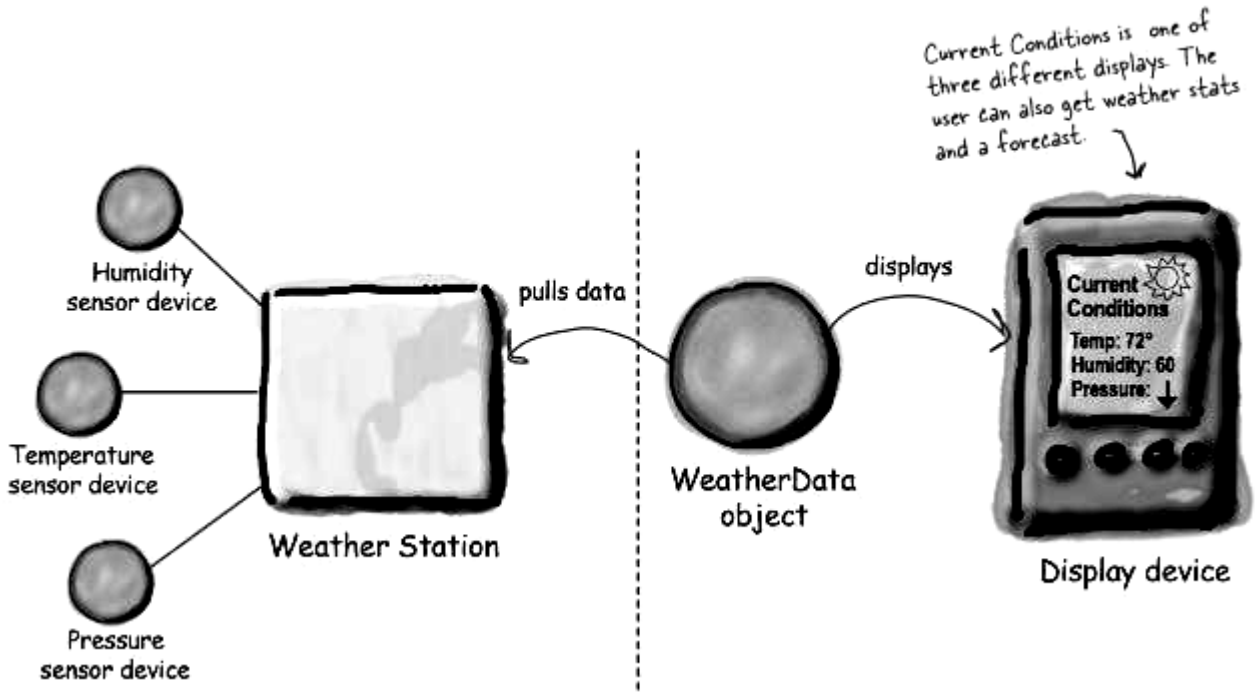
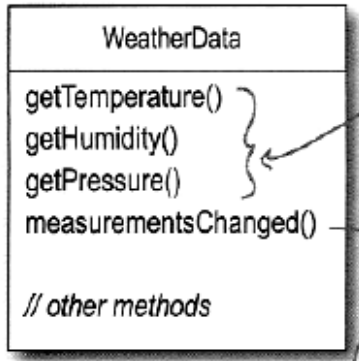
Observer Pattern

Weather Monitoring



Weather Data Class



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

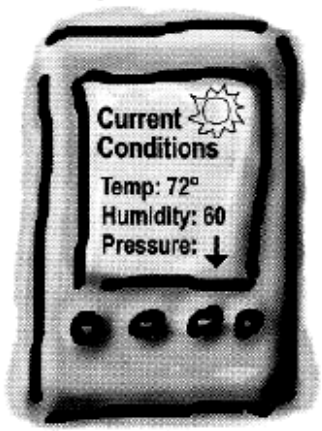
We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

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.java

Remember, this Current Conditions is just ONE of three different display screens.



Display device

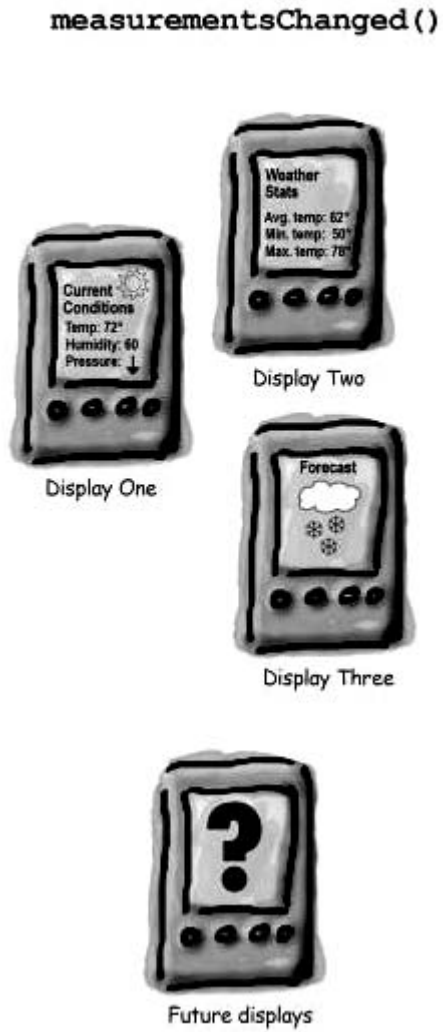
Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

What we know so far

System must be expandable.

Other developers can create new custom display elements.

Users can add or remove as many display elements as they want to the application.



First Implementation Possibility

```
public class WeatherData {
    …
    public void measurementsChanged() {
        float temperature = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

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

What's wrong?

What's wrong?

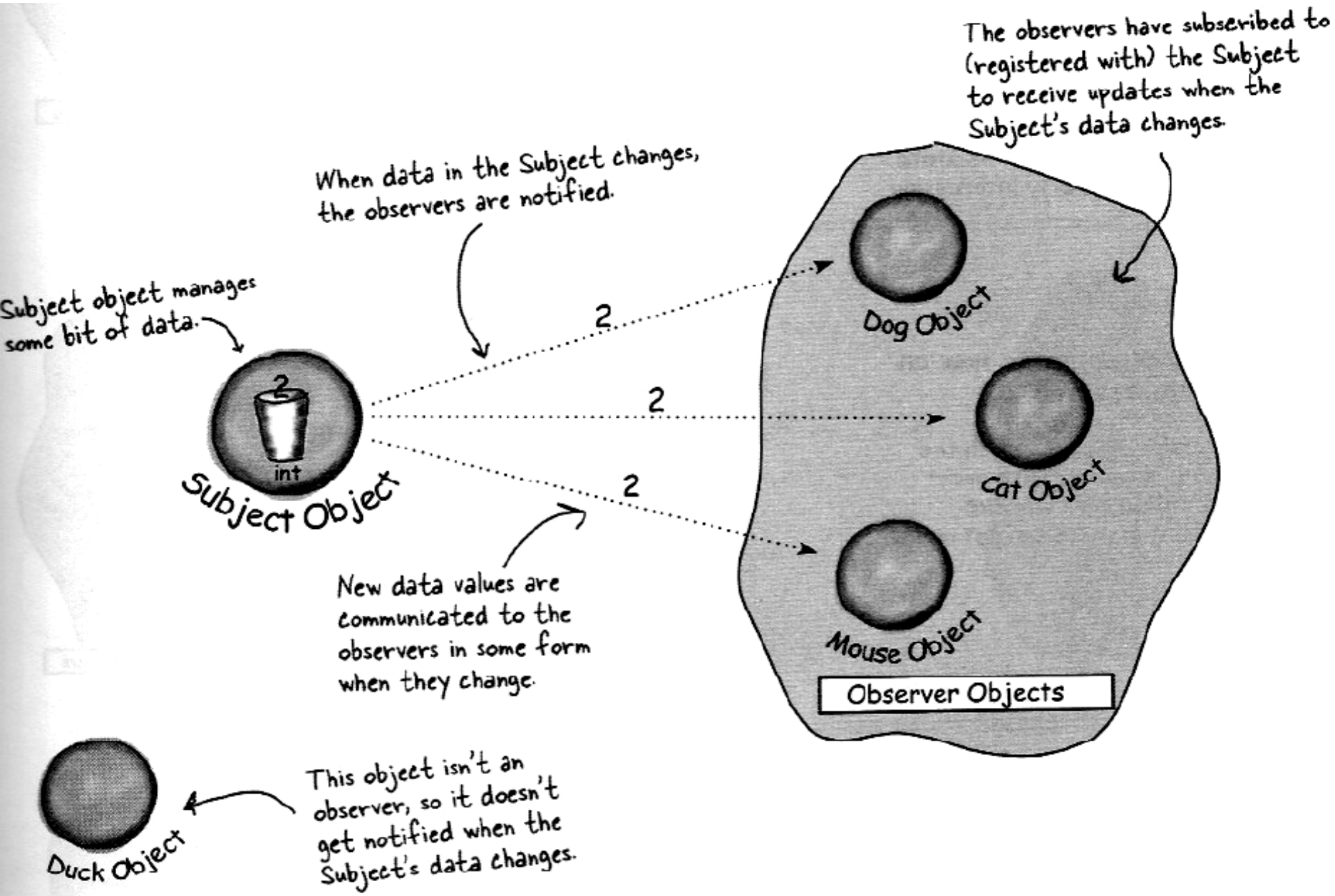
```
public void measurementsChanged() {

    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.

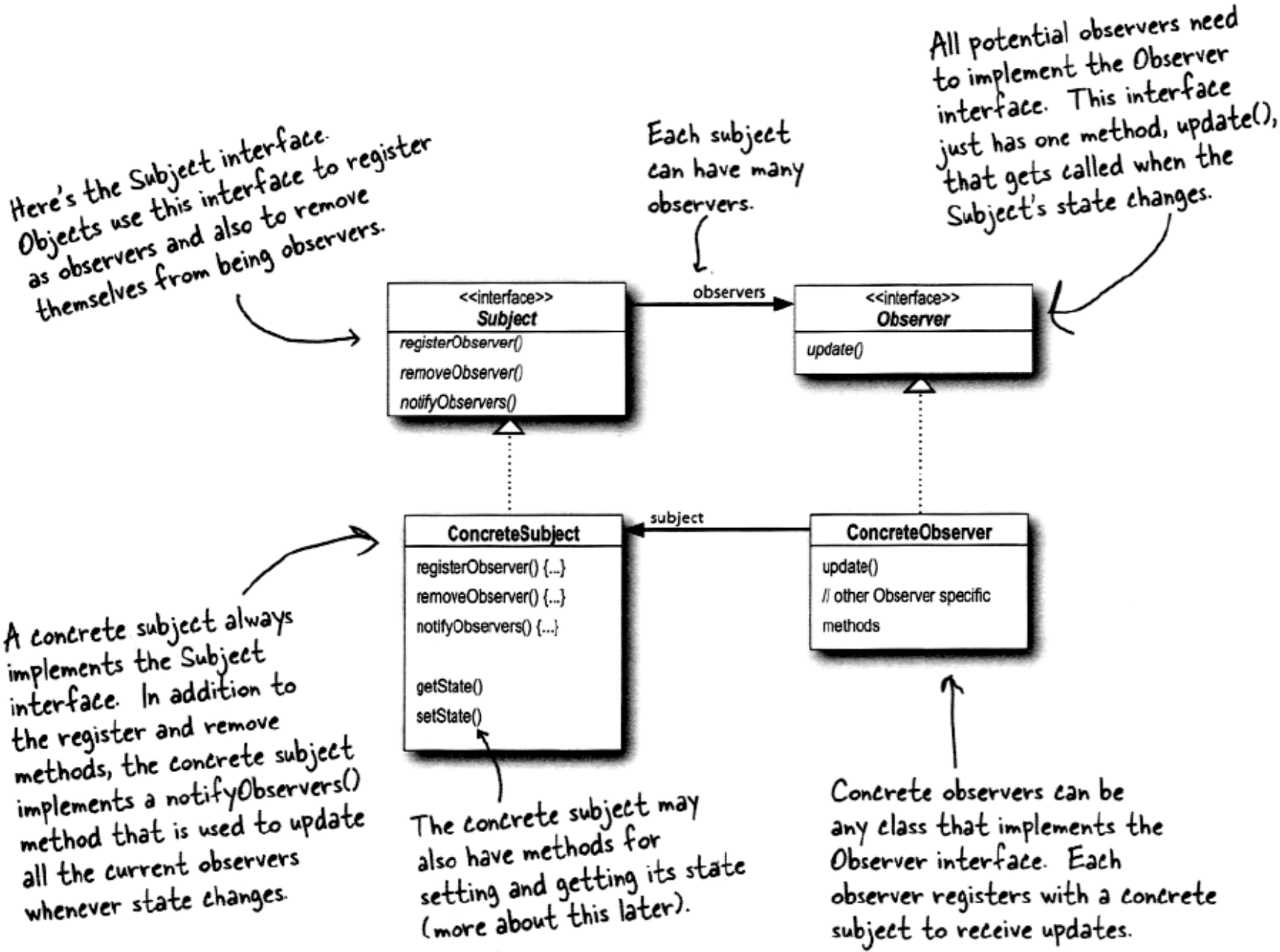
By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

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

Publisher + Subscribers



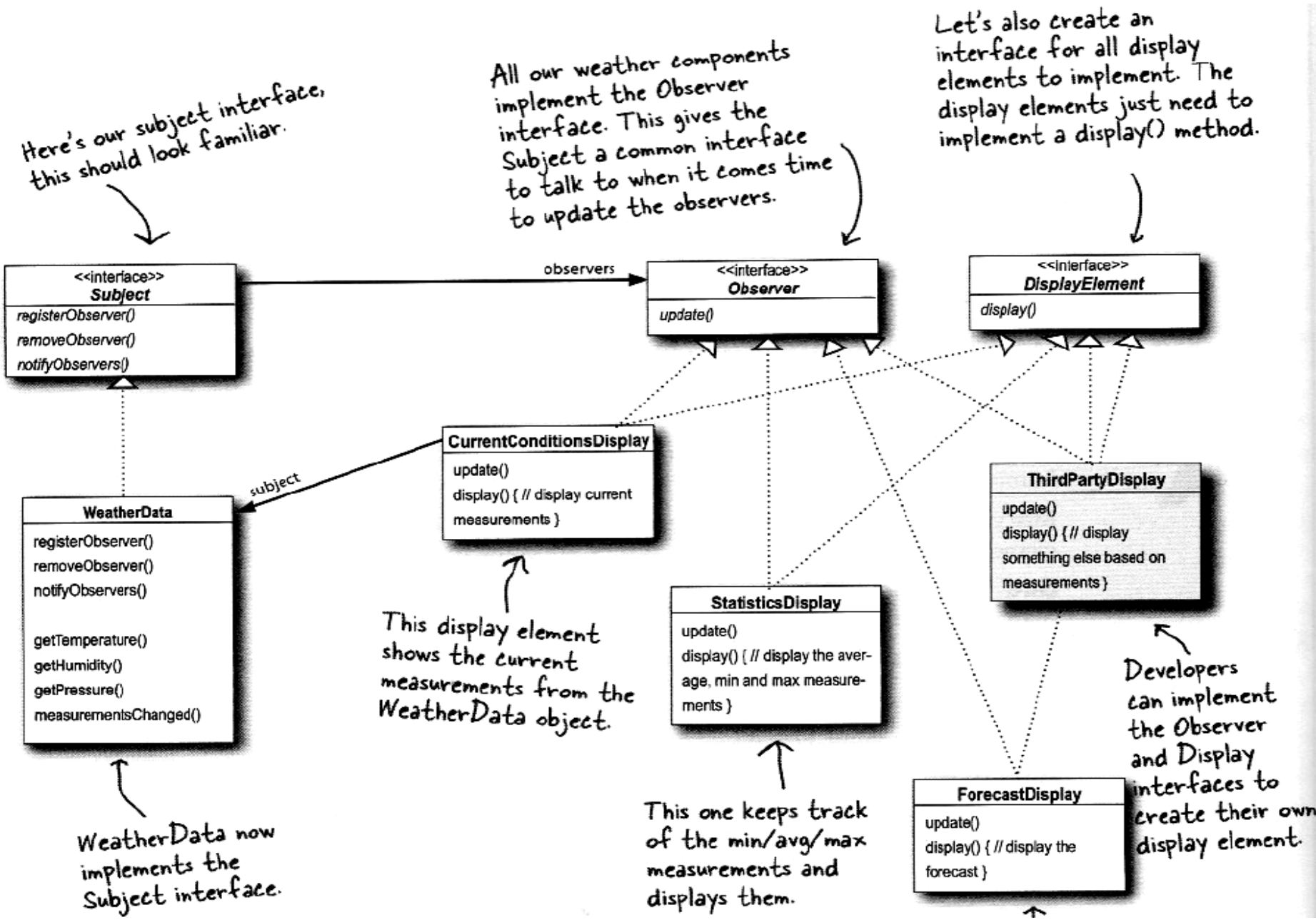
Observer Pattern



The Power of Loose Coupling

- The only thing the subject knows about an observer is that it implements a certain interface.
- We can add observers at any time. Likewise, we can remove observes 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.

UML design


Here's our subject interface, this should look familiar.
All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.
Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
<<interface>>
DisplayElement
display()
CurrentConditionsDisplay
update()
display() { // display current measurements }
subject
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
ForecastDisplay
update()
display() { // display the forecast }
This display element shows the current measurements from the WeatherData object.
Developers can implement the Observer and Display interfaces to create their own display element.
WeatherData now implements the Subject interface.
This one keeps track of the min/avg/max measurements and displays them.

Subject and Observer

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

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

WeatherData

```
import java.util.*;

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); }
    }
```

WeatherData

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

    public void measurementsChanged() { notifyObservers(); }

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

CurrentConditionsDisplay

```
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");
    }
}
```

ForecastDisplay

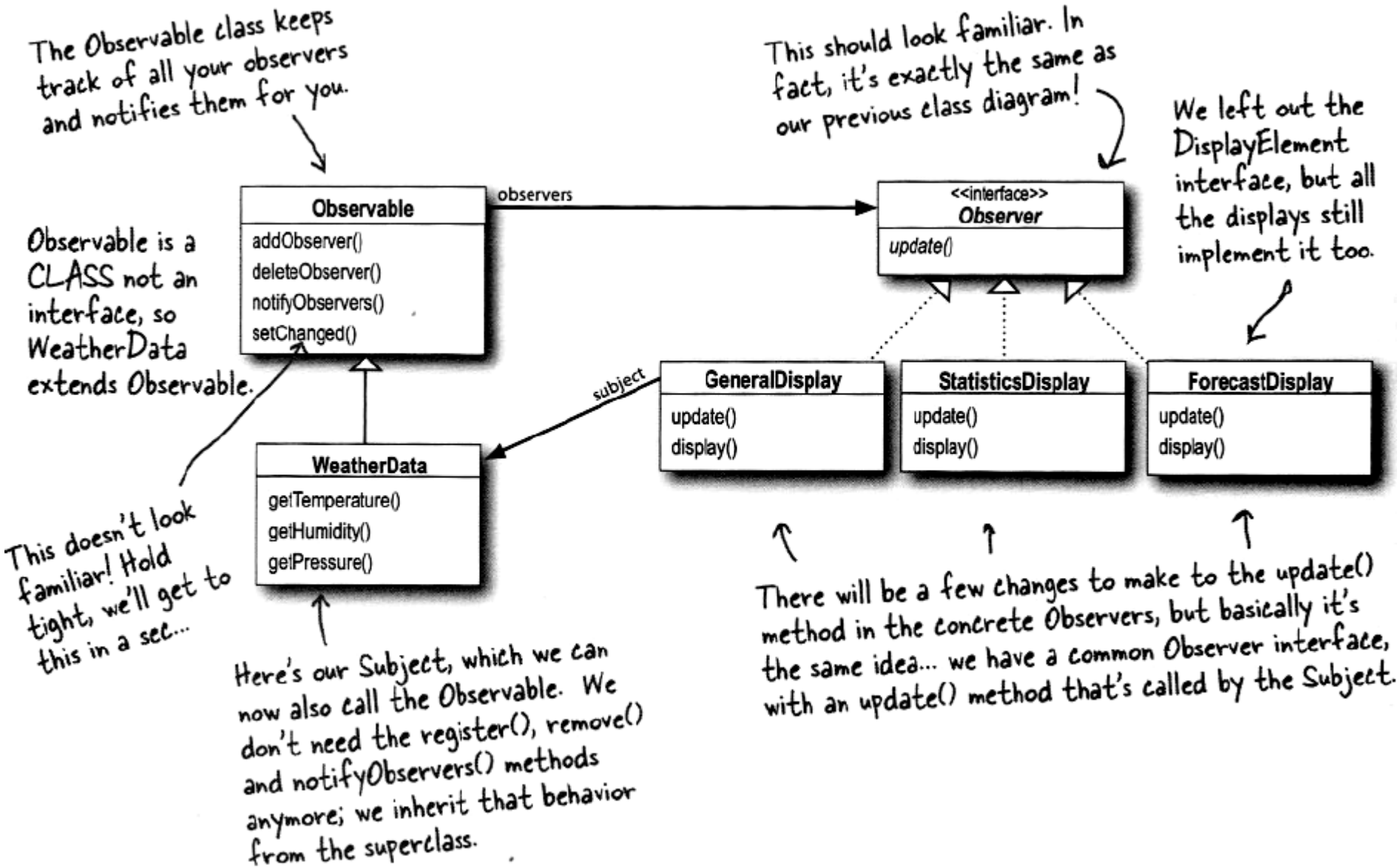
```
public class ForecastDisplay implements Observer, DisplayElement {
    private float currentPressure = 29.92f;
    private float lastPressure;
    private WeatherData weatherData;

    public ForecastDisplay(WeatherData weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temp, float humidity, float pressure) {
        lastPressure = currentPressure;
        currentPressure = pressure;
        display();
    }
```

ForecastDisplay

```
public class ForecastDisplay implements Observer, DisplayElement {
…
    public void display() {
        System.out.print("Forecast: ");
        if (currentPressure > lastPressure) {
            System.out.println("Improving weather on the way!");
        } else if (currentPressure == lastPressure) {
            System.out.println("More of the same");
        } else if (currentPressure < lastPressure) {
            System.out.println("Watch out for cooler, rainy weather");
        }
    }
}
```

…and other observer classes.

Java built in Observer Pattern



How it works

- For an Object to become an observer:
  - implement the java.util.Observer interface,
  - call addObserver() on any Observable object
  - To remove object as an observer call deleteObserver()
- For the Observable to send notifications:
  - first call setChanged() method to signify that the state has changed in the Observable.
  - then, call one of the two notifyObservers() methods:
    - notifyObservers() or notifyObservers(Object arg)
- For an Observer to receive notifications:
  - It implements the update method, as before, but the signature is a bit different: update(Observable o, Object arg)

For the Observable to send notifications...

1. **You first must call the setChanged() method to signify that the state has changed in your object**

2. **Then, call one of two notifyObservers() methods:**

either `notifyObservers()` **or** `notifyObservers(Object arg)`

This version takes an arbitrary data object that gets passed to each Observer when it is notified.

For an Observer to receive notifications...


```
update(Observable o, Object arg)
```

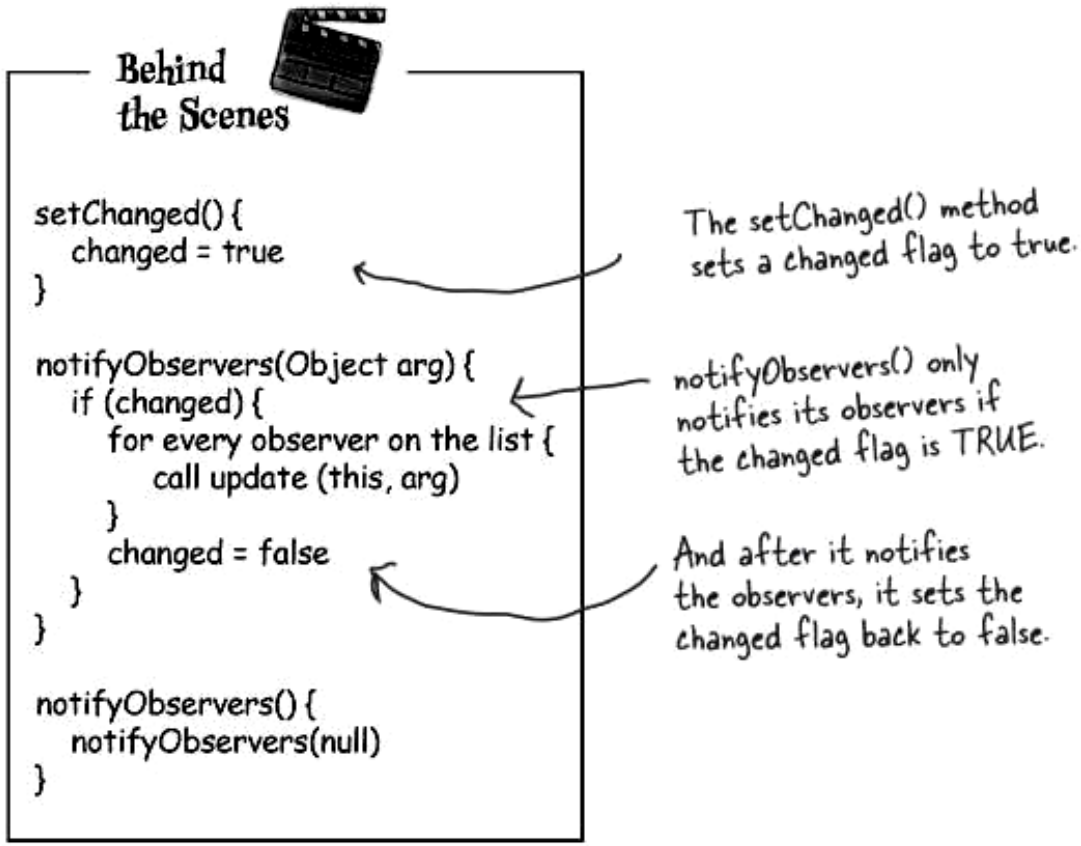
The Subject that sent the notification is passed in as this argument.

This will be the data object that was passed to notifyObservers(), or null if a data object wasn't specified.

setChanged() method

- The setChanged() method is used to signify that the state has changed and that notifyObservers(), when it is called, should update its observers.

- If notifyObservers() is called without first calling setChanged(), the observers will NOT be notified.



Push versus Pull

- If you want to “push” data to the observers you can pass the data as a data object to the notifyObserver(arg) method.
- If not, then the Observer has to “pull” the data it wants from the Observable object passed to it.
- How?
- By providing getter methods in the Observable.

WeatherData

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers(); //Will let observers to pull
    }
```

WeatherData (the rest)

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

    public float getTemperature() {  return temperature; }
    public float getHumidity() { return humidity; }
    public float getPressure() { return pressure; }
}
```

CurrentConditionsDisplay

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }
    public void update(Observable obs, Object arg) {  //Pull method
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }
```

CurrentConditionsDisplay (the rest)

```
…

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