CS525 Advanced Software Development

Lesson 3 – The Observer Pattern

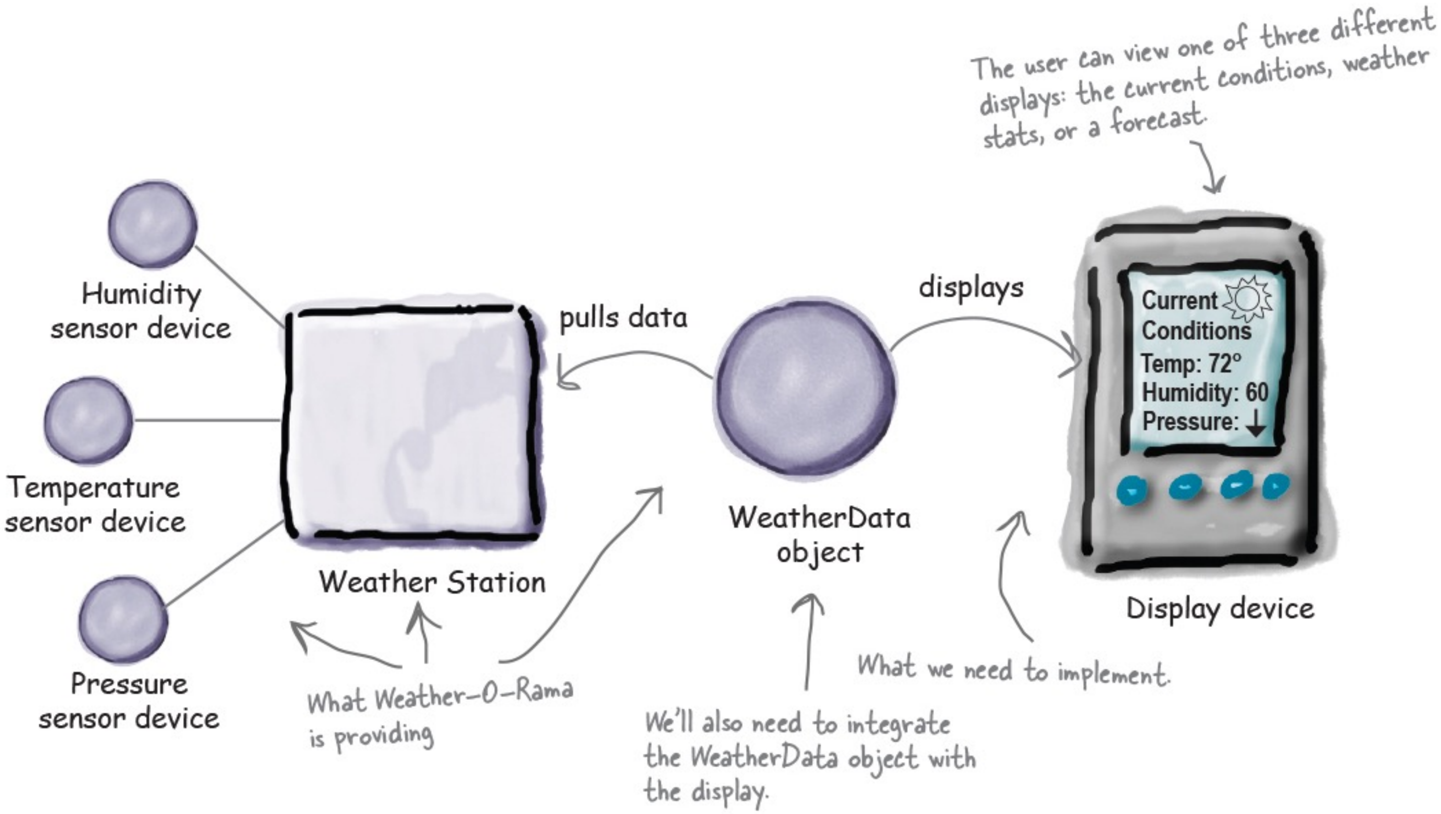
Design Patterns
Elements of Reusable Object-Oriented Software

Payman Salek, M.S.
March 2022

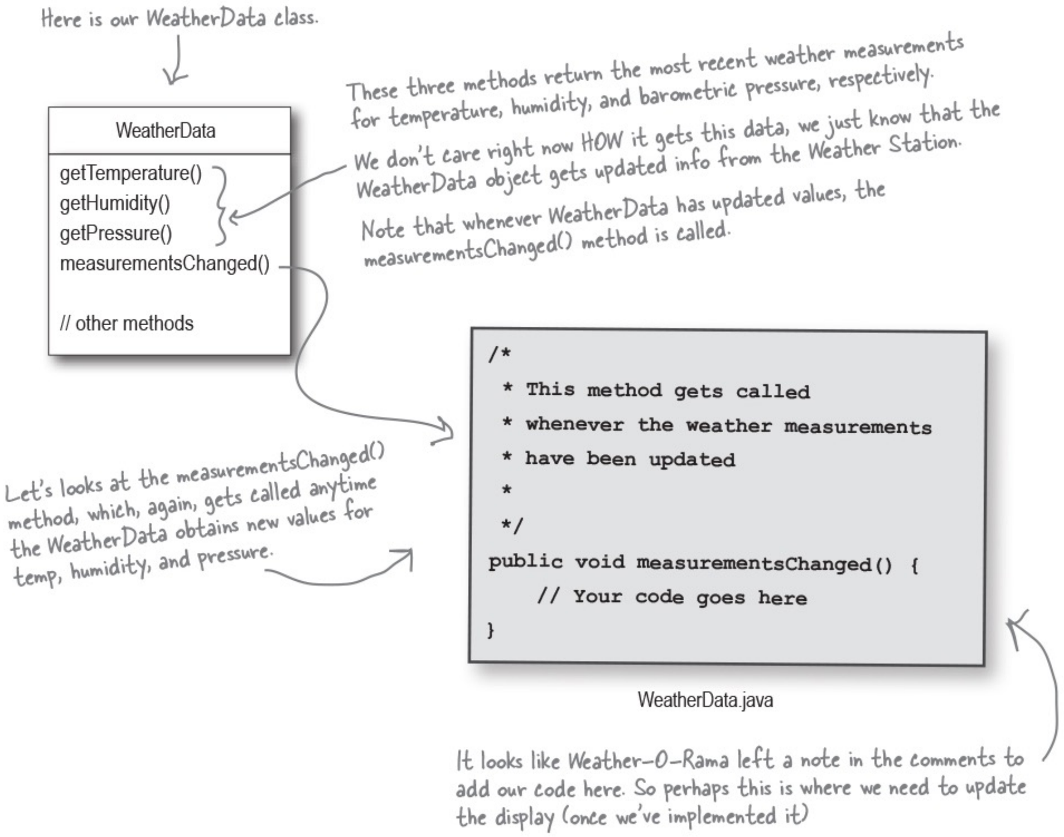
Introduction

We’ve got a pattern that keeps your objects in the know when something they care about happens. It’s the Observer Pattern. It is one of the most commonly used design patterns, and it’s incredibly useful. We’re going to look at all kinds of interesting aspects of Observer, like its one-to-many relationships and loose coupling.

Setting the stage (Weather-O-Rama)


The user can view one of three different displays: the current conditions, weather stats, or a forecast.
Humidity sensor device
Temperature sensor device
Pressure sensor device
Weather Station
pulls data
WeatherData object
displays
Current Conditions
Temp: 72°
Humidity: 60
Pressure: ↓
Display device
What Weather-O-Rama is providing
We'll also need to integrate the WeatherData object with the display.
What we need to implement.

WeatherData Class



Here is our WeatherData class.

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

We don't care right now HOW it gets this data, we just know that the WeatherData object gets updated info from the Weather Station.

Note that whenever WeatherData has updated values, the measurementsChanged() method is called.

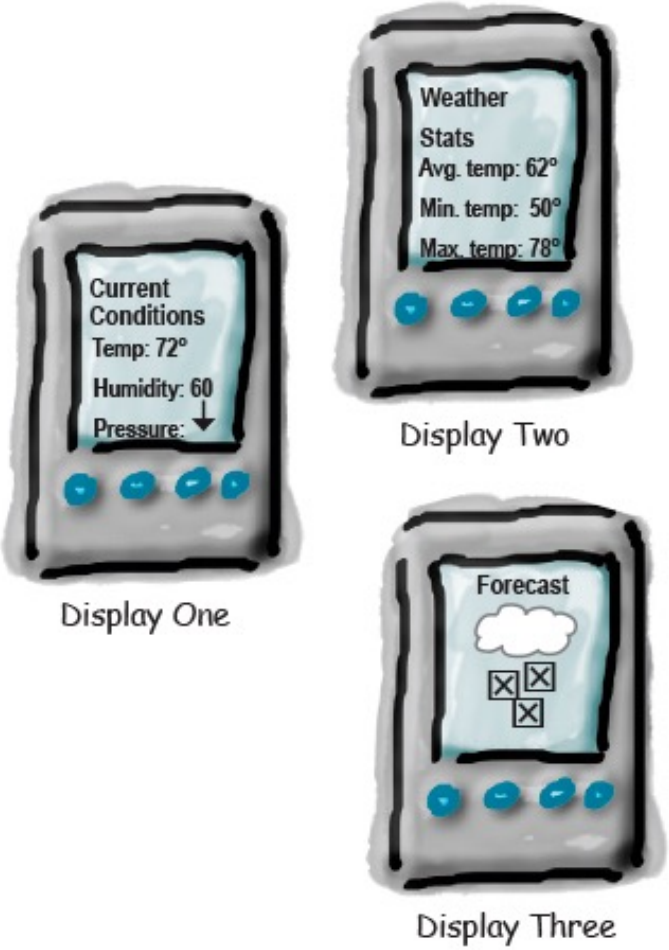
Let's looks at the measurementsChanged() method, which, again, gets called anytime the WeatherData obtains new values for temp, humidity, and pressure.

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

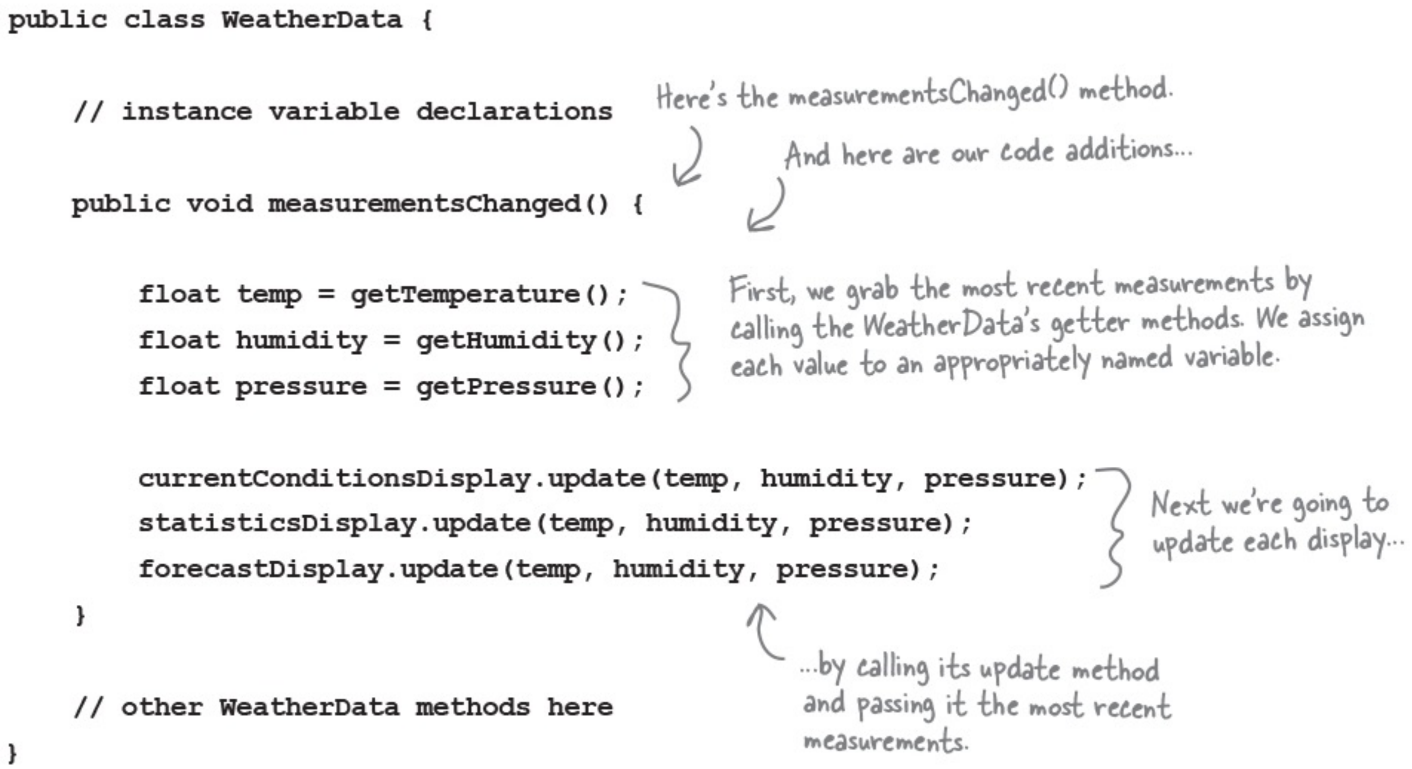
WeatherData.java

It looks like Weather-O-Rama left a note in the comments to add our code here. So perhaps this is where we need to update the display (once we've implemented it)

Our goal: Update views



First Attempt



```
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
}
```

Here's the measurementsChanged() method.

And here are our code additions...

First, we grab the most recent measurements by calling the WeatherData's getter methods. We assign each value to an appropriately named variable.

Next we're going to update each display...

...by calling its update method and passing it the most recent measurements.

What’s wrong?

What’s wrong?

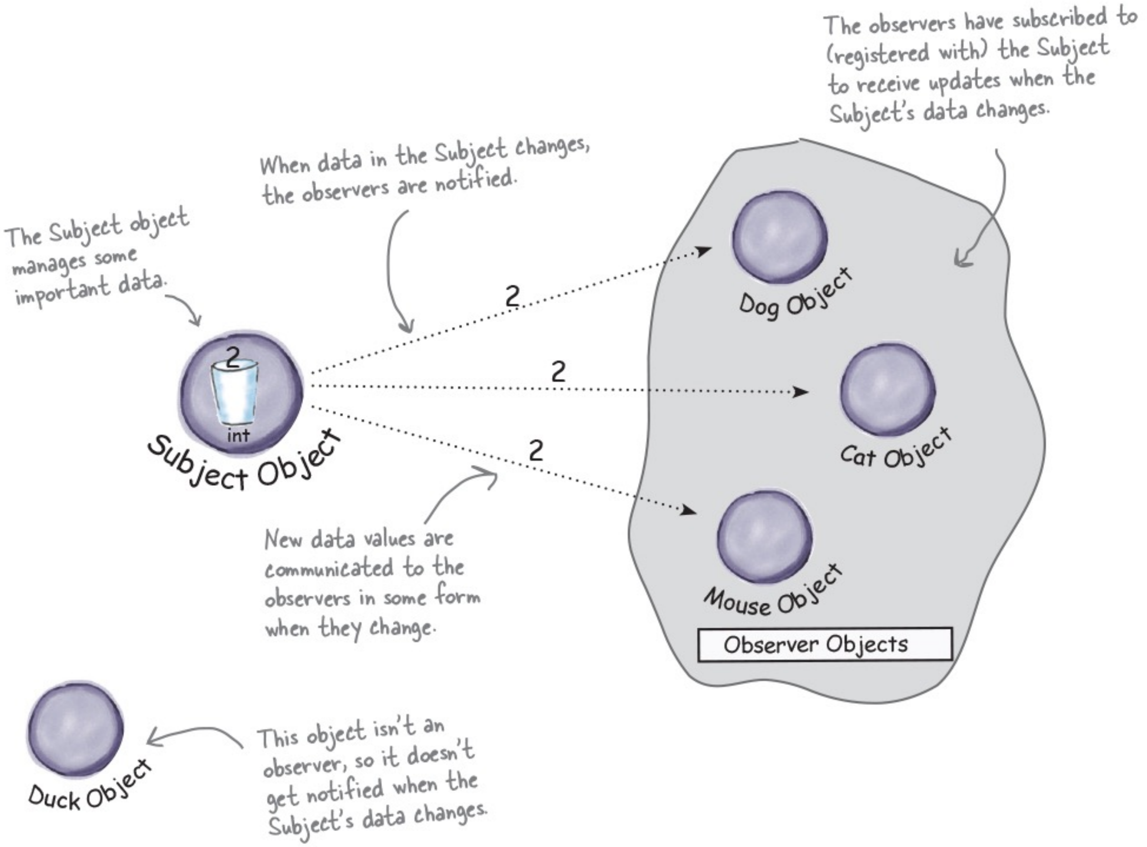
Violating open-closed

What’s wrong?

Violating open-closed

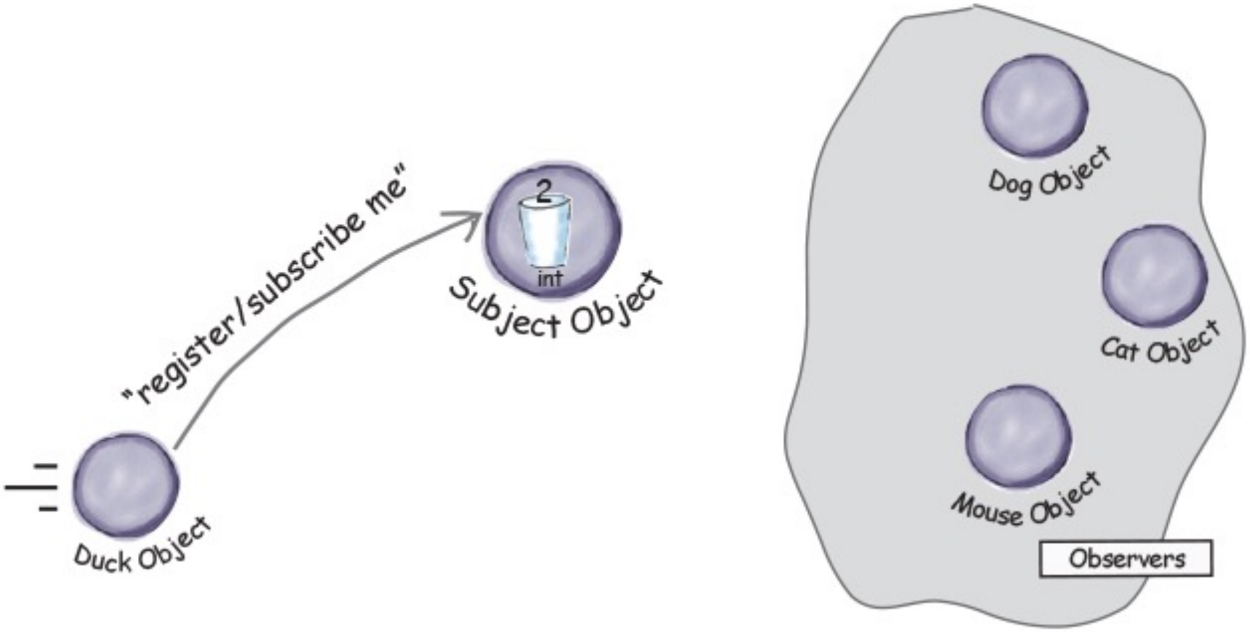
Impossible to add displays at runtime

Solution: The Observer Pattern

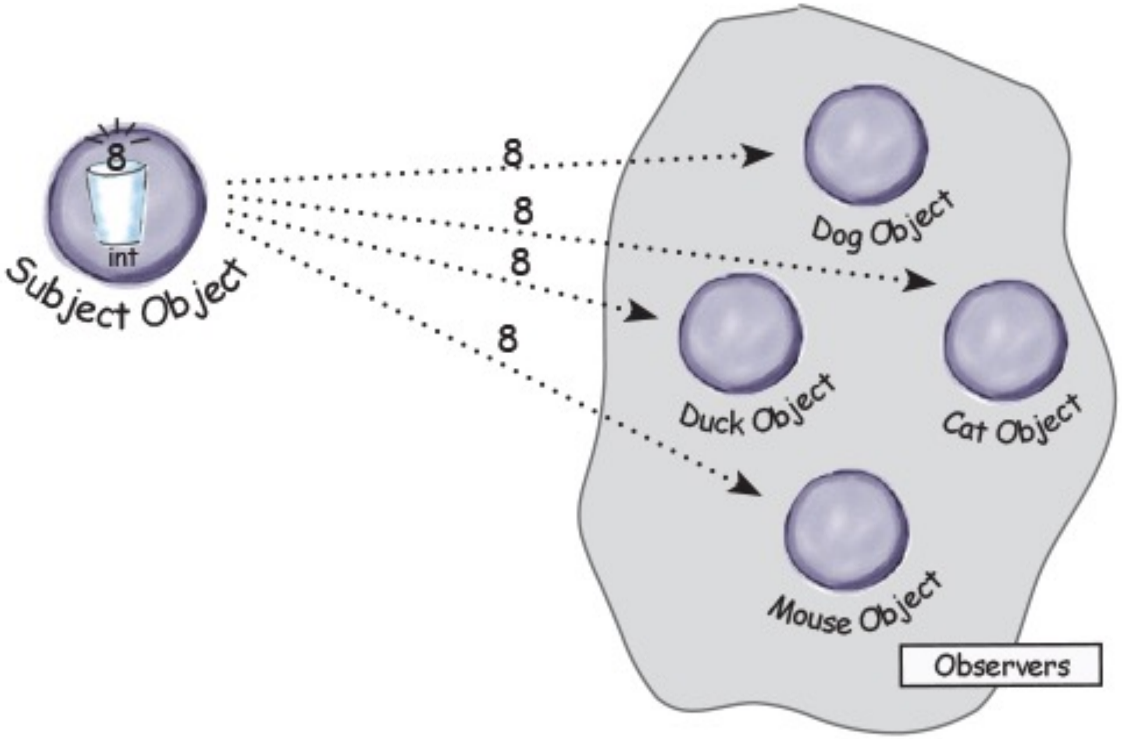


The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.
When data in the Subject changes, the observers are notified.
The Subject object manages some important data.
2
int
Subject Object
2
2
2
Dog Object
Cat Object
Mouse Object
Observer Objects
New data values are communicated to the observers in some form when they change.
Duck Object
This object isn't an observer, so it doesn't get notified when the Subject's data changes.

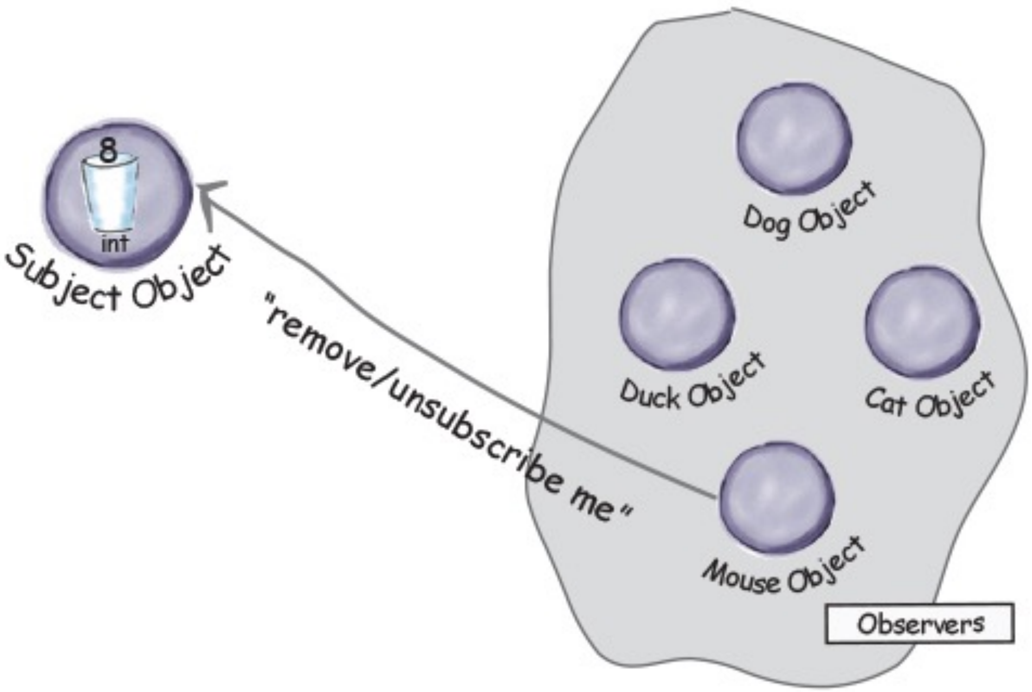
Registration Process



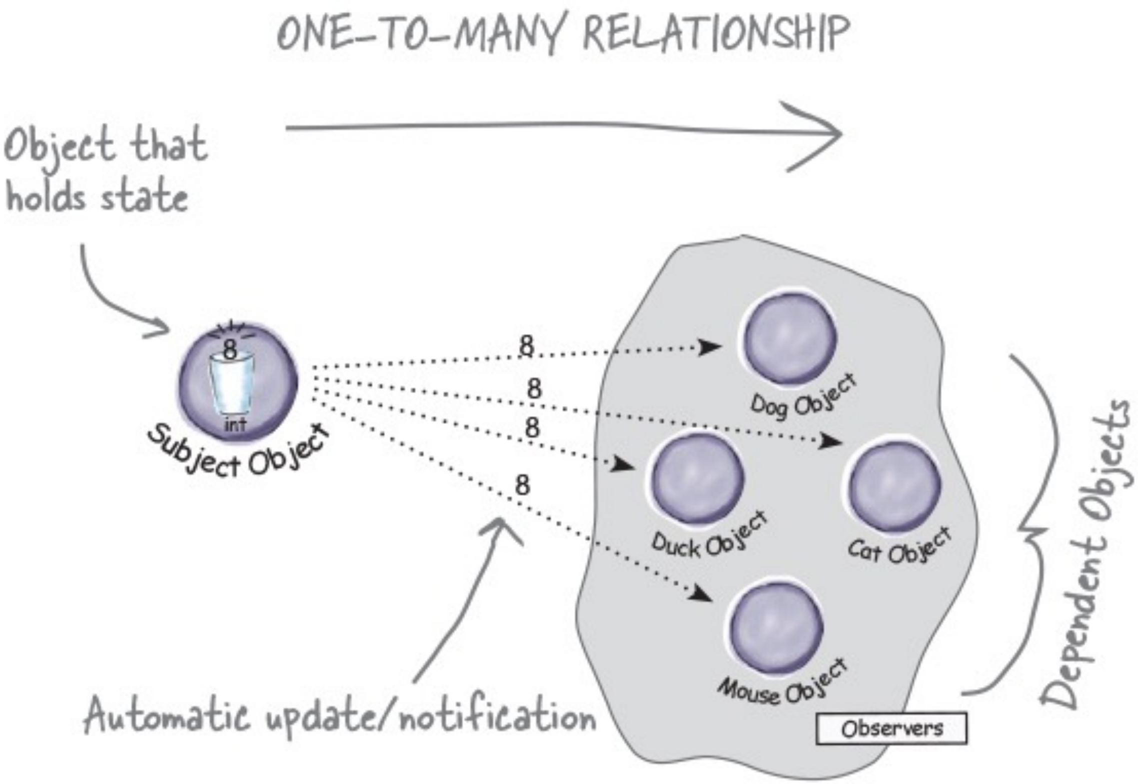
Registration Process



Un-Subscribe Process



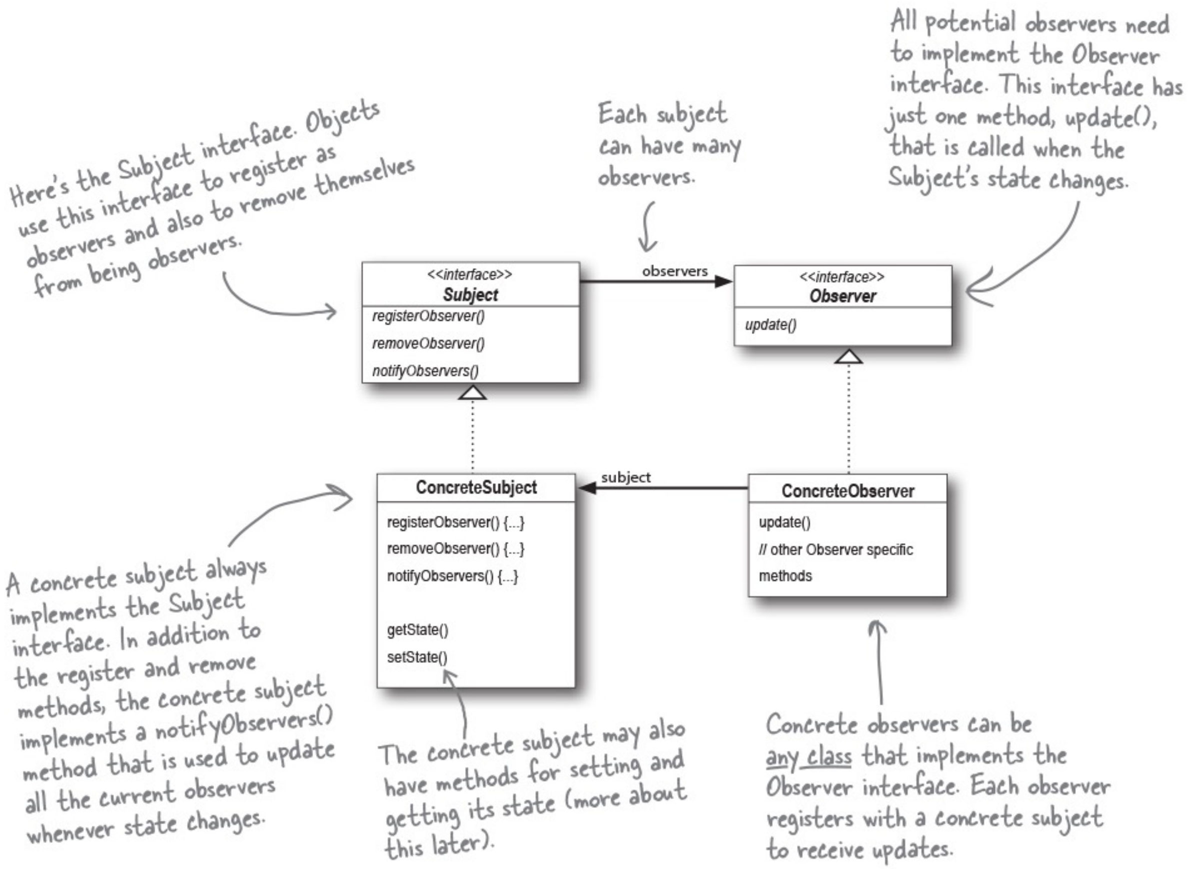
Summary


ONE-TO-MANY RELATIONSHIP
Object that holds state
8
int
Subject Object
8
8
8
8
Dog Object
Duck Object
Cat Object
Mouse Object
Observers
Dependent Objects
Automatic update/notification

Observer: Explained

The subject and observers define the one-to-many relationship. We have one subject, who notifies many observers when something in the subject changes. The observers are dependent on the subject—when the subject’s state changes, the observers are notified.

Class Diagram



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 has just one method, update(), that is 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.

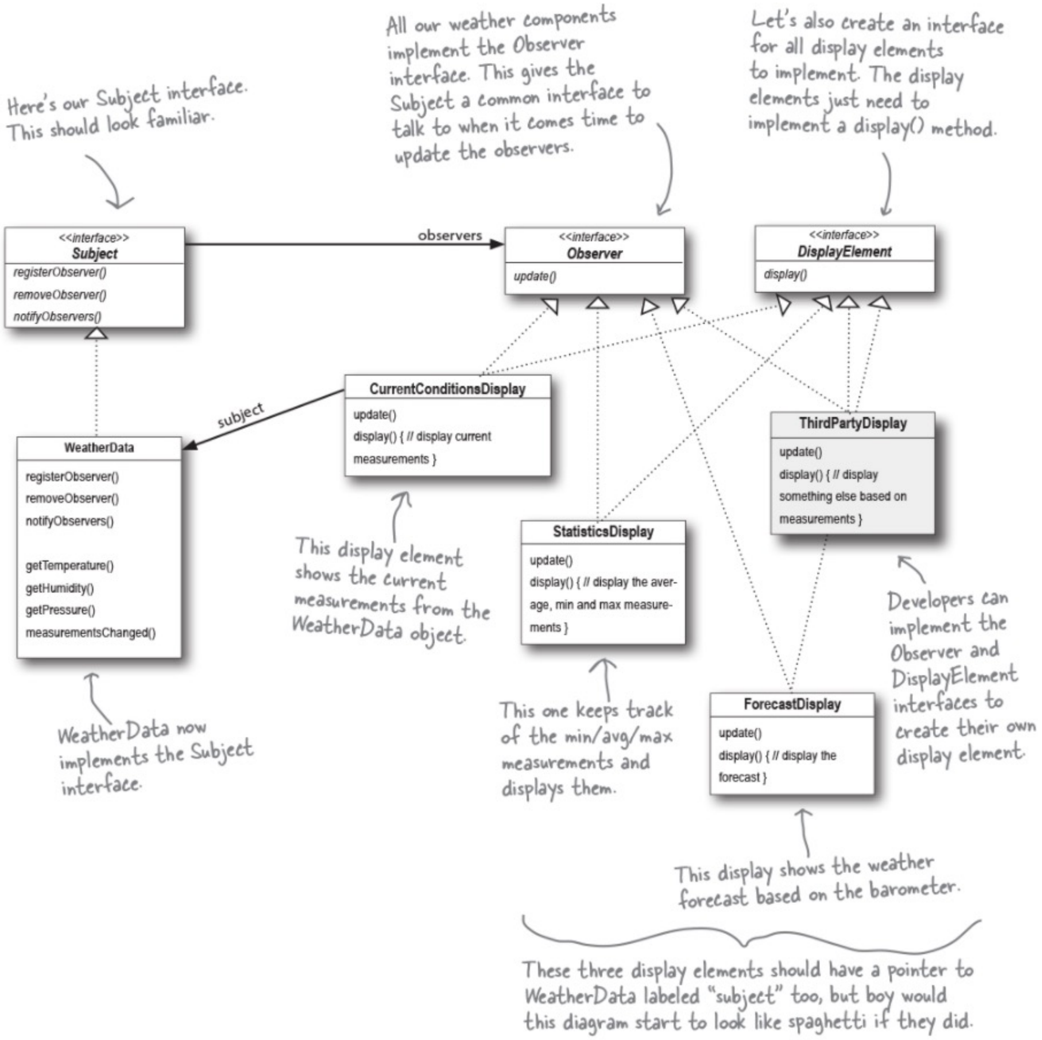
The concrete subject may also have methods for setting and getting its state (more about this later).

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

Loose Coupling

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

Weather Station Redesign



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.

WeatherData now implements the Subject interface.

This display element shows the current measurements from the WeatherData object.

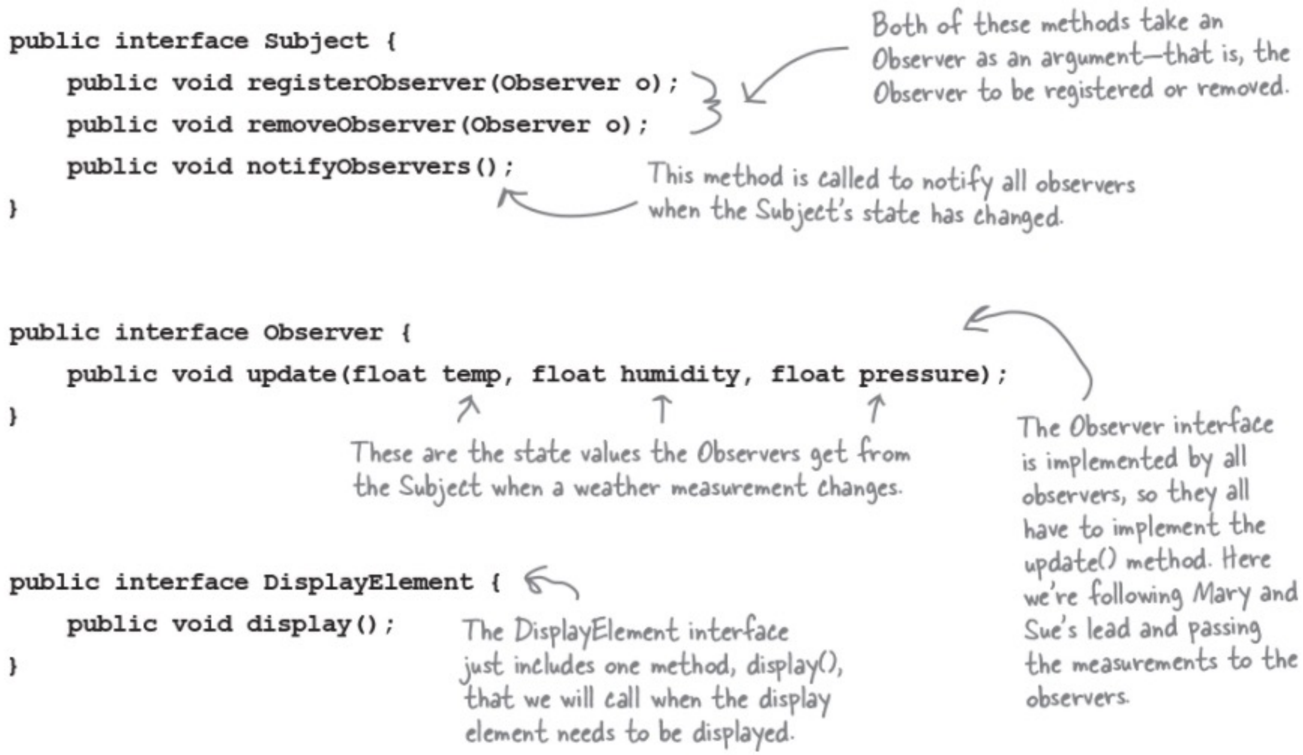
This one keeps track of the min/avg/max measurements and displays them.

Developers can implement the Observer and DisplayElement interfaces to create their own display element.

This display shows the weather forecast based on the barometer.

These three display elements should have a pointer to WeatherData labeled "subject" too, but boy would this diagram start to look like spaghetti if they did.

Weather Station Code



```
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.

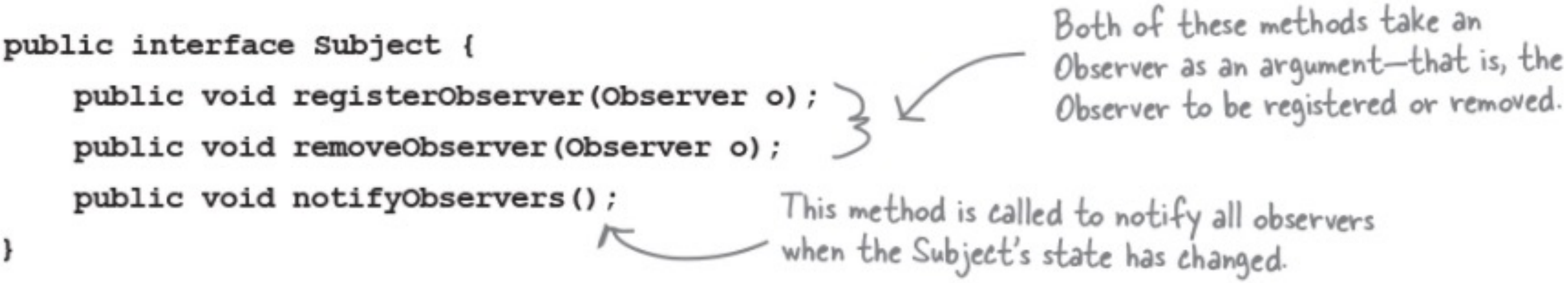
```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

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.



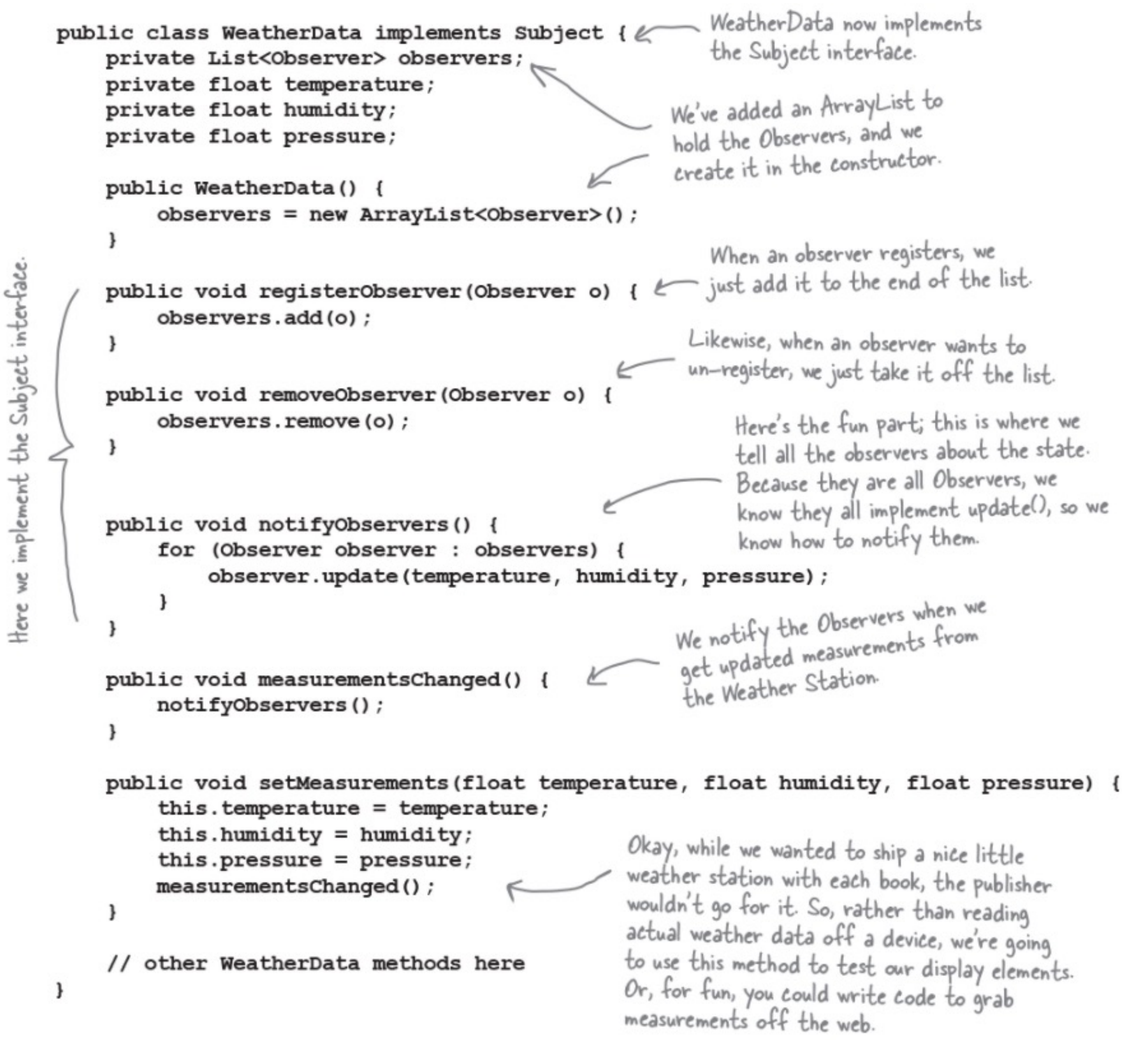
```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

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.

Practice: Implement Subject

Solution



```
public class WeatherData implements Subject {
    private List<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }

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

    public void removeObserver(Observer o) {
        observers.remove(o);
    }


    public void notifyObservers() {
        for (Observer observer : observers) {
            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.

Here we implement the Subject interface.

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

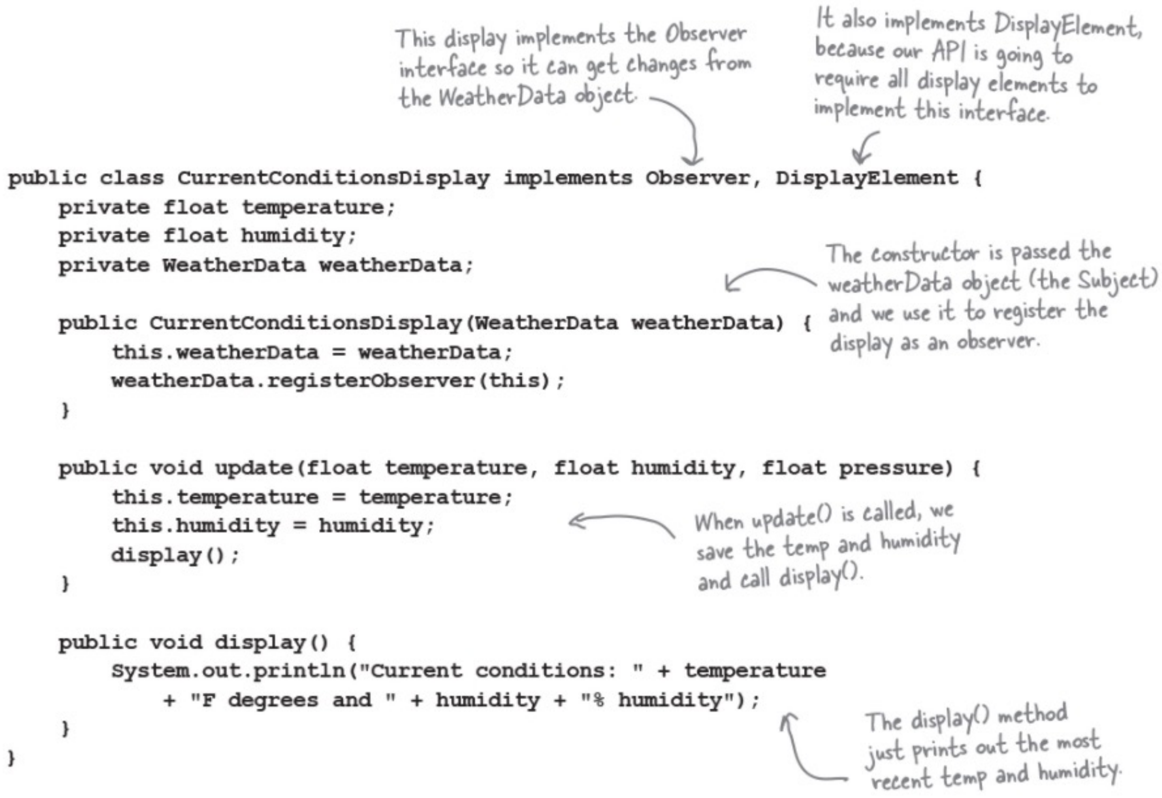
Likewise, when an observer wants to un-register, we just take it off the list.

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.

Solution



This display implements the Observer interface 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 WeatherData weatherData;

    public CurrentConditionsDisplay(WeatherData 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.

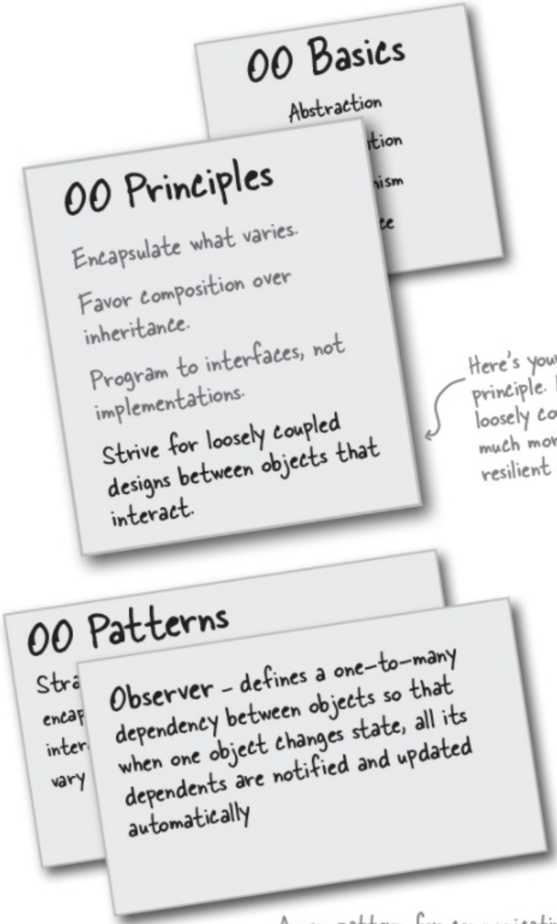
When update() is called, we save the temp and humidity and call display().

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

The Observer Pattern

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

Summary



OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

OO Patterns

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

Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.

A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern—just wait until we talk about MVC!