Observer Pattern

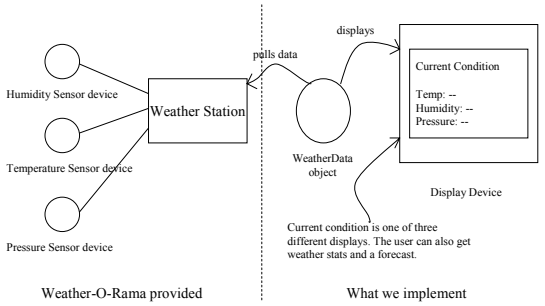
Head First Design Pattern

O'Reilly, First Ed. Oct 2004

Eric Freeman & Elisabeth Freeman

With Kathy Sierra & Bert Bates

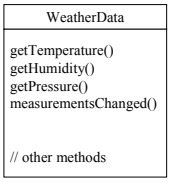
The Weather Monitoring Application



Our job is to create an app that uses the WeatherData object to update three displays for current condition, weather stats, and a forecast.

The system must be expandable – other developer can create new custom display elements and users can add or remove as many display elements as they want.

The WeatherData Object



Our job is to implement measurementChanged() so that it updates the three displays for current condition, weather stats, and a forecast.

First implementation …

```
public class WeatherData {
    // instance variable declarations

    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

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

    // other WeatherData methods here
}
```

What's wrong with our first implementation ?

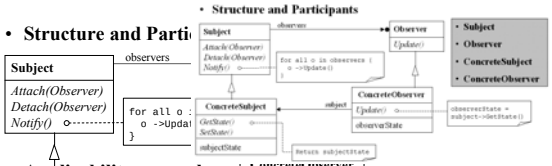
```
public class WeatherData {
    // instance variable declarations

    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

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

    // other WeatherData methods here
}
```

Area of change

Observer

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

- **Structure and Parti**



- **Applicability - use when:**
  - an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - a change to one object requires changing others, and you don't know how many objects need to be changed.
  - an object should be able to notify other objects without making assumptions about who these objects are.

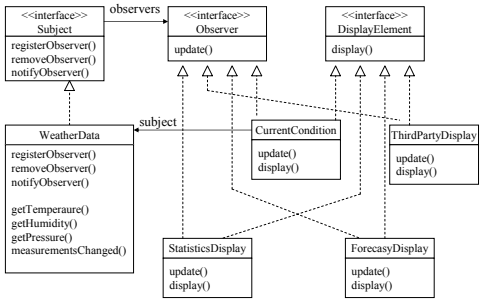
Design Principle

- Strive for loosely coupled designs between objects that interact.

The Power of Loose Coupling

- The Observer Pattern provides an object design where subjects and observers are loosely coupled
  - We can add new observers at any time
  - We never need to modify the subject to add new types of observers
  - We can reuse subject and observers independently of each other
  - Changes to either the subject or an observers will not affect the other

Designing The Weather Station



Implementing The Weather Station

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
public interface DisplayElement {
    public void display();
}
```

The WeatherData (1)

```
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);
        }
    }
```

The WeatherData (2)

```
        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;
            measurementChanged();
        }

        // other WeatherData methods
    }
```

Display Element

```
public class CurrentConditionDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

     public CurrentConditionDisplay(Subject weatherData) {
            this.weatherData = weatherData;
            weartherData.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 condition: "+temperature+"F degree"+
            humidity+"% humidity");
    }
}
```

The Weather Station

```
public class WeatherStation {

    public static void main(String[] args) {
            WeatherData weatherData = new WeatherData();
            CurrentConditionDisplay currentDisplay =  new CurrentConditionDisplay(weatherData);
            StatisticDisplay statisticDisplay = new StatisticDisplay(weatherData);
            ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

            // simulate the weather
            weatherData.setMeasurement(80,65,30.4f);
            weatherData.setMeasurement(82,70,29.2f);
            weatherData.setMeasurement(78,90,29.2f);

    }

}
```

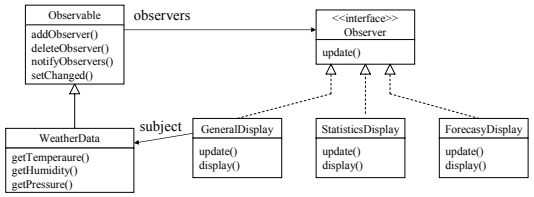
Java's built-in Observer Pattern

- java.util.Observable
- java.util.Observer

How Java's built-in Observer Pattern works

- For an object to become an observer…
  - implement the Observer interface and call addObserver() on any Observable object
- For the Observable to send notifications…
  - call the setChanged() method to signify that the state has changed in your object
  - call one of two notifyObservers() methods
    - **either** notifyObservers() **or** notifyObservers(Object arg)
- For an Observer to receive notifications…
  - Implement the update(Observable o, Object arg) method

Reworking The Weather Station



The WeatherData

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
      private float temperature;
      private float humidity;
      private float pressure;

      public WeatherData() { }

      public void measurementsChanged() {
        setChanged();
        notifyObservers();
      }

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

The WeatherData (2)

```
      public float getTemperature() {
            return temperature;
      }

      public float getHumidity() {
          return humidity;
      }

      public float getPressure() {
          return pressure;
      }
}
```

CurrentConditionDisplay

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionDisplay implements Observer, DisplayElement {
      Observable observable;
      private float temperature;
      private float humidity;

       public CurrentConditionDisplay(Observable observable) {
              this. observable = observable;
               observable.addObserver(this);
      }
      public void update(Observable obs, Object arg) {
              if (obs instanceof WeatherData) {
                   WeatherData weatherdata = (WeatherData) obs;
                   this.temperature = weatherdata.getTemperature();
                   this.humidity = weatherdata.getHumidity();
                   display();
              }
      }
      public void display() {
              System.out.println("Current condition: "+temperature+"F degree"+
              humidity+"% humidity");
      }
```

The dark side of java.util.Observable

- Observable is a class
  - You have to subclass it. That means you can't add on the Observable behavior to an existing class that already extends another superclass (limit its reuse potential).
  - Observable protect crucial method (setChanged())
    - You can't call setChanged() unless you've subclass Observable. You can't create an instance of the Observable, you have to subclass. The design violates design principles: … favor composition over inheritance.