Observer Pattern

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.

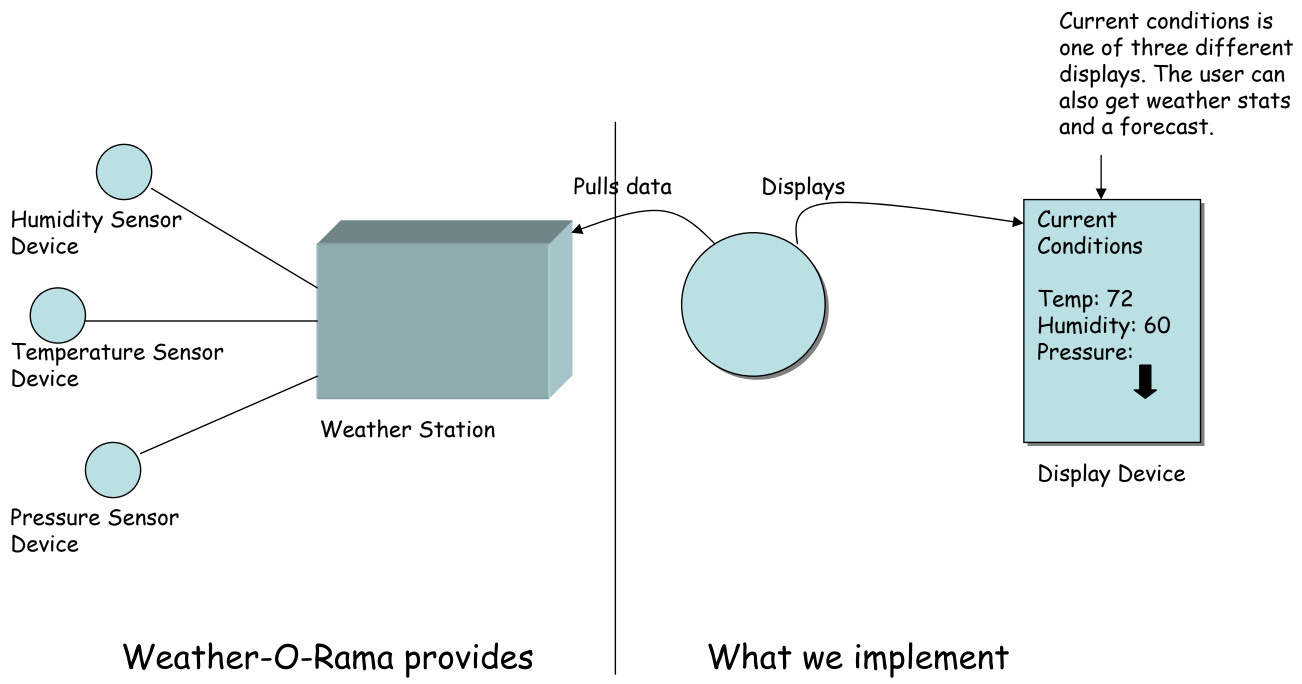
Toni Sellarès
Universitat de Girona

OO Design Principle

Strive for loosely coupled designs
between objects that interact.

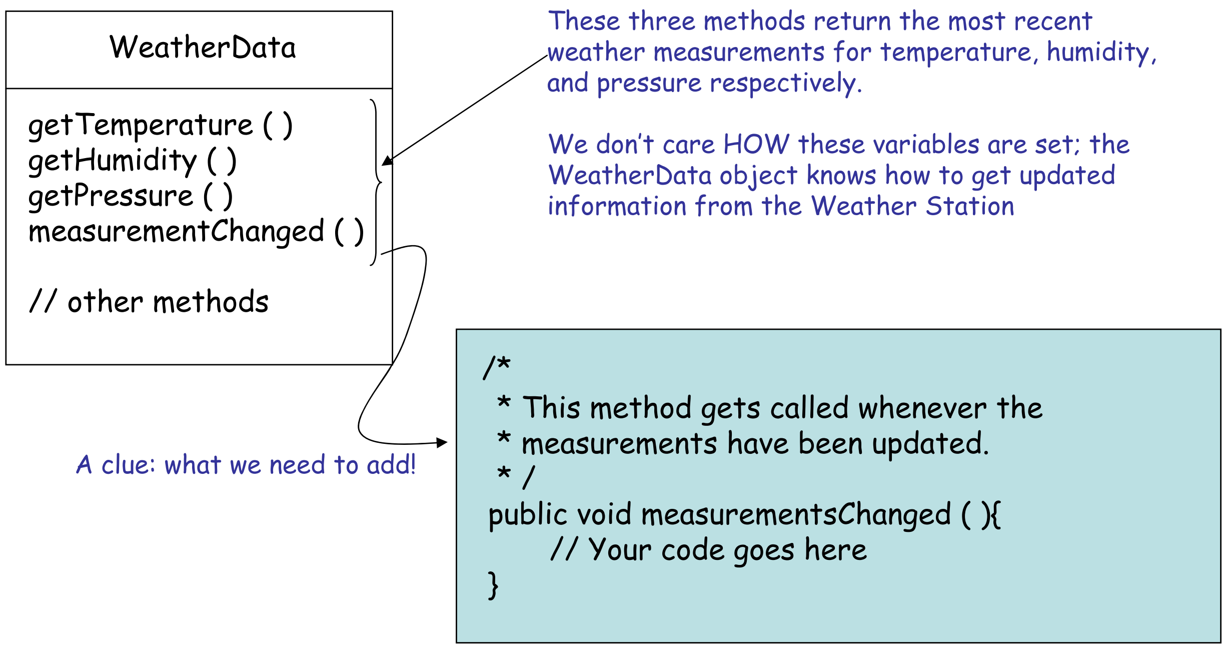
Loosely coupled designs allow us to build flexible OO systems that can handle changes because they minimize the interdependency between objects.

The Weather-O-Rama!



The Job: Create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

The WeatherData class


WeatherData
getTemperature ( )
getHumidity ( )
getPressure ( )
measurementChanged ( )
// other methods
These three methods return the most recent weather measurements for temperature, humidity, and pressure respectively.
We don't care HOW these variables are set; the WeatherData object knows how to get updated information from the Weather Station
A clue: what we need to add!

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

The Specs so far

- The WeatherData class has getter methods for three measurement values: temperature, humidity, and pressure.
- The measurementsChanged ( ) method is called anytime new weather measurement data is available (we don't know or care how this method is called; we just know that it is).
- 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.
- The system must be expandable - other developers can create new custom display elements and users can add or remove as many display elements as they want to the application.

A First Misguided Attempt at the Weather Station

```
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.

What’s wrong with the first implementation?

```
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
}
```

Area of change, we need to encapsulate this.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

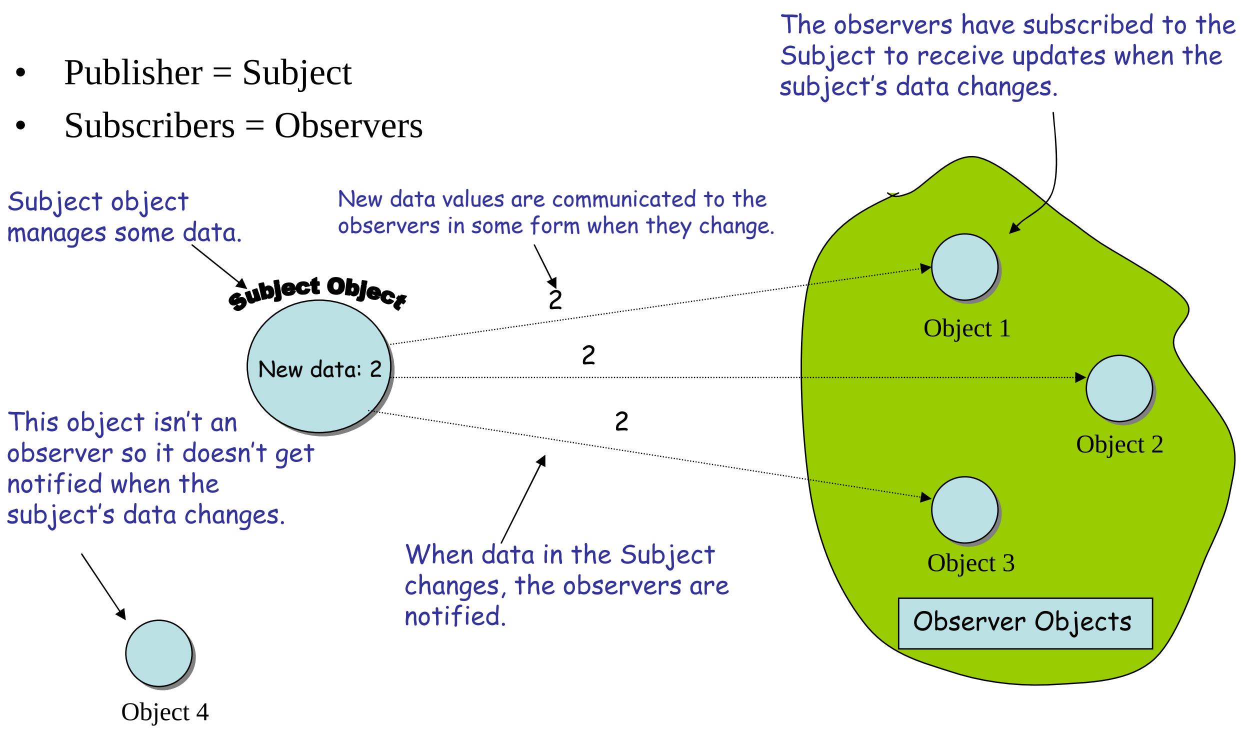
At least we seem to be using a common interface to talk to the display elements…they all have an update ( ) method that takes temp, humidity and pressure values.

Time for the Observer!

- The Newspaper or Magazine subscription model:
  - A newspaper publisher goes into business and begins publishing newspapers.
  - You subscribe to a particular newspaper, and every time there is a new edition, its gets delivered to you. As long as you remain a subscriber, you get new newspapers.
  - You unsubscribe when you don’t want the newspapers anymore -- and they stop being delivered.
  - While the publisher remains in business people, hotels, airlines etc constantly subscribe and unsubscribe to the newspaper.

Publishers + Subscribers = Observer Pattern

- Publisher = Subject
- Subscribers = Observers


The observers have subscribed to the Subject to receive updates when the subject's data changes.
Subject object manages some data.
New data values are communicated to the observers in some form when they change.
Subject Object
New data: 2
2
2
2
Object 1
Object 2
Object 3
Observer Objects
This object isn't an observer so it doesn't get notified when the subject's data changes.
When data in the Subject changes, the observers are notified.
Object 4

The Observer Pattern Defined

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

- Some observers may observe more than one subject (many-to-many relation).
- The update should specify which subject changed.

Motivation:

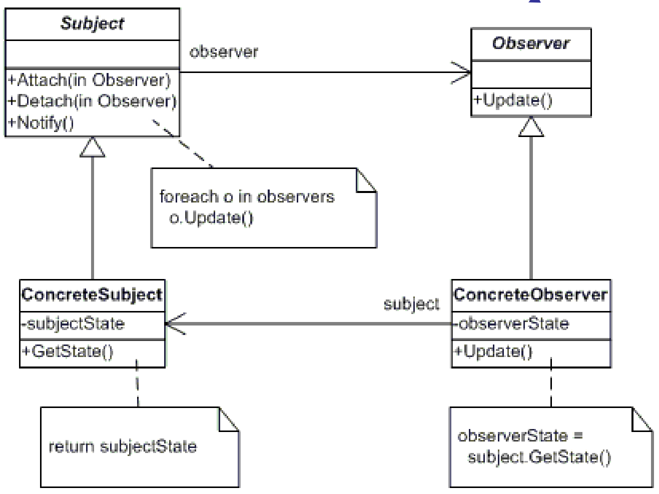
- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

Observer Class Diagram

Objects use Subject 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 gets called when the Subject's state changes.



The concrete subject may also have methods for setting and getting its state.

A concrete subject always implements the Subject interface. In addition to the register (attach) and remove (detach) methods, the concrete subject implements a notify() method to notify observers whenever state changes.

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

Observer pattern: Structural Code

```
/**
 * Maintains a reference to a ConcreteSubject object. Stores
 * state that should stay consistent with the subject's.
 * Implements the Observer updating interface to keep its
 * state consistent with the subject's.
 */

public class Test{
     public static void main( String arg[] ){
         Subject subject = new ConcreteSubject();
         Observer observer = new ConcreteObserver();
         subject.attach( observer );
         subject.notify();
         }
}
```

```java
import java.util.Vector;
/**
 * Knows its observers. Any number of Observer objects may observe a subject.
 */
public class Subject{
          private Vector observers = new Vector();
          public void attach( Observer observer ){
              if( observers.contains( observer ) == false )
                  {observers.addElement( observer );}
          }
          public boolean detach( Observer observer ){
              return observers.removeElement( observer );
          }
          public void notifyObservers(){
             // Synchronous update. Beware, this mechanism will block as each
            // observer is updated. Depending on how sensitive your code is to
            // blocking, you may want to kick-off a thread for each observer that
            // requires notification and then wait for all threads to complete.
             for( int i = 0; i < observers.size(); ++i ){
                              Observer observer = (Observer) observers.elementAt(i);
                              observer.update(); }
          }
}
```

```java
/**
 * Defines an updating interface for objects that should
 * be notified of changes in a subject.
 */
public interface Observer{
          void update();
}

/**
 * Stores state fo interest to ConcreteObserver objects. Sends
 * a notification to its observers when its state change.
 */
public class ConcreteSubject extends Subject {
          public int getState() { return 1; }
}

/**
 * Maintains a reference to a ConcreteSubject object. Stores state that should
 * stay consistent with the subject's. Implements the Observer updating
 * interface to keep its state consistent with the subject's.
 */
public class ConcreteObserver implements Observer {
          public void update() { System.out.println( "update() called." ); }
}
```

Design choices

- When to notify Observers:

  - automatic on each change
  - triggered by client

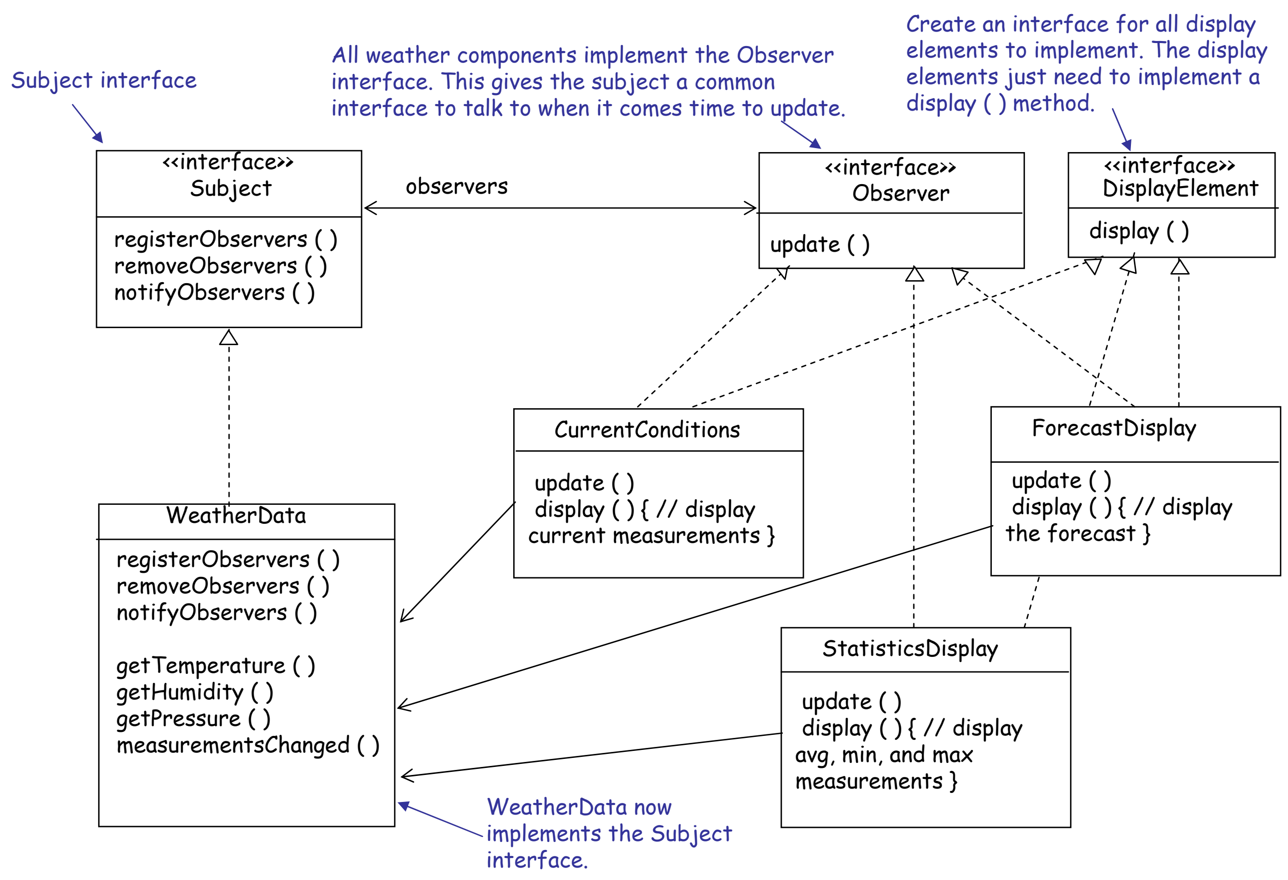
- How to communicate information about the change:

  - push: subject gives details to observers
  - pull: subject notifies only that a change has occurred, observer queries subject about details

Summary

- Subject(s) update Observers using a common interface.
- Observers are loosely coupled in that Subject(s) knows nothing about them, other than they implement the Observer interface.
- Don’t depend on a specific order of notification for your Observers.

Designing the Weather Station


Subject interface
All weather components implement the Observer interface. This gives the subject a common interface to talk to when it comes time to update.
Create an interface for all display elements to implement. The display elements just need to implement a display ( ) method.
<<interface>> Subject
registerObservers ( )
removeObservers ( )
notifyObservers ( )
observers
<<interface>> Observer
update ( )
<<interface>> DisplayElement
display ( )
CurrentConditions
update ( )
display ( ) { // display current measurements }
ForecastDisplay
update ( )
display ( ) { // display the forecast }
WeatherData
registerObservers ( )
removeObservers ( )
notifyObservers ( )
getTemperature ( )
getHumidity ( )
getPressure ( )
measurementsChanged ( )
StatisticsDisplay
update ( )
display ( ) { // display avg, min, and max measurements }
WeatherData now implements the Subject interface.

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 ( );
}
```

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.

The Observer interface is implemented by all observers, so they all have to implement the update ( ) method.

These are the state values the Observers get from the Subject when a weather measurement changes.

The DisplayElement interface just includes one method, display ( ), that we will call when the display element needs to be displayed.

Implementing the Subject Interface in WeatherData

```
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 j = observer.indexOf(o);
            if (j >= 0) {
                 observers.remove(j);
            } }
      public void notifyObservers ( ) {
            for (int j = 0; j < observers.size(); j++) {
                 Observer observer = (Observer)observers.get(j);
                 observer.update(temperature, humidity, pressure);
            }}
      public void measurementsChanged ( ) {
             notifyObservers ( ); }
      // add a set method for testing + other methods.
}
```

Added an ArrayList to hold the Observers, and we create it in the constructor

Here we implement the Subject Interface

Notify the observers when measurements change.

The Display Elements

Implements the Observer and DisplayElement interfaces

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
      private float temperatue;
      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 constructors 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.