜야되나?

하나더생기면?

Observer Pattern (Publishers + Subscribers)

Weather Data

관찰대상

바꾸면
관찰자들에게 통보해줌
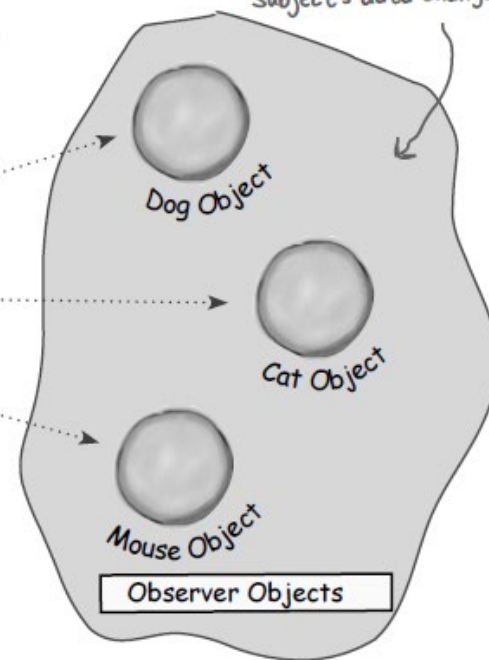
Subject object manages
some bit of data.

When data in the Subject changes,
the observers are notified.

2

2

2

New data values are
communicated to the
observers in some form
when they change.

Subject Object
int
2

The observers have subscribed to
(registered with) the Subject
to receive updates when the
Subject's data changes.

Display

관찰자

Dog Object

Cat Object

Mouse Object

Observer Objects

This object isn't an
observer, so it doesn't
get notified when the
Subject's data changes.
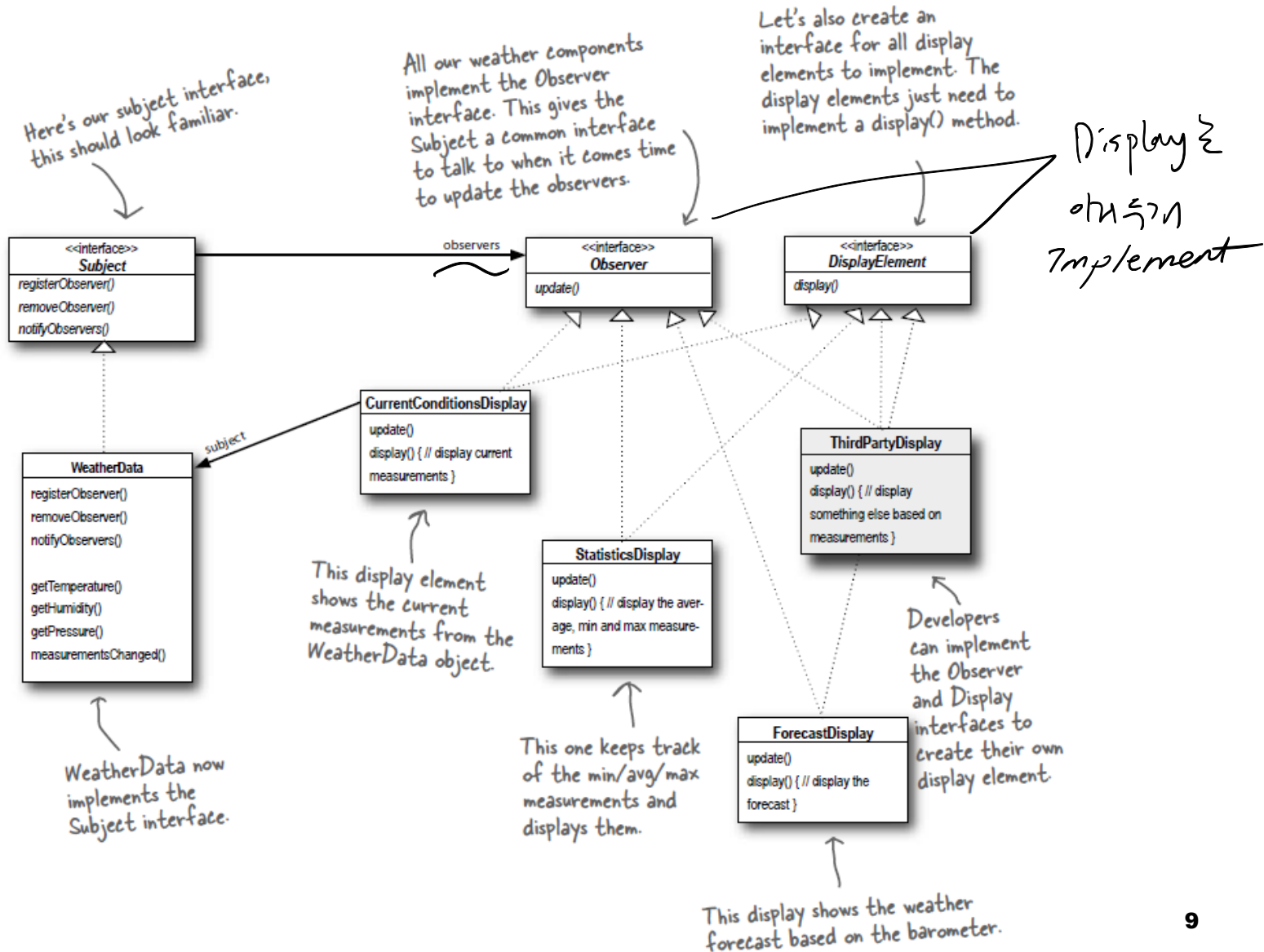
Duck Object

Definitions of Observer Pattern

- Subject
  - □ Sends a notification message to one or many observers when there is a change in state.
  - □ Along with the message, provides information on the state that has changed.

- Observers – The objects that want to be notified about changes in the Subject's state

Designing the Weather Station

Implementing the Weather Station

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

Implementing the Weather Station

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

Here we implement the Subject Interface.

Implementing the Weather Station

```
public void measurementsChanged() {
    notifyObservers();
}
```

We notify the Observers when we get updated measurements from the Weather Station.

```
public void setMeasurements(float temperature, float humidity, float pressure) {
    this.temperature = temperature;
    this.humidity = humidity;
    this.pressure = pressure;
    measurementsChanged();
}

// other WeatherData methods here
}
```

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the web.

Implementing the Weather Station

This display implements Observer so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```java
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.

Implementing the Weather Station

```java
public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

First, create the WeatherData object

If you don't want to download the code, you can comment out these two lines and run it

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

Observer Pattern

- **Power of loose coupling**
  - ☐ The only thing the subject knows about an observer is that it implements a certain interface.
  - ☐ We can add new observers at any time.
  - ☐ We never need to modify the subject to add new types of observers.
  - ☐ We can reuse subjects or observers independently of each other.
  - ☐ Changes to either the subject or an observer will not affect the other.

- **Allows flexible OO systems that can handle changes easily, because they minimize the interdependency between objects.**

Observer Pattern

- GoF Intent: "Defines a one-to-many dependency between objects so that when one object *changes state*, all of its dependents are notified and updated automatically."

- Java's built-in support for the observer pattern
  - java.util.Observer
  - java.util.Observable

- Listener method model for Swing API

Decorator Pattern

- ## Starbuzz Coffee
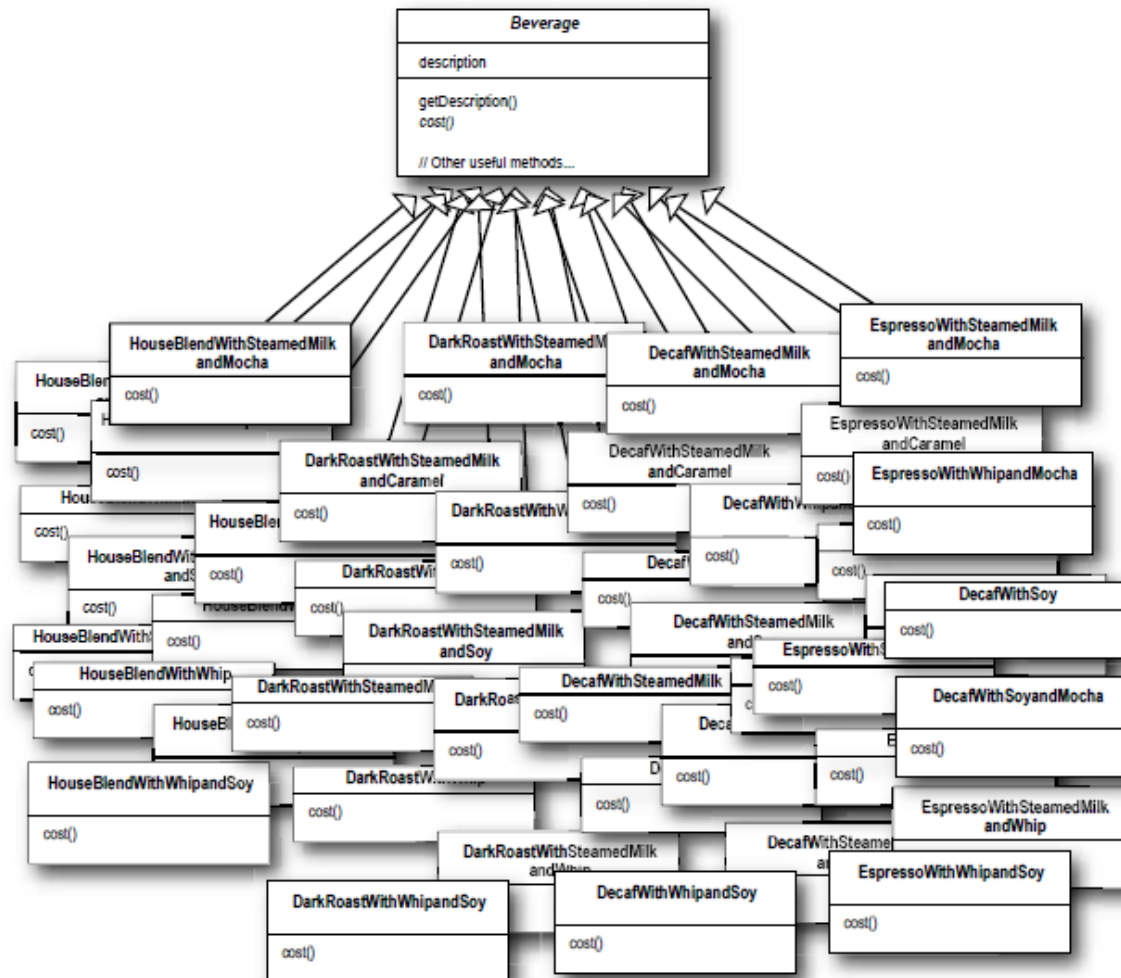
Starbuzz Coffee

Now let's add in the subclasses, one for each beverage on the menu:

The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.

Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().

Beverage

description
milk
soy
mocha
whip

getDescription()
cost()

hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()

// Other useful methods..

HouseBlend

cost()

DarkRoast

cost()

Decaf

cost()

Espresso

cost()
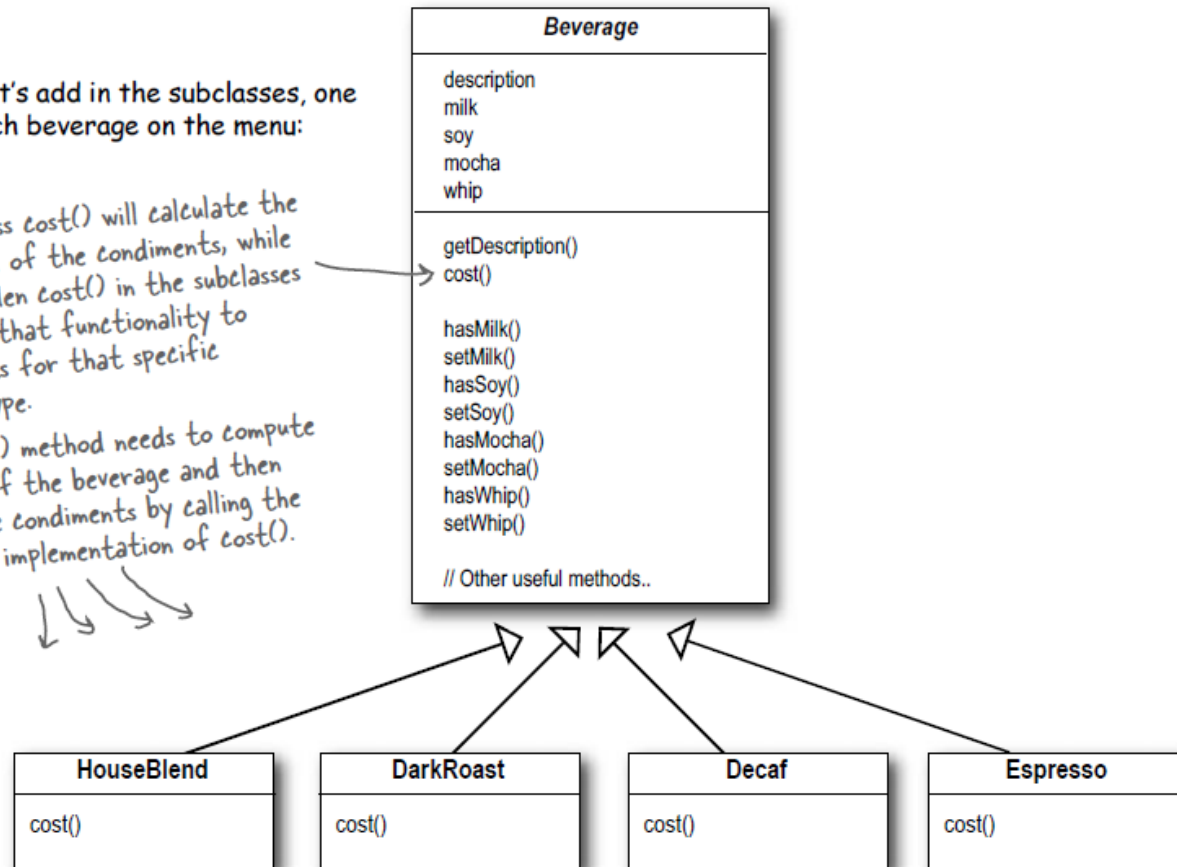
Starbuzz Coffee

- The Open-Closed Principle: "Classes should be open for extension, but closed for modification."
  - ☐ Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.
  - ☐ We're going to strive to design our system so that the closed parts are isolated from our new extensions.

- Designs that are resilient to changes and flexible enough to take on new functionality to meet changing requirements

Meet the Decorator Pattern

- We'll start with a beverage and "decorate" it with the condiments at runtime.

- For example, if the customer wants a Dark Roast with Mocha and Whip, then we'll:
  - ☐ Take a DarkRoast object
  - ☐ Decorate it with a Mocha object
  - ☐ Decorate it with a Whip object
  - ☐ Call the cost() method and rely on delegation to add on the condiment costs

Meet the Decorator Pattern

- We can compute the cost by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.

② Whip calls cost() on Mocha.

① First, we call cost() on the outmost decorator, Whip.

③ Mocha calls cost() on DarkRoast.

cost() cost() cost()

$1.29 ← .10 .20 .99 DarkRoast

Mocha

Whip

④ DarkRoast returns its cost, 99 cents.

⑤ Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

⑤ Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

Decorating our Beverages

Beverage acts as our abstract component class.

Beverage
description
getDescription()
cost()
// other useful methods

component

HouseBlend
cost()

DarkRoast
cost()

Espresso
cost()

Decaf
cost()

CondimentDecorator
getDescription()

The four concrete components, one per coffee type.

Milk
Beverage beverage
cost()
getDescription()

Mocha
Beverage beverage
cost()
getDescription()

Soy
Beverage beverage
cost()
getDescription()

Whip
Beverage beverage
cost()
getDescription()

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

Writing the Starbuzz code

```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```
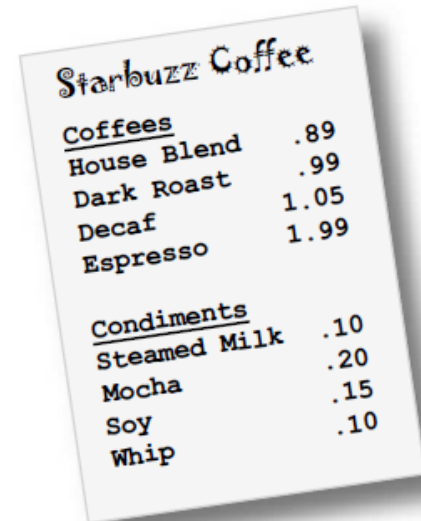
Writing the Starbuzz code

```
public class HouseBlend extends Beverage {
    public HouseBlend() {
        description = "House Blend Coffee";
    }

    public double cost() {
        return .89;
    }
}


public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

Starbuzz Coffee

Coffees
House Blend   .89
Dark Roast   .99
Decaf   1.05
Espresso   1.99

Condiments
Steamed Milk   .10
Mocha   .20
Soy   .15
Whip   .10

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

Writing the Starbuzz code

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to to pass the beverage we're wrapping to the decorator's constructor.

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Serving some coffees

```java
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

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

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

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.
Wrap it with a Mocha.
Wrap it in a second Mocha.
Wrap it in a Whip.

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

```
File  Edit  Window  Help  CloudsInMyCoffee
% java StarbuzzCoffee
Espresso $1.99
Dark Roast Coffee, Mocha, Mocha, Whip $1.49
House Blend Coffee, Soy, Mocha, Whip $1.34
%
```
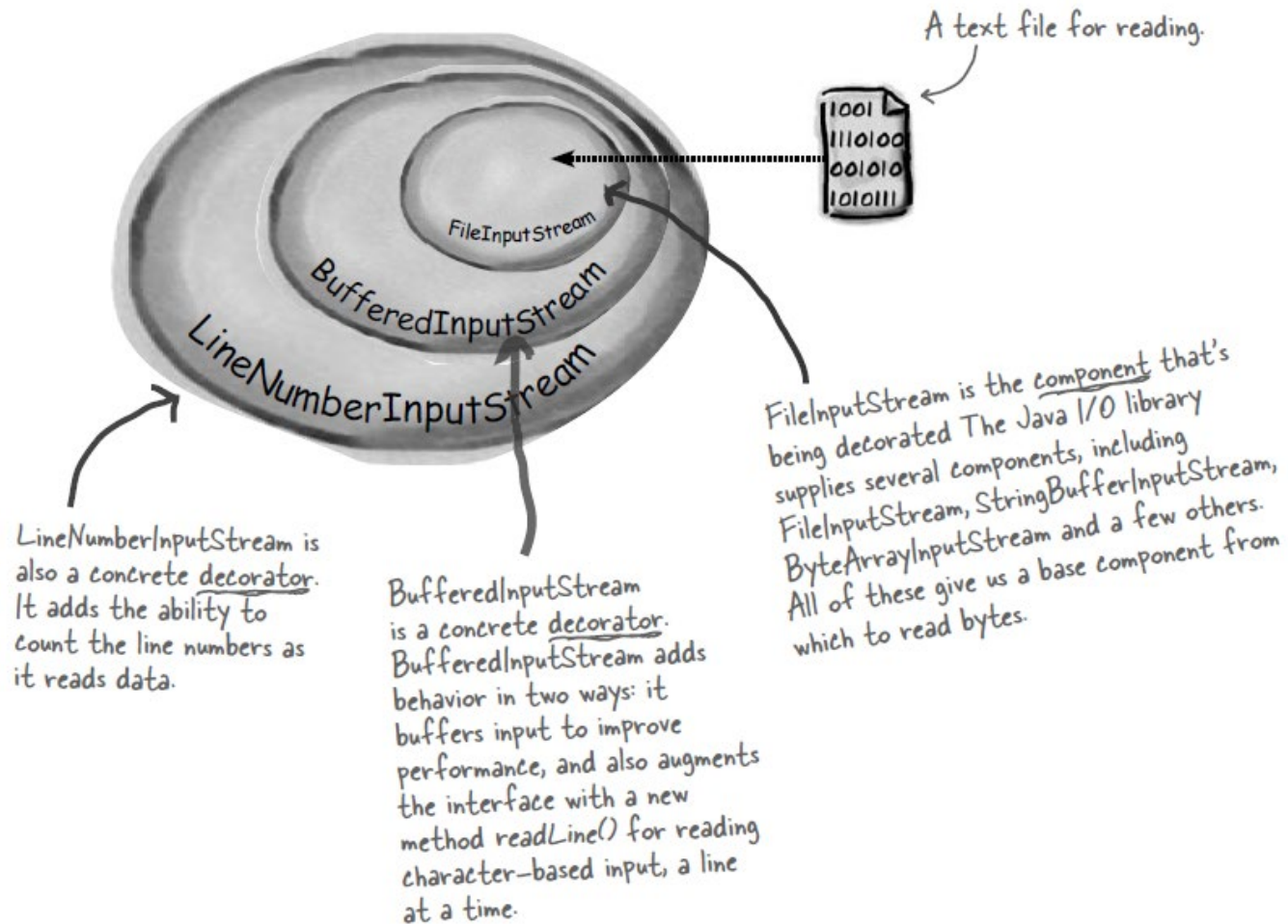
6/8/2020

Real World Decorators: Java I/O

A text file for reading.

FileInputStream

BufferedInputStream

LineNumberInputStream

LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

FileInputStream is the component that's being decorated The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

Real World Decorators: Java I/O

Here's our abstract component.

InputStream

FilterInputStream is an abstract decorator.

FileInputStream | StringBufferInputStream | ByteArrayInputStream | FilterInputStream

These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.

PushbackInputStream | BufferedInputStream | DataInputStream | LineNumberInputStream

And finally, here are all our concrete decorators.

Lab (Java I/O Package)

.

Writing your own Java I/O Decorator – Refer to the HFDP book

Decorator

- GoF Intent: "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."