

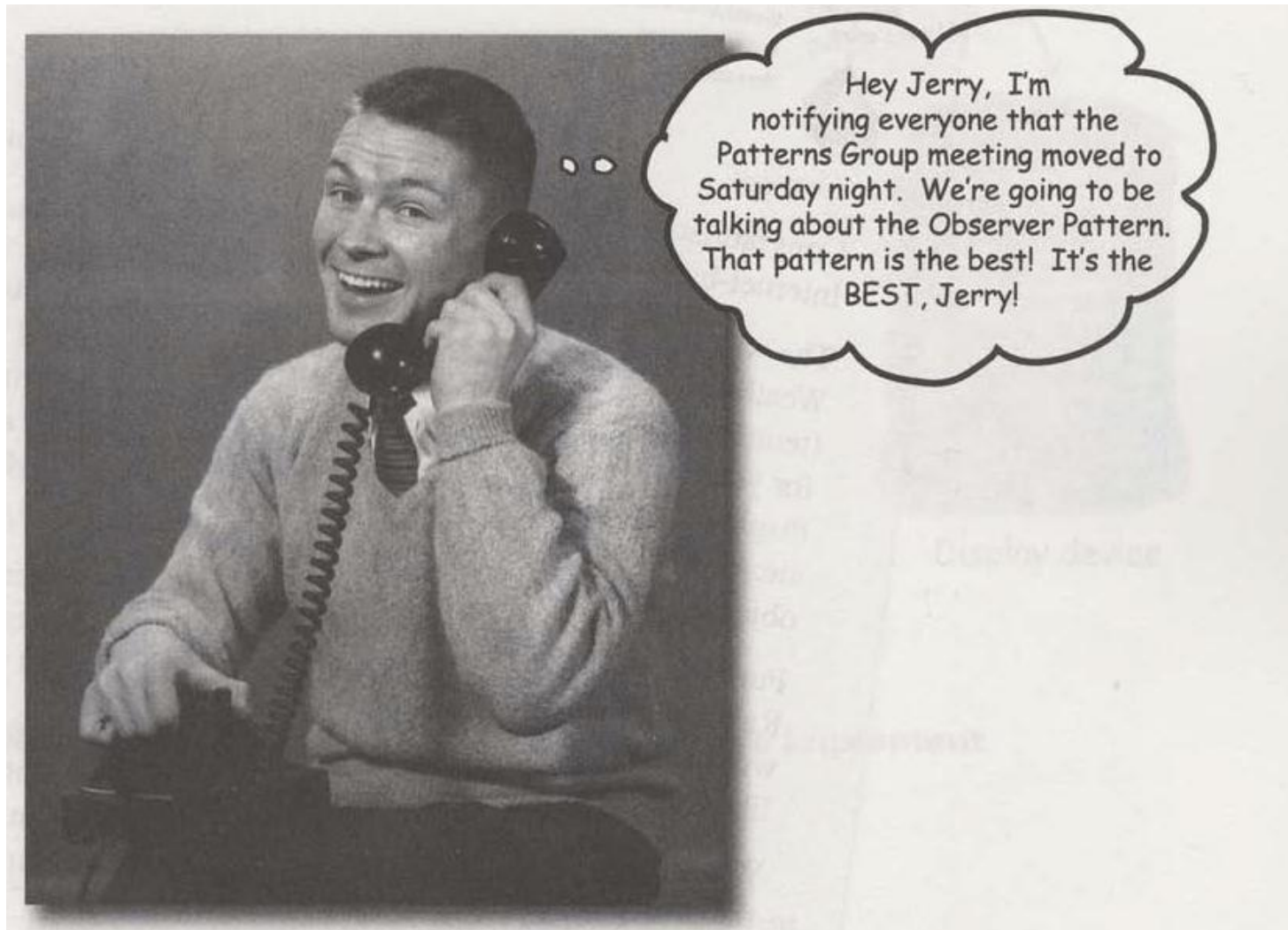
# Design Patterns

Observer Pattern

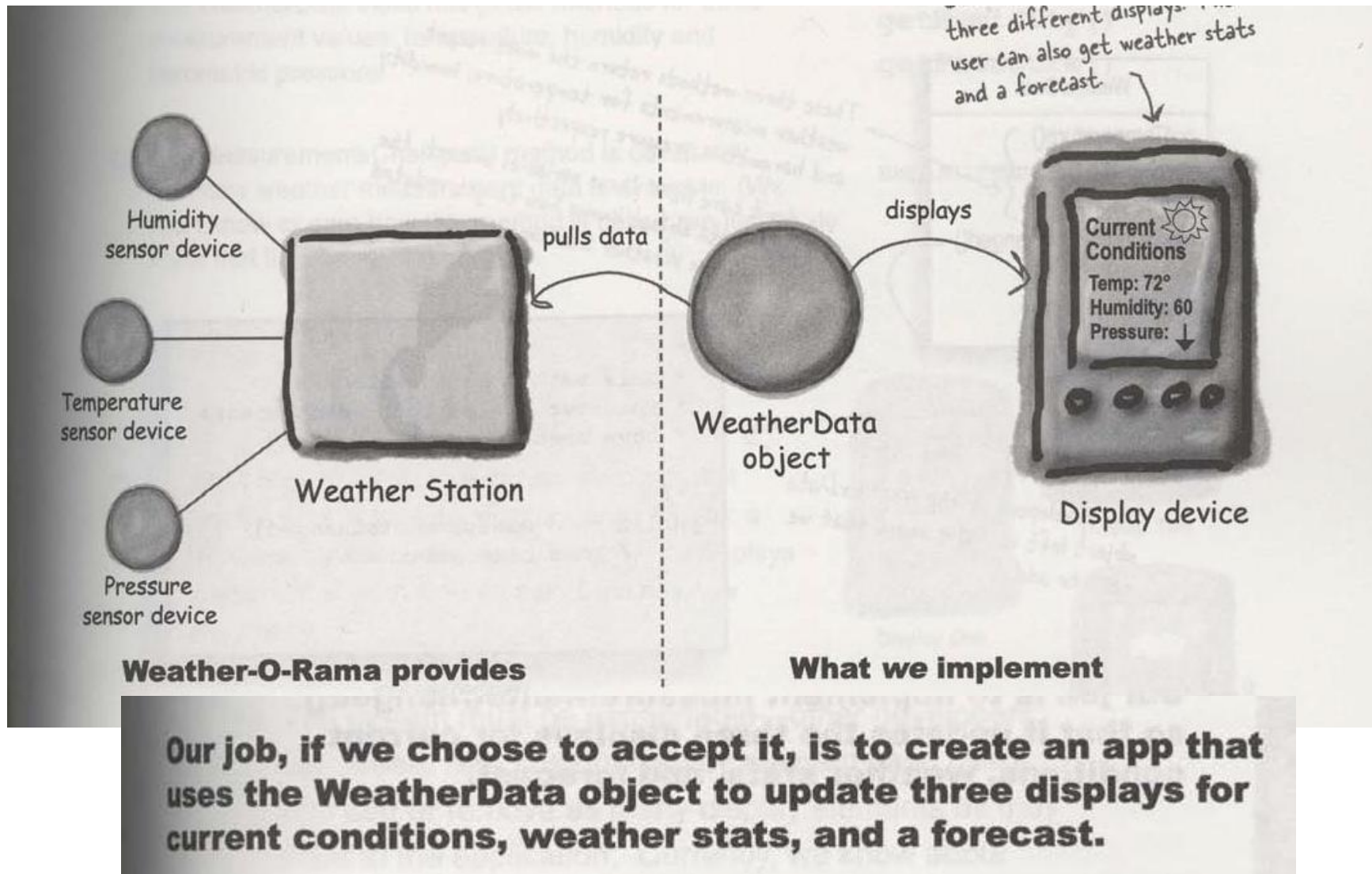
Dr, Ayman Ezzat

swe1@fcih.net

# Observer Pattern



# Weather Information machine



# Code look

```
WeatherData
{
  getTemperature()
  getHumidity()
  getPressure()
  measurementsChanged()
  // other methods
}
```



Display device

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

# Lets implement

- ⚙ 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.

```
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).

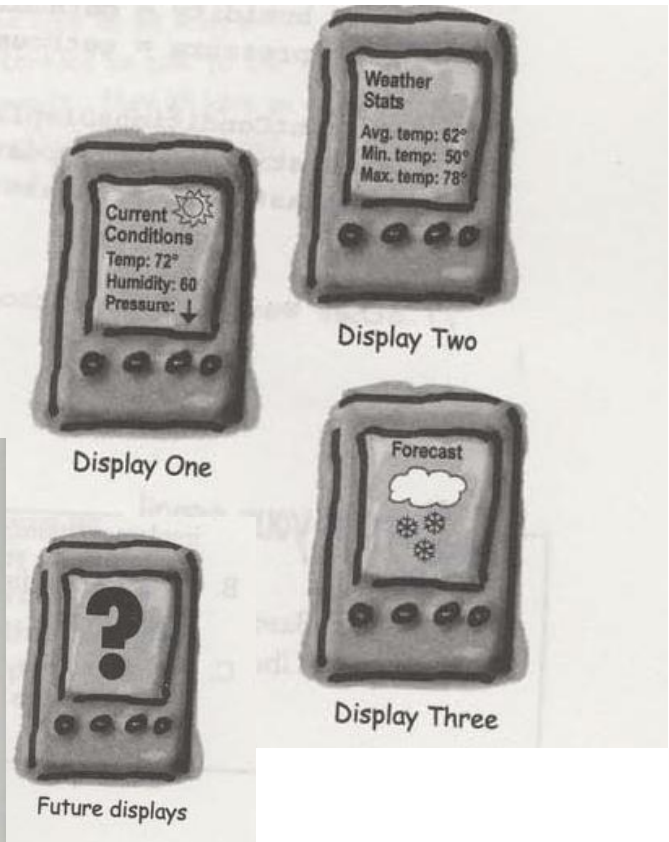
Now update the displays...

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

**Concrete Class**

**Same update**

**Varies**



# Observer Concept

Publishers + Subscribers = Observer Pattern

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

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

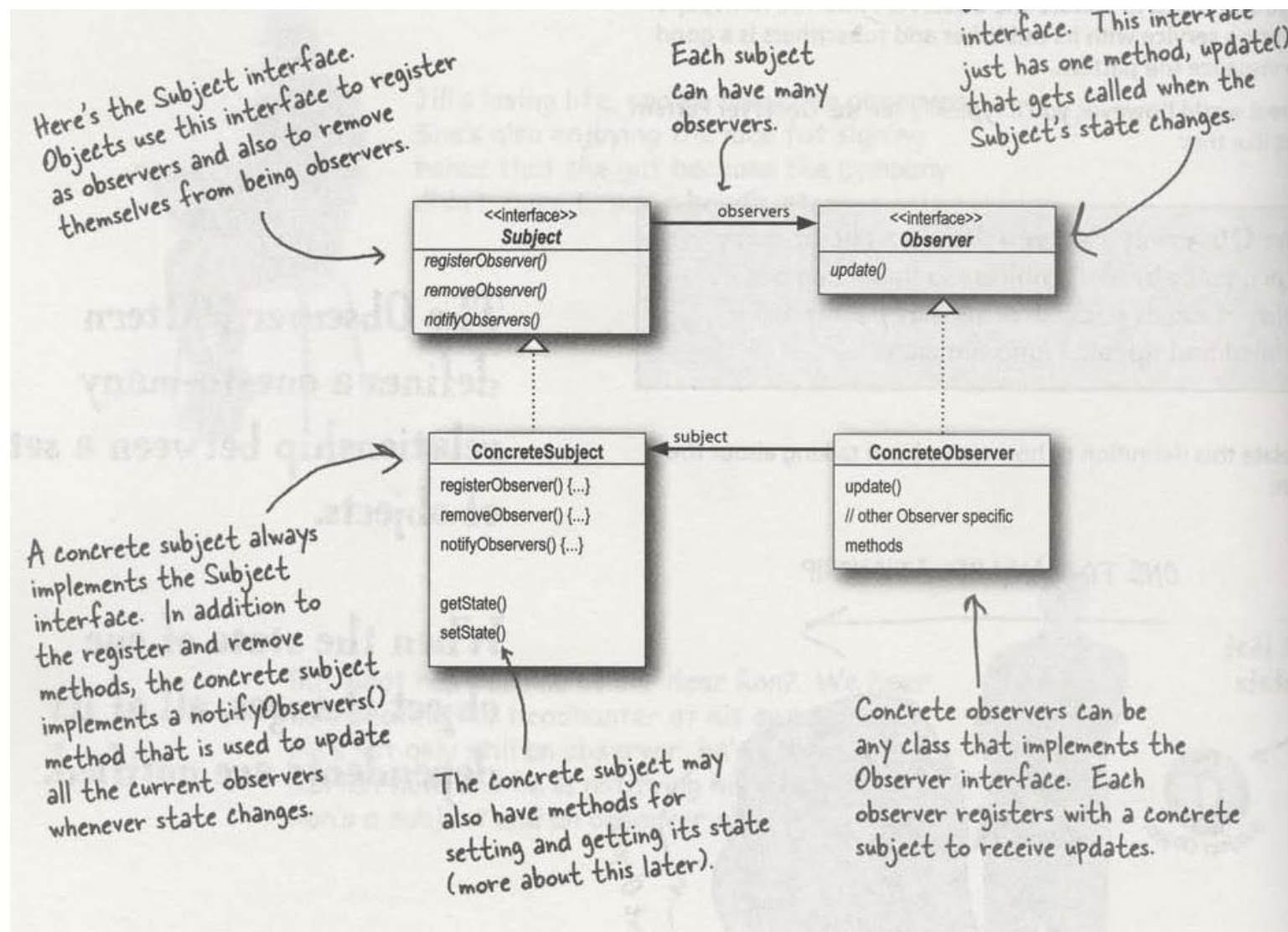
**The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



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

Observer Objects

# Observer Class Definition



# Power of loose couple

- The subject doesn't know anything about the implementation except the interface they implement.
- Add new observers anytime.
- Minimum change if we added new observers.
- Change in observer or subject will never change in each other.



## ***Design Principle***

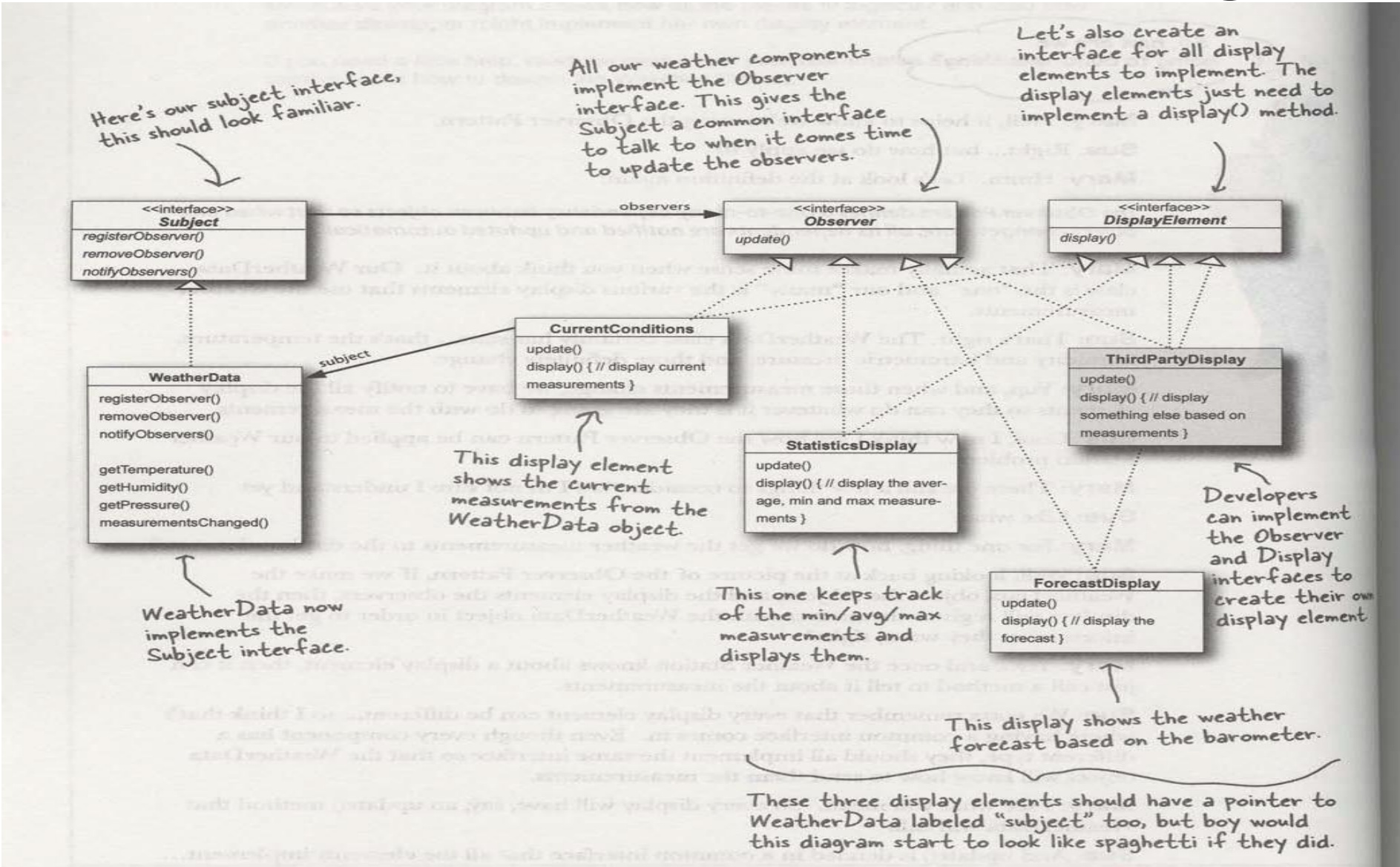
*Strive for loosely coupled designs between objects that interact.*

## **The power of Loose Coupling**

**When two objects are loosely coupled, they can interact, but have very little knowledge of each other.**

**The Observer Pattern provides an object design where subjects and observers are loosely coupled.**

# Weather Information new design



# Code for subject

```
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);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

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

    // other WeatherData methods here
}
```

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.

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

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.

Here we implement the Subject interface.

# Code for Observer

```
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");  
    }
```

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.

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.

# Main Program

```
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);
    }
}
```

don't  
o  
ad the  
ou can  
nt out  
two lines  
n it.

First, create the  
WeatherData  
object.

Create the three  
displays and  
pass them the

```
File Edit Window Help StormyWeather
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```