الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

مقرر تصميم نظم البرمجيات | كلية الهندسة المعلوماتية | المحاضرة 7

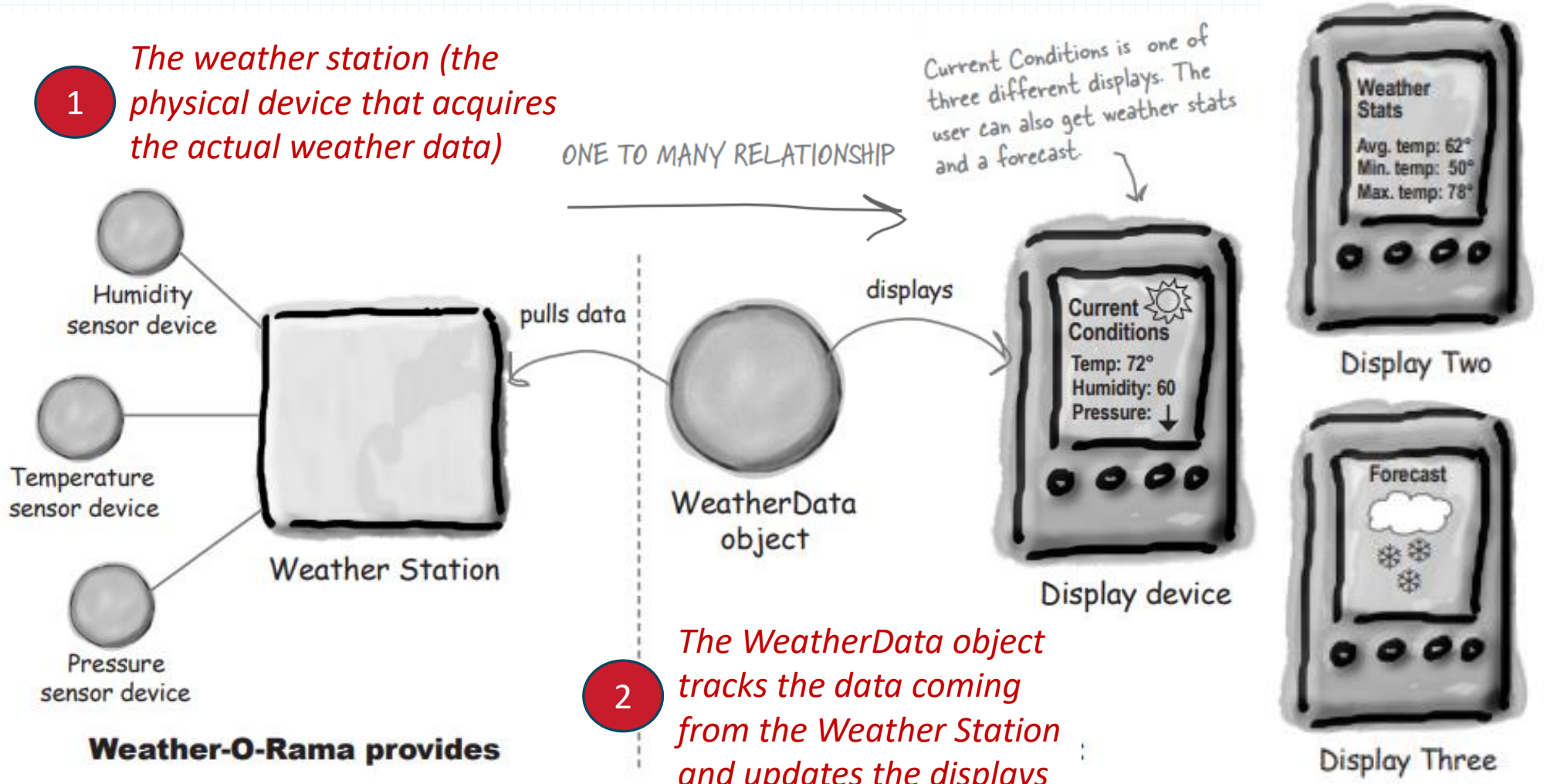
Design Patterns
The Observer Pattern

د. رياض سنبل

Sample Problem

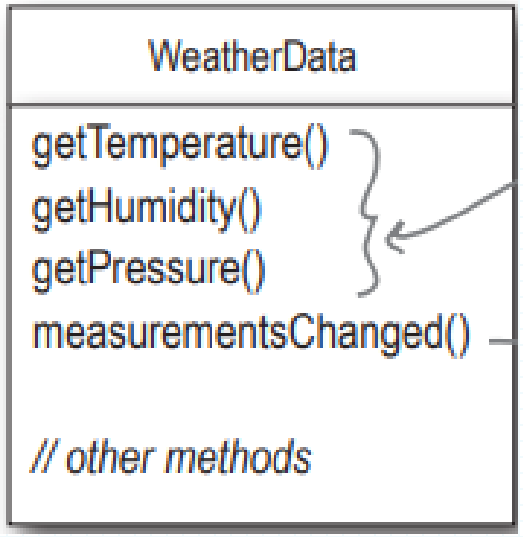
3 *the display that shows users the current weather conditions.*

1 *The weather station (the physical device that acquires the actual weather data)*



2 *The WeatherData object tracks the data coming from the Weather Station and updates the displays*

First Solution!

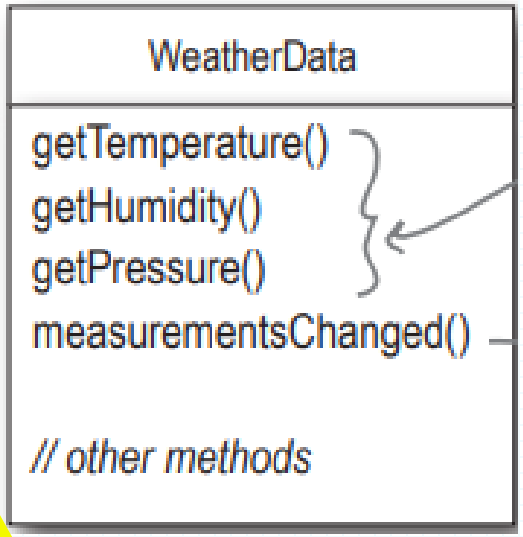


```
public void measurementsChanged() {

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

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

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

Now update the displays...

First Solution!



Display One

Display Two

Display Three

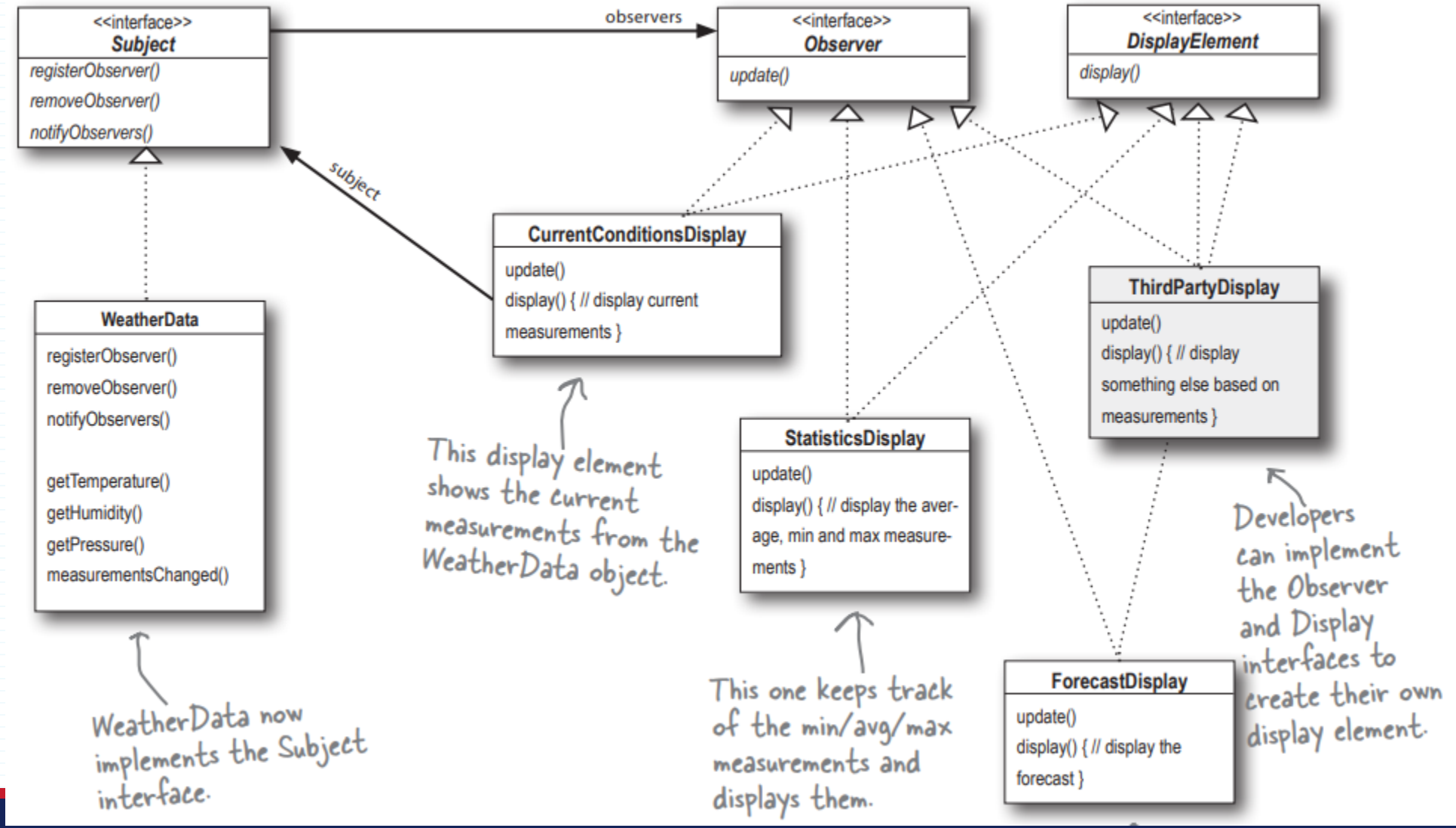
For every new display element we need to alter code.
We have no way to add (or remove) display elements at run time.

```
tsChanged() {

getTemperature();
dity = getHumidity();
pressure = getPressure();

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

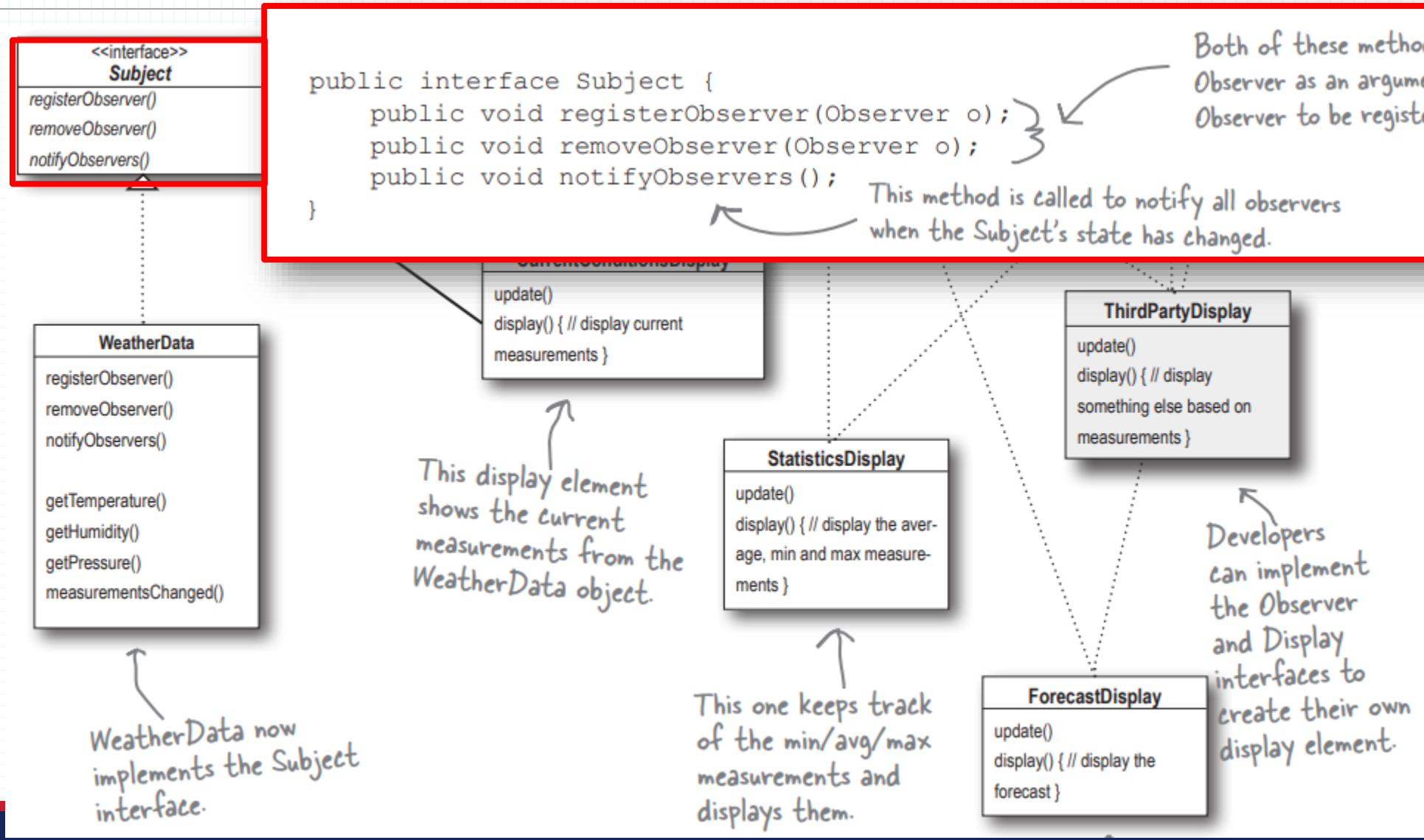
Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

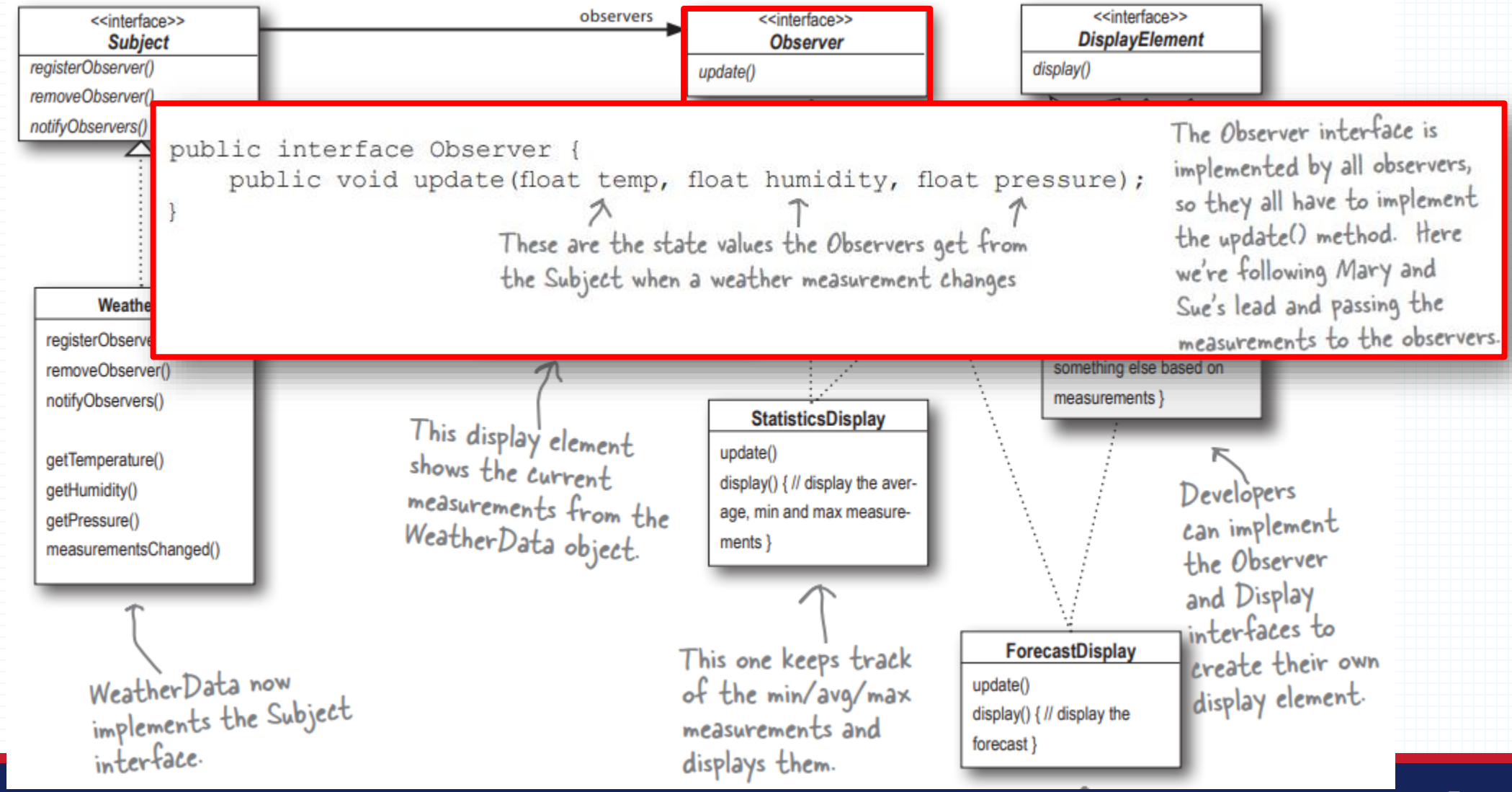
Better Solution



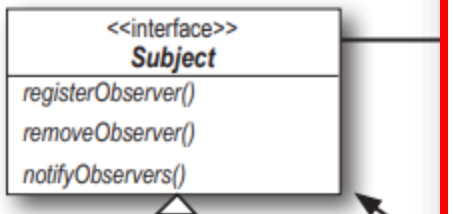
Better Solution

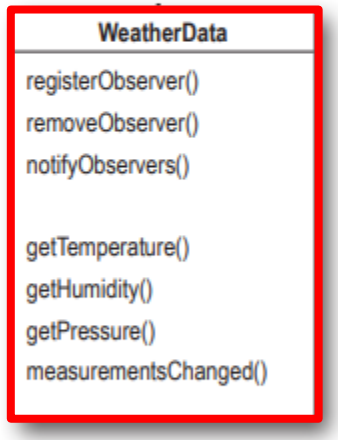


Better Solution



Better Solu





WeatherData now implements the Subject interface.

```
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

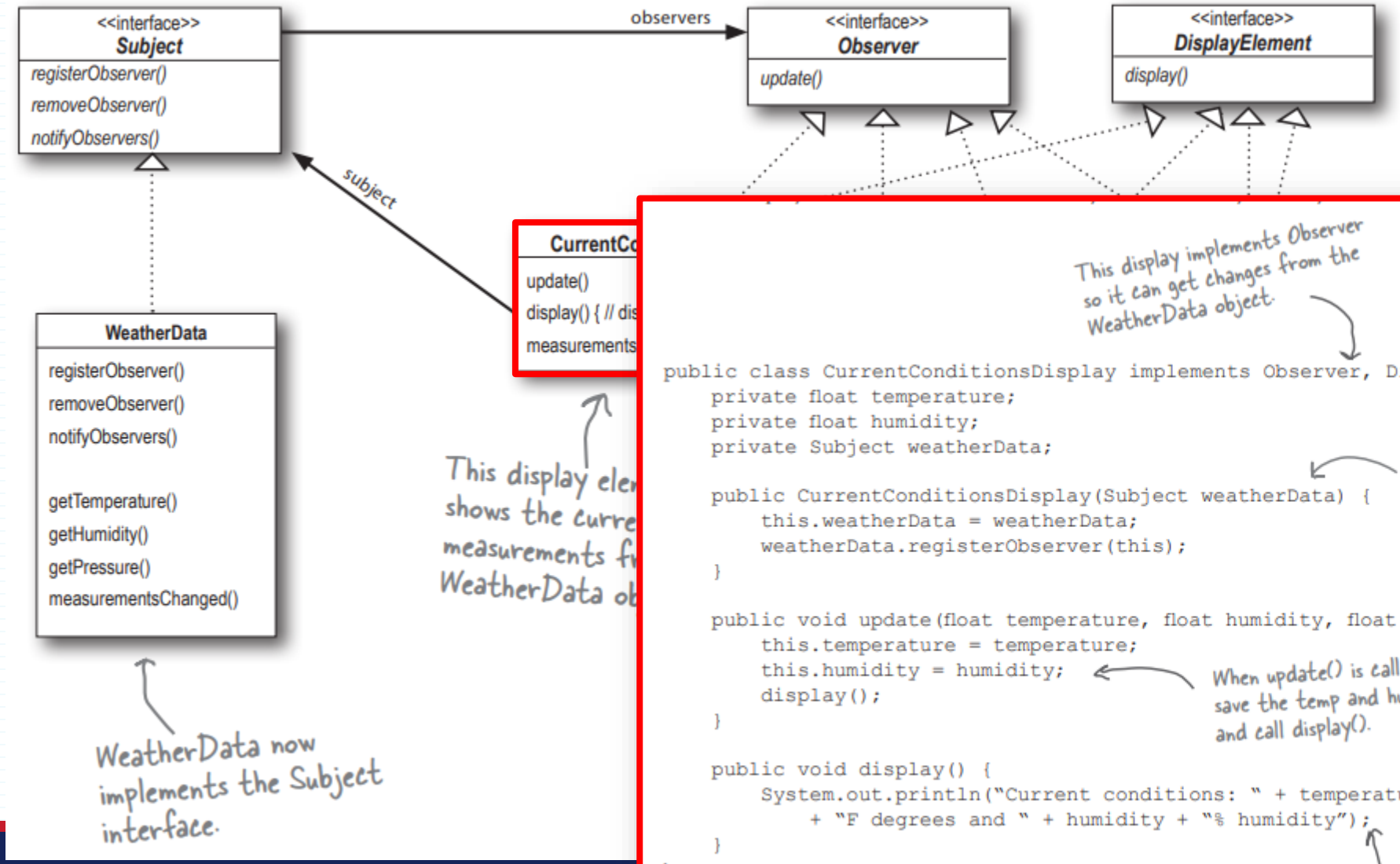
Likewise, when an observer wants to un-register, we just take it off the list

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.

Better Solution



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.

```
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.

Sample usage

```
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);
    }
}
```

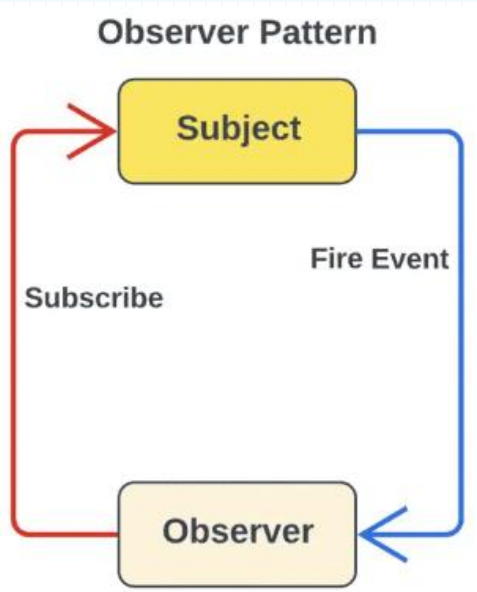
First, create the WeatherData object.

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

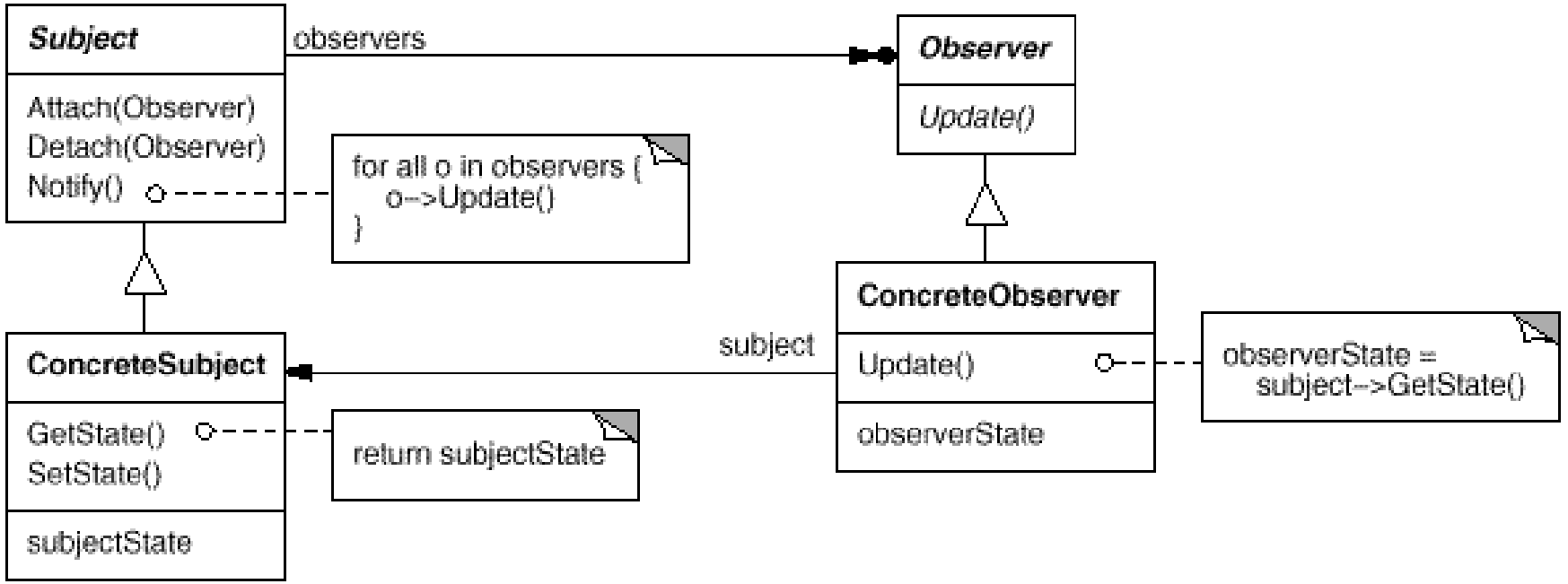
Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

So.. Observer Pattern



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



Other Examples of observer design patterns

- Social Media Platforms for example, many users can follow a particular person(subject) on a social media platform. All the followers will be updated if the subject updates his/her profile. The users can follow and unfollow the subject anytime they want.
- Newsletters or magazine subscription
- Journalists providing news to the media
- E-commerce websites notify customers of the availability of specific products
- More?