



Observer



¡Felicidades!

Su grupo acaba de ganar el
contrato para construir la Estación
de Monitoreo del Clima Weather-O-
Rama, Inc.



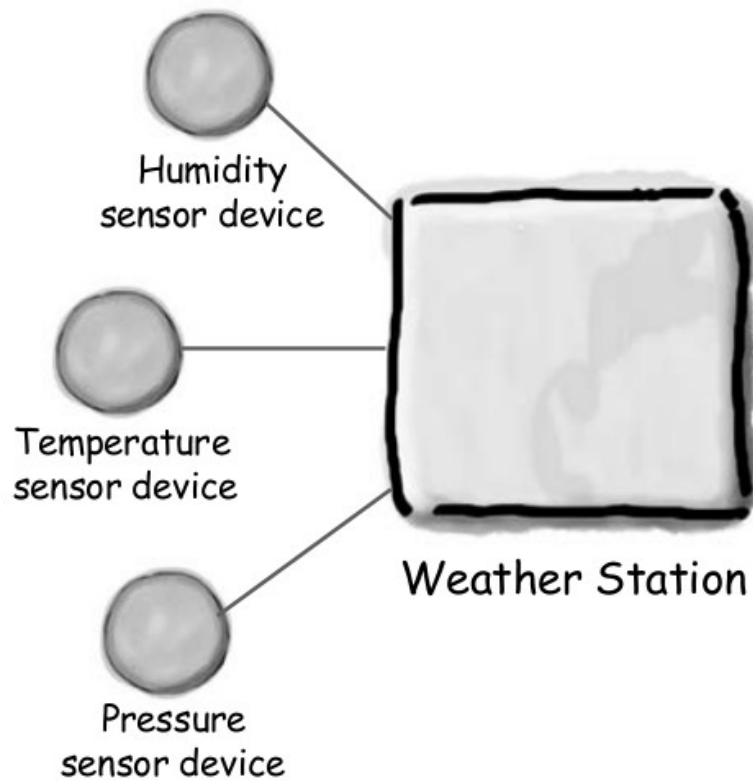
- Debe ser posible visualizar en tres pantallas diferentes: las condiciones actuales del clima, estadísticas meteorológicas y un pronóstico.
- Todo debe ser actualizado en tiempo real.



Los tres participantes en el sistema son:

- la **estación del clima**: el dispositivo físico que adquiere los datos meteorológicos reales.
- el objeto “**DatosMeteorológicos**”: realiza un seguimiento de datos procedentes de la estación Meteorológica y actualiza las pantallas.
- la **pantalla**: muestra a los usuarios las condiciones actuales del clima.



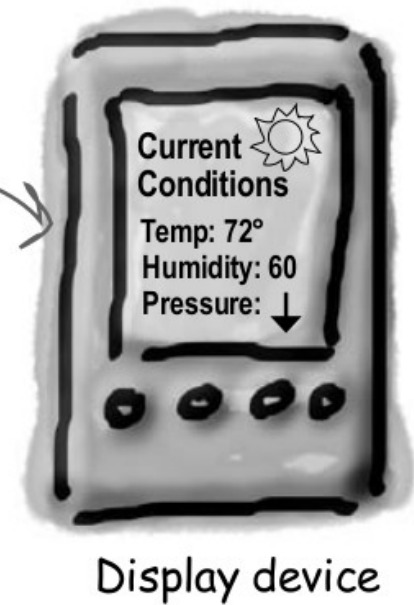


Weather-O-Rama provides

pulls data

WeatherData
object

displays

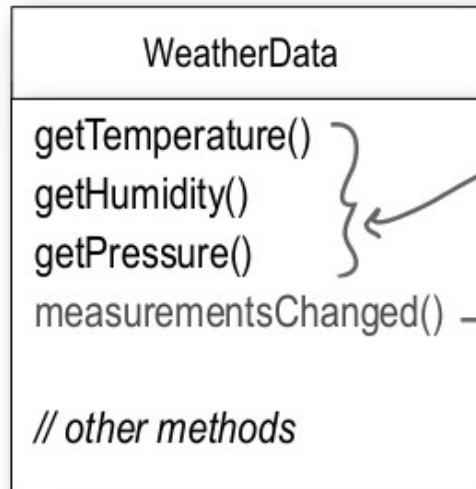


What we implement



- Nuestro trabajo, si decidimos aceptarlo, es crear una aplicación que use el objeto “DatosMeteorológicos” para actualizar tres pantallas para las condiciones actuales, las estadísticas meteorológicas y un pronóstico.





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.

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

WeatherData.java



- Su trabajo es implementar `measurementsChanged()` para que actualice las tres pantallas para las condiciones actuales, las estadísticas meteorológicas y el pronóstico.

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



Display device



- El método `measurementsChanged()` es llamado en cualquier momento en el que existan nuevos valores de los datos: temperatura, humedad, presión barométrica.



Más requerimientos

- El sistema debe ser expandible: otros desarrolladores pueden crear nuevos elementos de visualización personalizados y los usuarios pueden agregar o eliminar tantos elementos de visualización como deseen a la aplicación. Actualmente, solo conocemos los tres tipos de visualización iniciales (condiciones actuales, estadísticas y pronóstico).





Display One



Display Two



Display Three



Future displays



- ¿Cómo implementarías el método `measurementsChanged()` para que sea posible utilizarlo para actualizar la información correspondiente a las tres pantallas?



Primer diseño



```
public class WeatherData {
```

```
    // instance variable declarations
```

```
    public void measurementsChanged() {
```

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

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

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

Now update the displays..

```
    }
```

```
    // other WeatherData methods here
```

```
}
```

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





¿Qué está mal en la anterior
implementación?



¿Qué está mal en la anterior implementación?

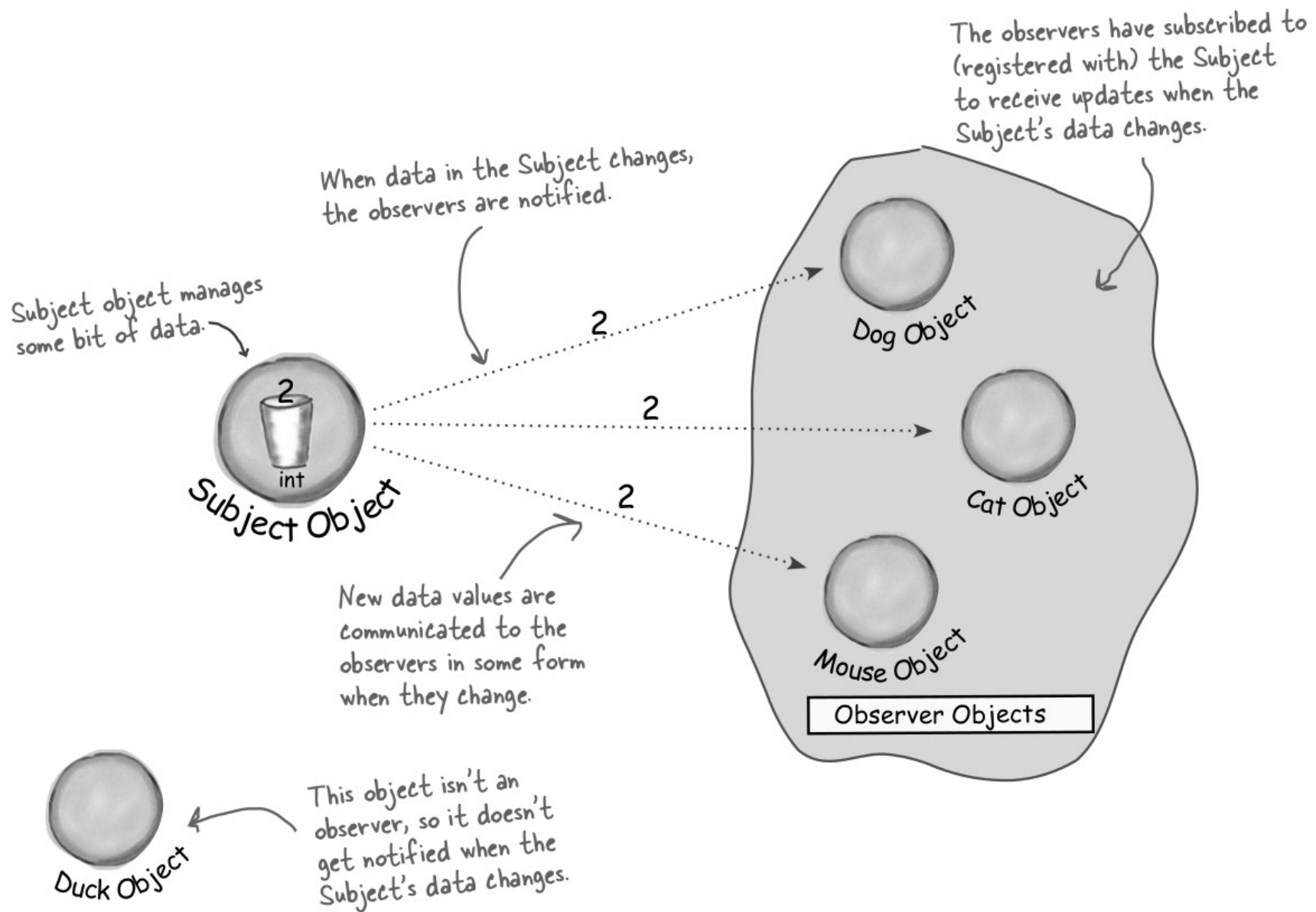
- Falta encapsular lo que cambiará constantemente.
- No existe una manera de agregar o eliminar los elementos que se visualizarán sin necesidad de modificar el código.
- Al menos parece que estamos usando una interfaz común para hablar con los elementos de la pantalla ... todos tienen un método `update()` que toma los valores de temperatura, humedad y presión.

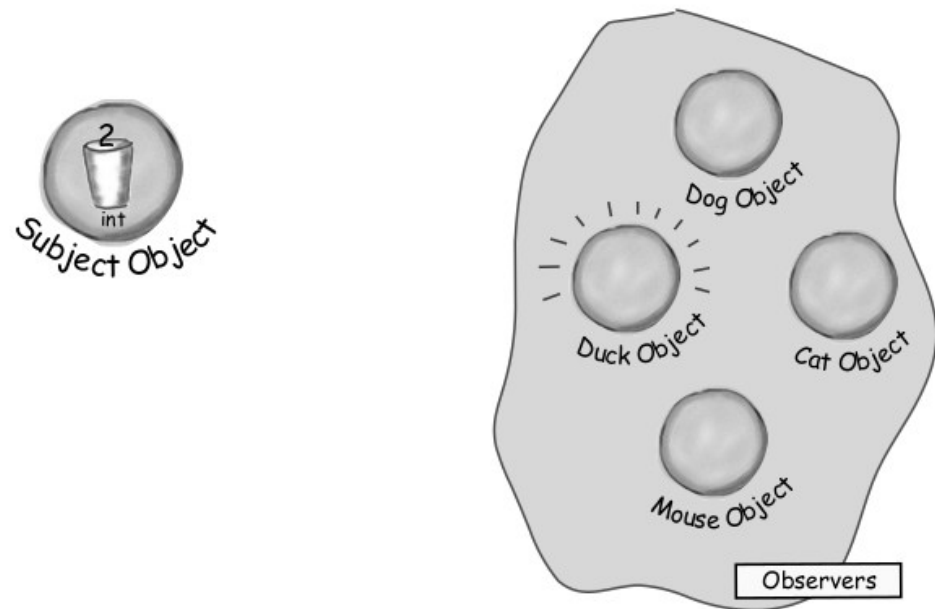
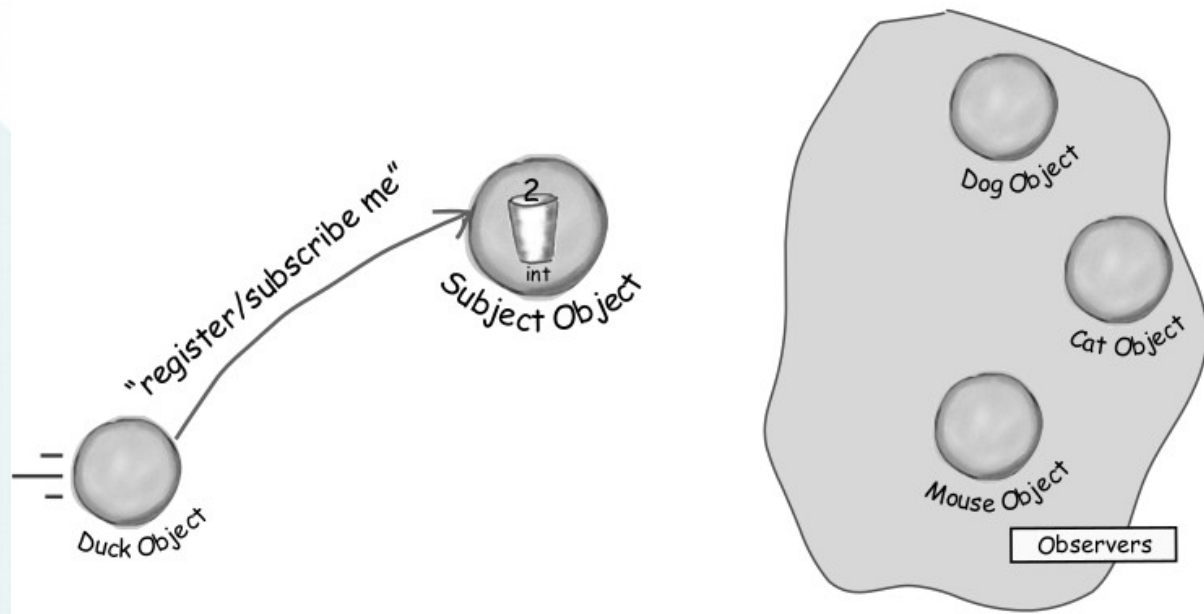


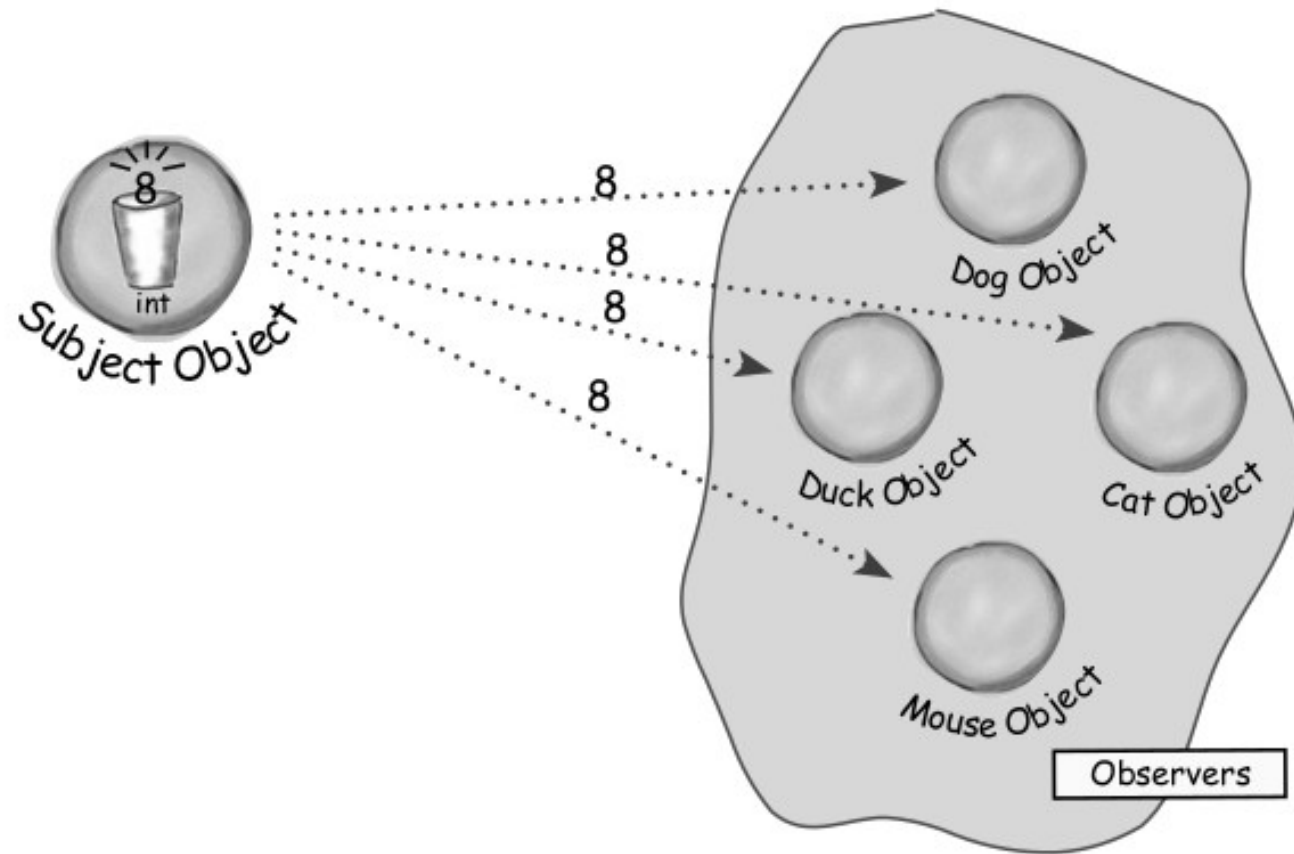
Conozcamos al patron Observer

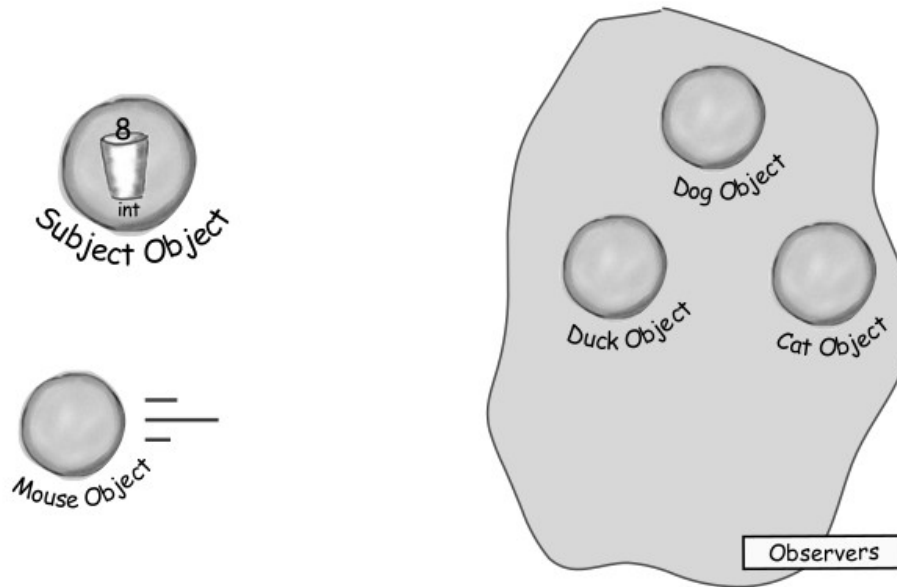
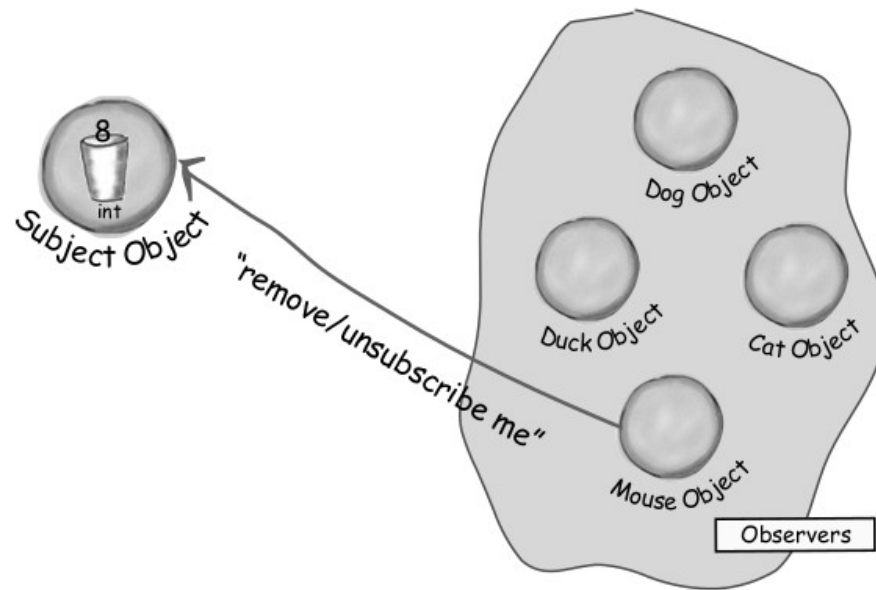
Define una dependencia de “uno a muchos” entre objetos de tal forma que cuando un objeto cambia de estado, todos sus dependientes son notificados y se actualizan automáticamente.

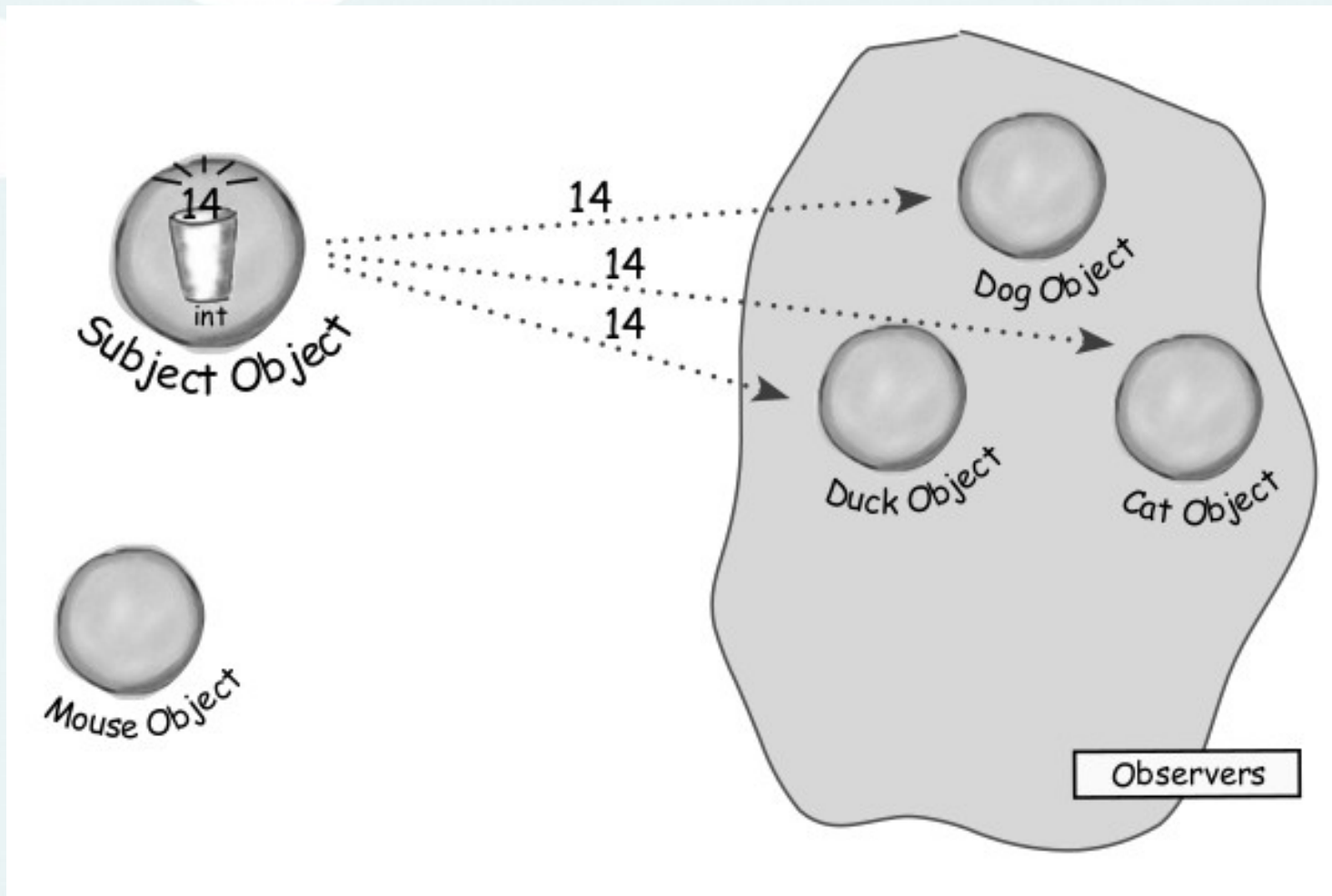












ONE TO MANY RELATIONSHIP

Object that
holds state



Subject Object

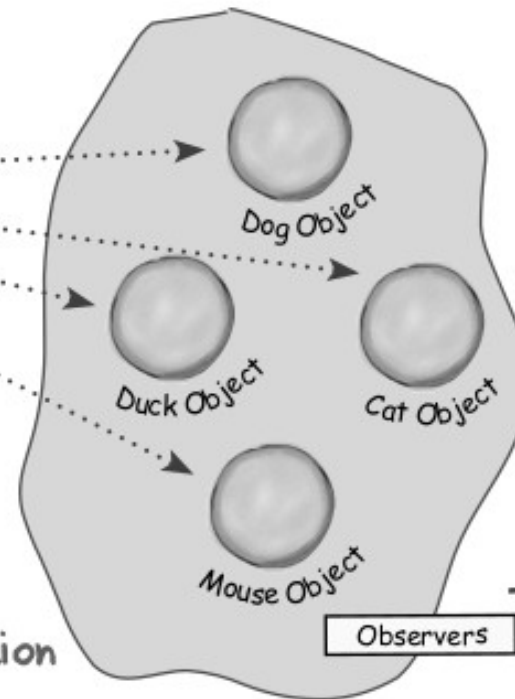
8

8

8

8

Automatic update/notification



Observers

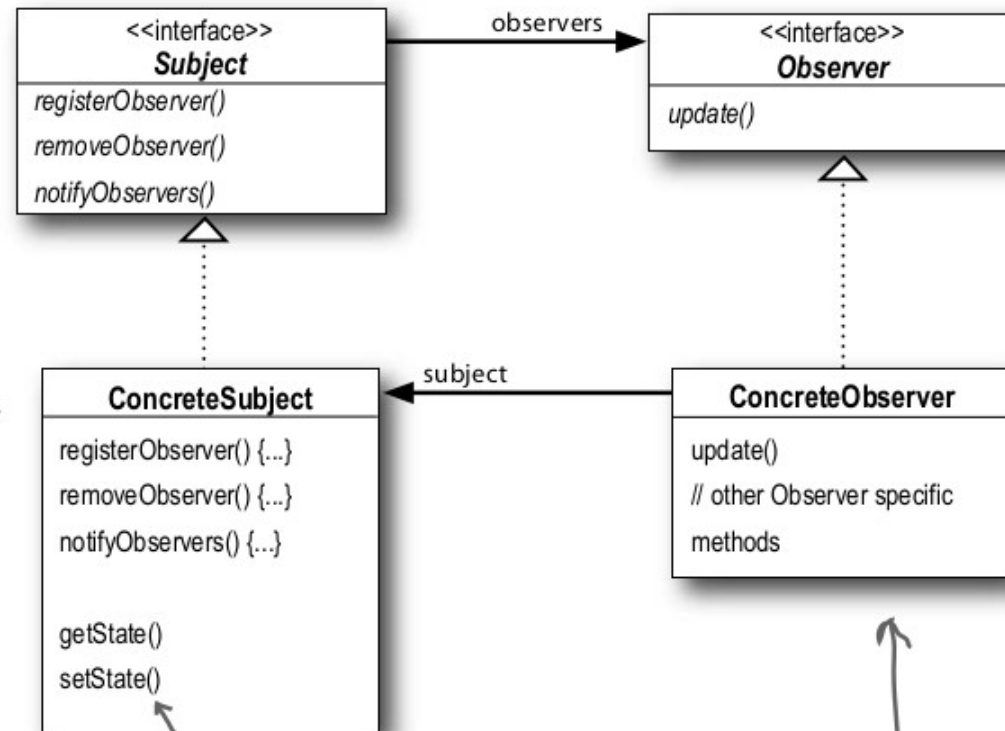
Dependent Objects

The Observer Pattern defined: the 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 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.

- El sujeto y los observadores define la relación “uno a muchos”.
- Los observadores son dependientes del sujeto, ya que cuando el estado del sujeto cambia, los observadores son notificados.
- El observador puede también ser actualizado con nuevos valores.



Con este diseño tenemos que:

- Lo único que un Sujeto sabe de un Observador es que implementa una interfaz determinada.



Con este diseño tenemos que:

- Lo único que un Sujeto sabe de un Observador es que implementa una interfaz determinada.
- Se pueden agregar observadores en cualquier momento.



Con este diseño tenemos que:

- Lo único que un Sujeto sabe de un Observador es que implementa una interfaz determinada.
- Se pueden agregar observadores en cualquier momento.
- No es necesario modificar al Sujeto para agregar nuevos Observadores.
- Los cambios entre el Sujeto u Observadores no afectan al otro.






Aquí viene un nuevo principio de diseño



Principio de diseño

- Esfuércese por obtener diseños débilmente acoplados entre objetos que interactúen.






Los diseños poco acoplados nos permiten construir sistemas OO más flexibles que pueden manejar el cambio porque minimizan la interdependencia entre los objetos.





Regresemos al problema del día de hoy





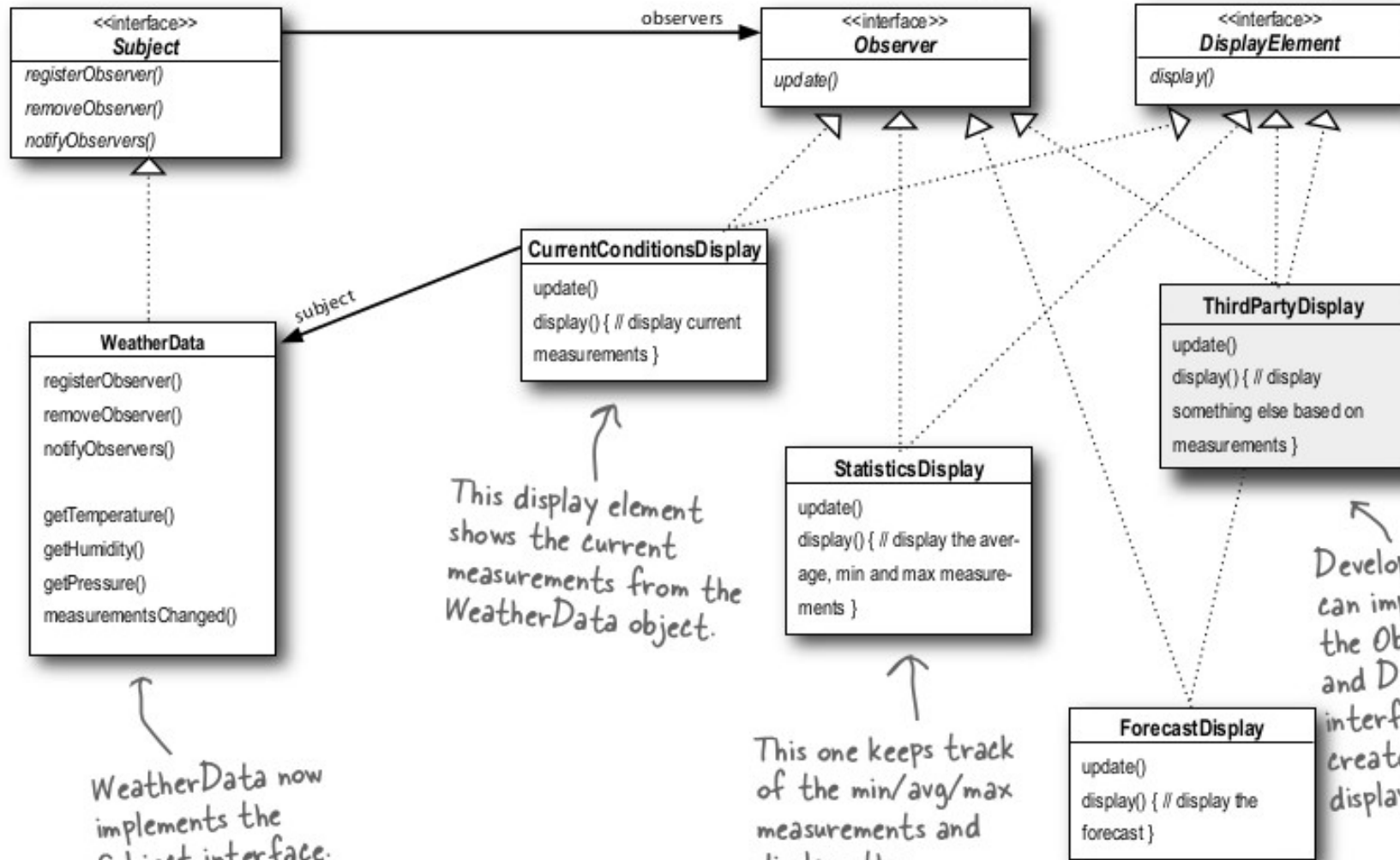
Entonces, ¿Cómo modelarías la aplicación
que monitorea el clima, utilizando el patrón
Observador?



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.



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



```
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 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);  
        }  
    }
```

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.

Here we implement the Subject Interface.

```
    }  
}  
  
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  
}
```

← 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.





Y la implementación de los observadores?



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);  
    }
```

The constructor is passed the
weatherData object (the Subject)
and we use it to register the
display as an observer.

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

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

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

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



```
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();  
    }  
  
    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");  
        }  
    }  
}
```



```
public class StatisticsDisplay implements Observer, DisplayElement {
    private float maxTemp = 0.0f;
    private float minTemp = 200;
    private float tempSum = 0.0f;
    private int numReadings;
    private WeatherData weatherData;

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

    public void update(float temp, float humidity, float pressure) {
        tempSum += temp;
        numReadings++;

        if (temp > maxTemp) {
            maxTemp = temp;
        }

        if (temp < minTemp) {
            minTemp = temp;
        }

        display();
    }

    public void display() {
        System.out.println("Avg/Max/Min temperature = " + (tempSum / numReadings)
            + "/" + maxTemp + "/" + minTemp);
    }
}
```



```
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.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.



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
```

```
%
```

