

# THE OBSERVER PATTERN

---

Chandan R. Rupakheti

Week 1-2

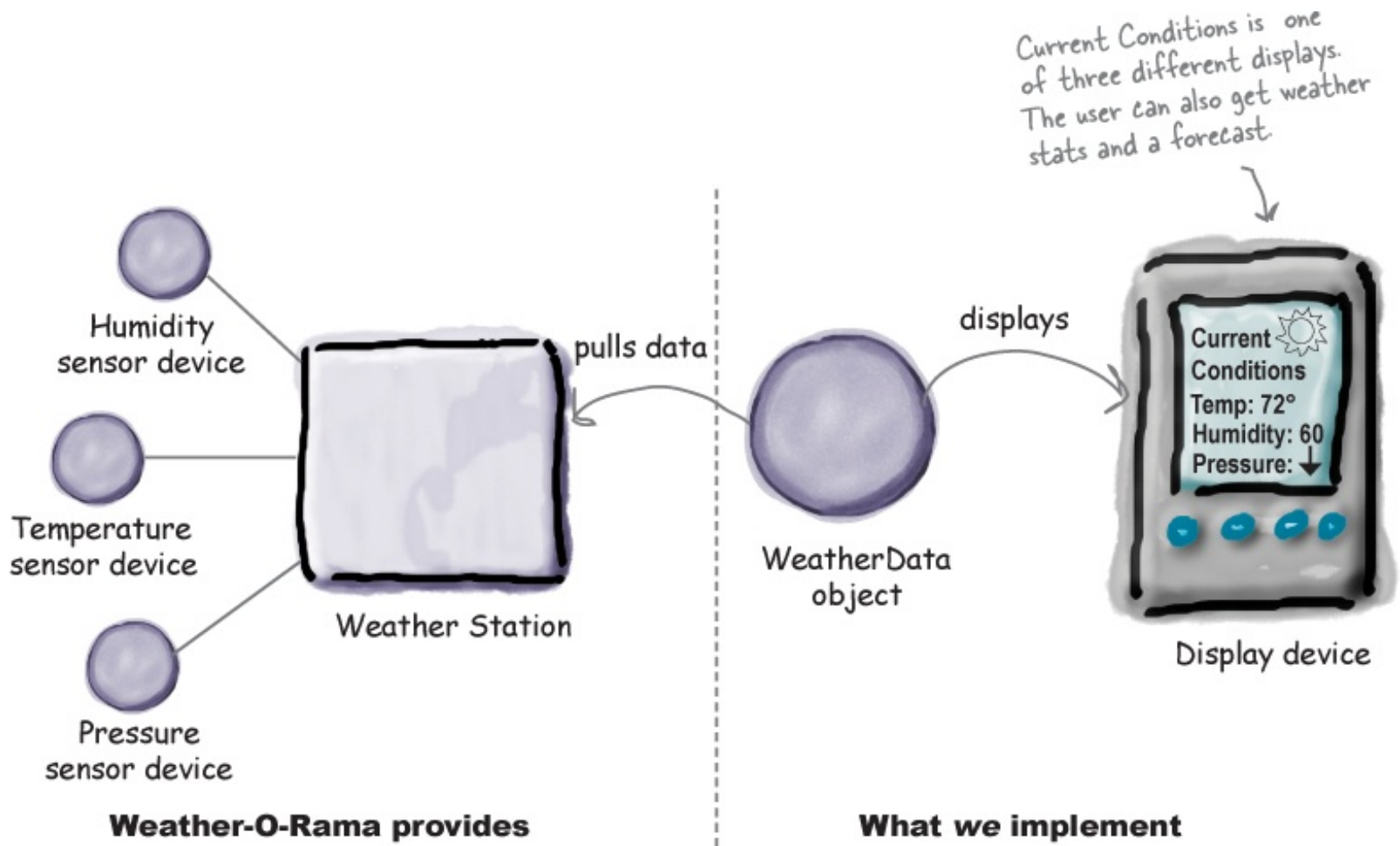
# Today



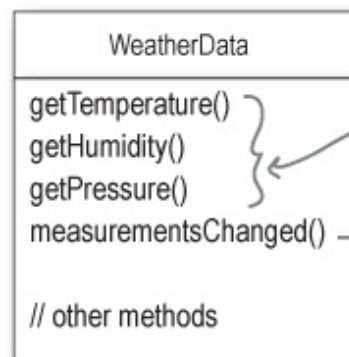
Hey Jerry, I'm notifying everyone that the Patterns Group meeting moved to Saturday night. We're going to be talking about the Observer Pattern. That pattern is the best! It's the BEST, Jerry!

- The Observer Pattern
- In-House vs Java's Built-In implementations
- Usage of the Observer Patterns
- But first, let's discuss about Monday's client meeting!

# The Weather Monitoring Application



# The WeatherData 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

# What do we know so far

The WeatherData class has getter methods for three measurement values:

**getTemperature()**  
**getHumidity()**  
**getPressure()**

The measurementsChanged() method is called any time new weather measurement data is available.

**measurementsChanged()**

We need to implement three display elements that use the weather data: a *current conditions* display, a *statistics display*, and a *forecast* display.

These displays must be updated each time WeatherData has new measurements.

**The system must be extensible** - other developers can create new custom display elements and users can add or remove as many display elements as they want to the application.

# First Misguided SWAG at the Weather Station

**SWAG: Scientific Wild A\*\* Guess**

```
public class WeatherData {  
  
    // instance variable declarations  
  
    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);  
    }  
  
    // other WeatherData methods here  
}
```

Grab the most recent measurements by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

# What's wrong with our implementation?

```
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);  
}
```

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.

Umm, I know I'm new here, but given that we are in the Observer Pattern chapter, maybe we should start using it?

Area of change. We need to encapsulate this.



# Meet the Observer Pattern

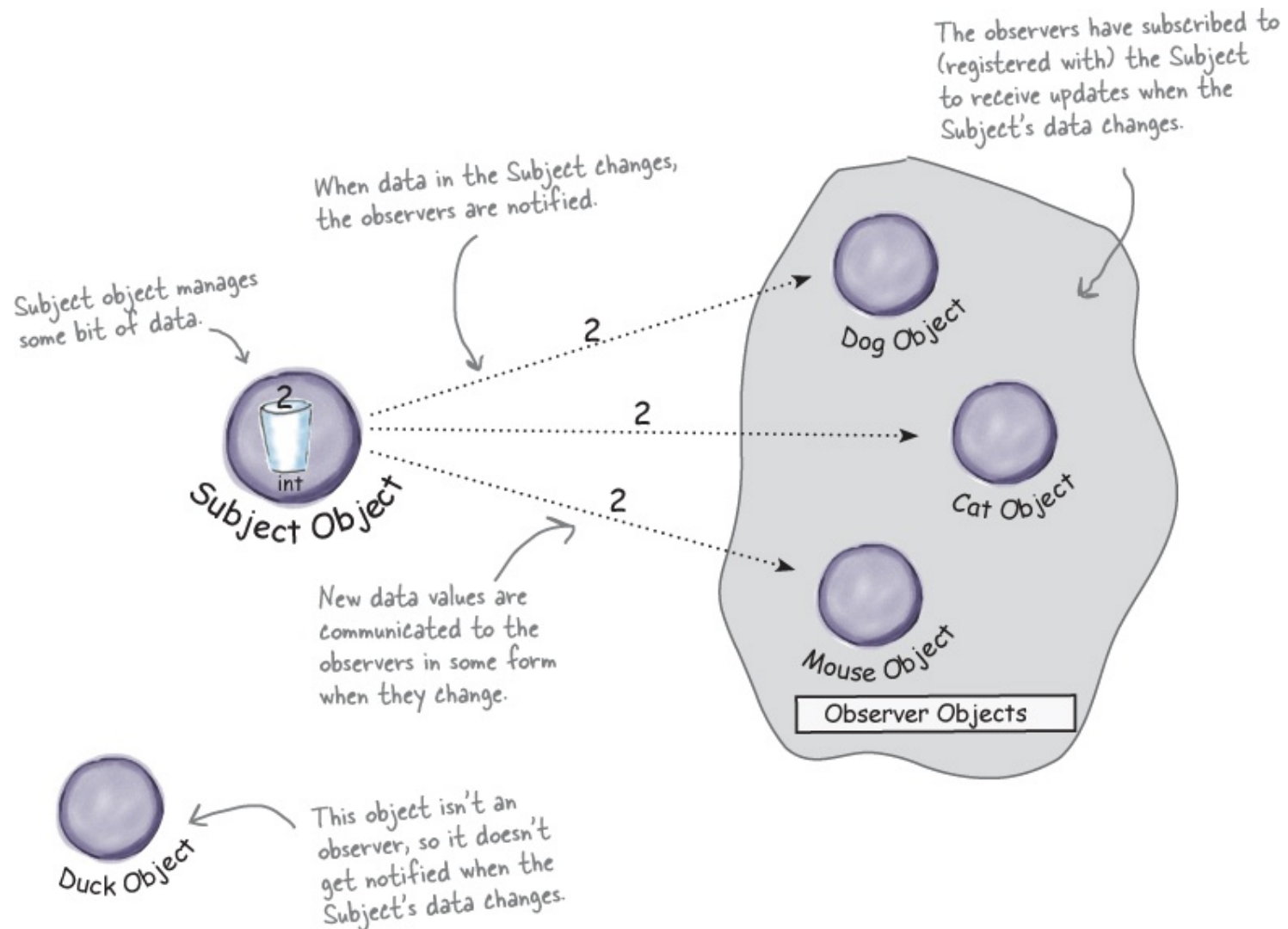
## Newspaper subscription



- A publisher goes into business
- If you subscribe, every time there's a new edition, it gets delivered to you
- If you unsubscribe, they stop being delivered
- Any organizations and people can subscribe and unsubscribe while the publisher remains in business



# Publishers + Subscribers = Observer Pattern



# Loose Coupling

**Design Principle:**

*Strive for loosely coupled designs  
between objects that interact*

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

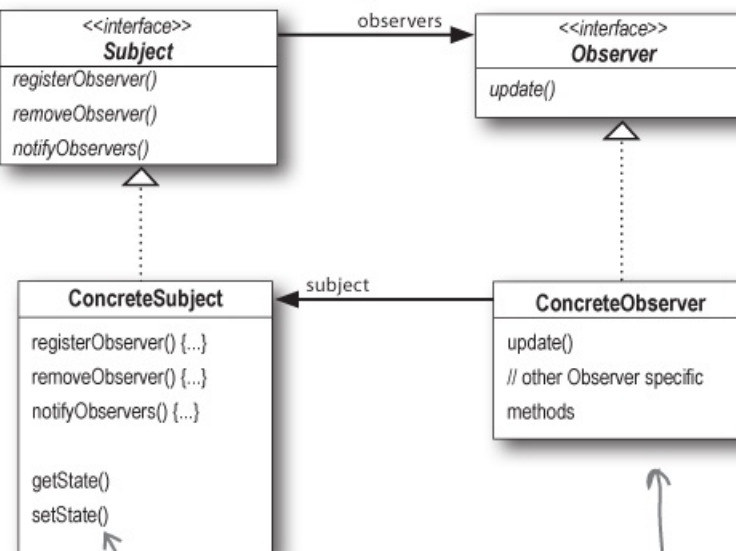
# The Observer Pattern

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

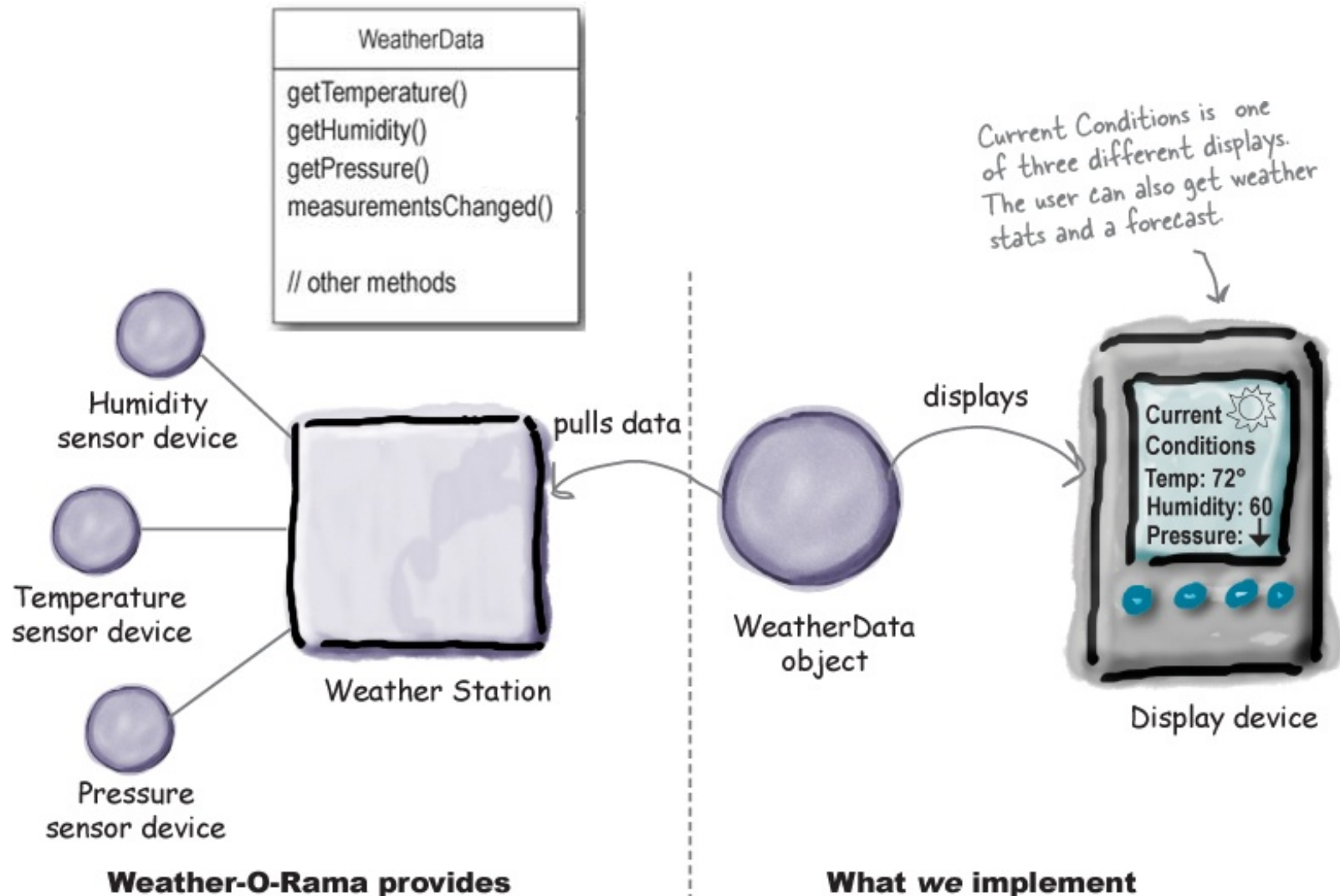


The concrete subject may also have methods for setting and getting its state (more about this later).

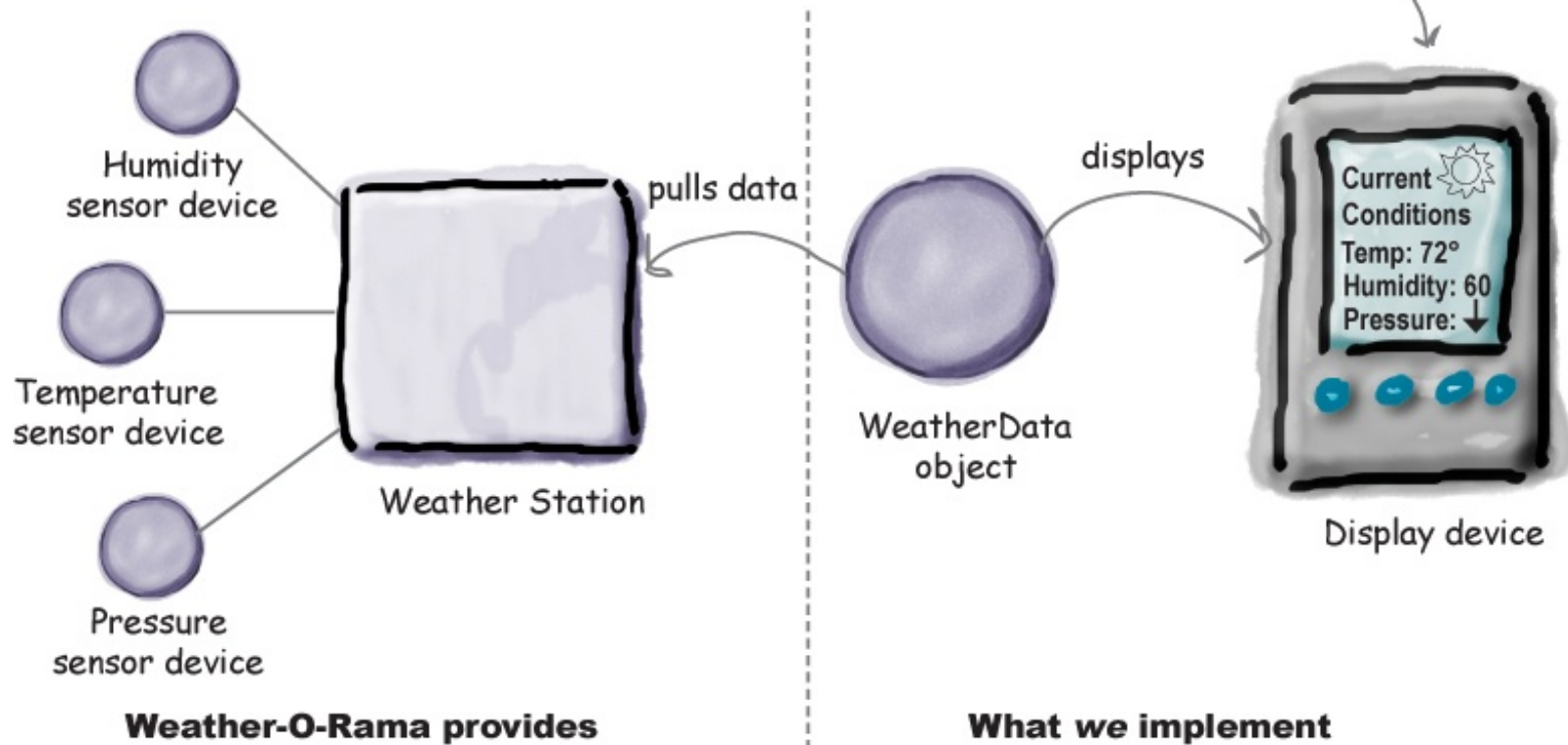
Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

# Design Studio

## Design the Weather Monitoring Application

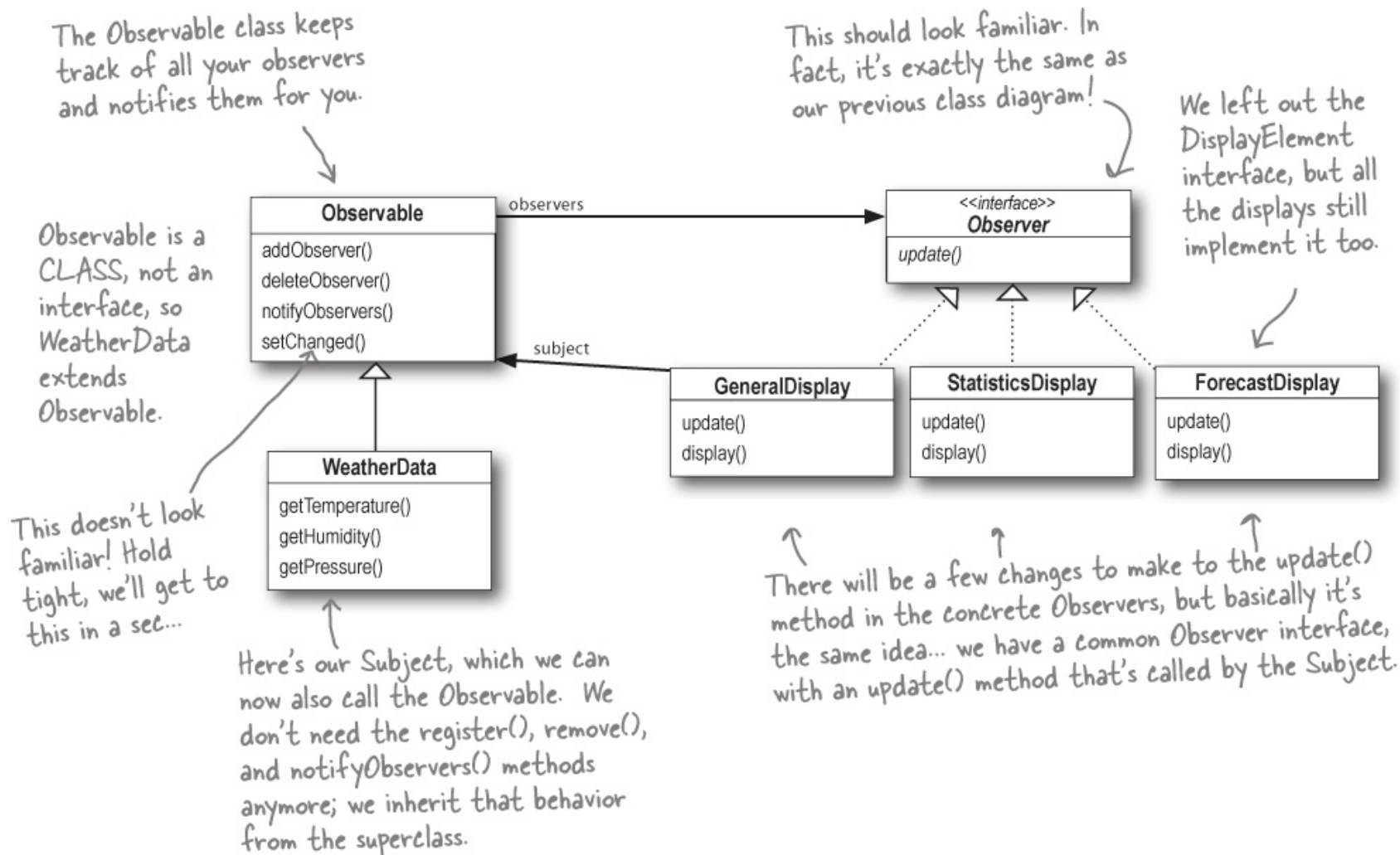


# Design Studio



See pp. 58 – 70 for solutions

# Solution Using Java's Observer Pattern



# Recap

## **Design Principle:**

*Strive for loosely coupled designs  
between objects that interact*

The Observer pattern allows loose  
coupling between Subject and  
Observers

Java's built-in Observer pattern  
violates DIP

Java's built-in Observer pattern  
favors inheritance over  
composition

# Next Week

- Things Due
  - Client meeting, during class
  - Homework 1 by Tuesday, 8:00 am
- Concepts
  - The Decorator Pattern
  - The Open-Close Principle
  - Coupling and Cohesion