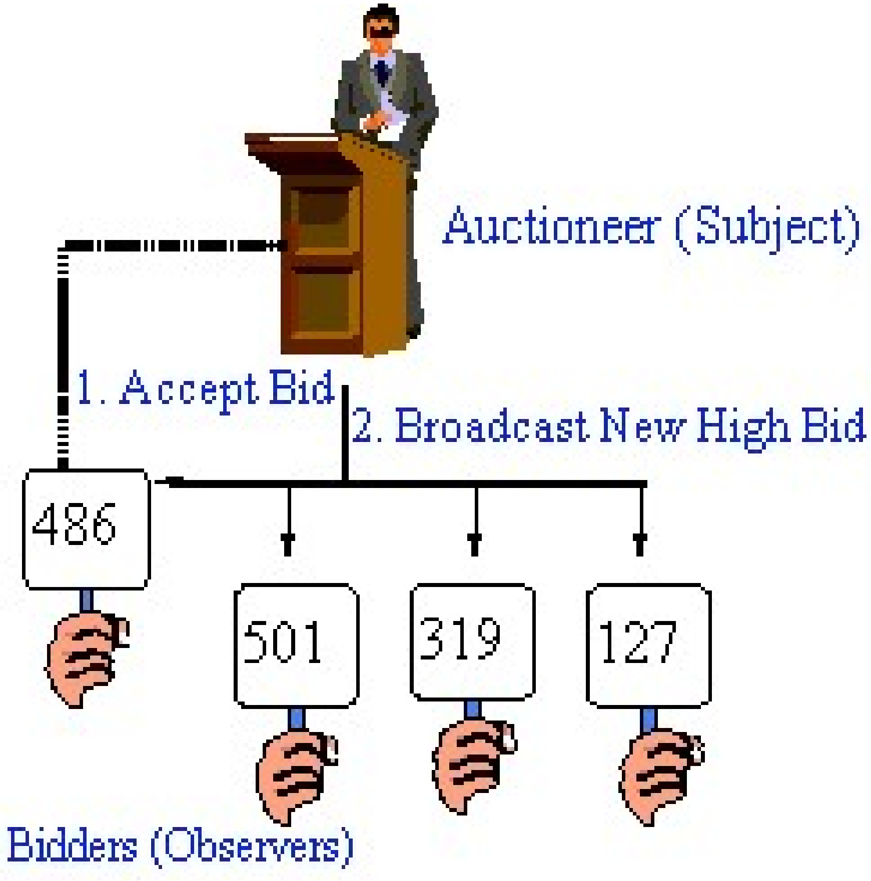
Observer Pattern

Keeping your Objects in the Know!

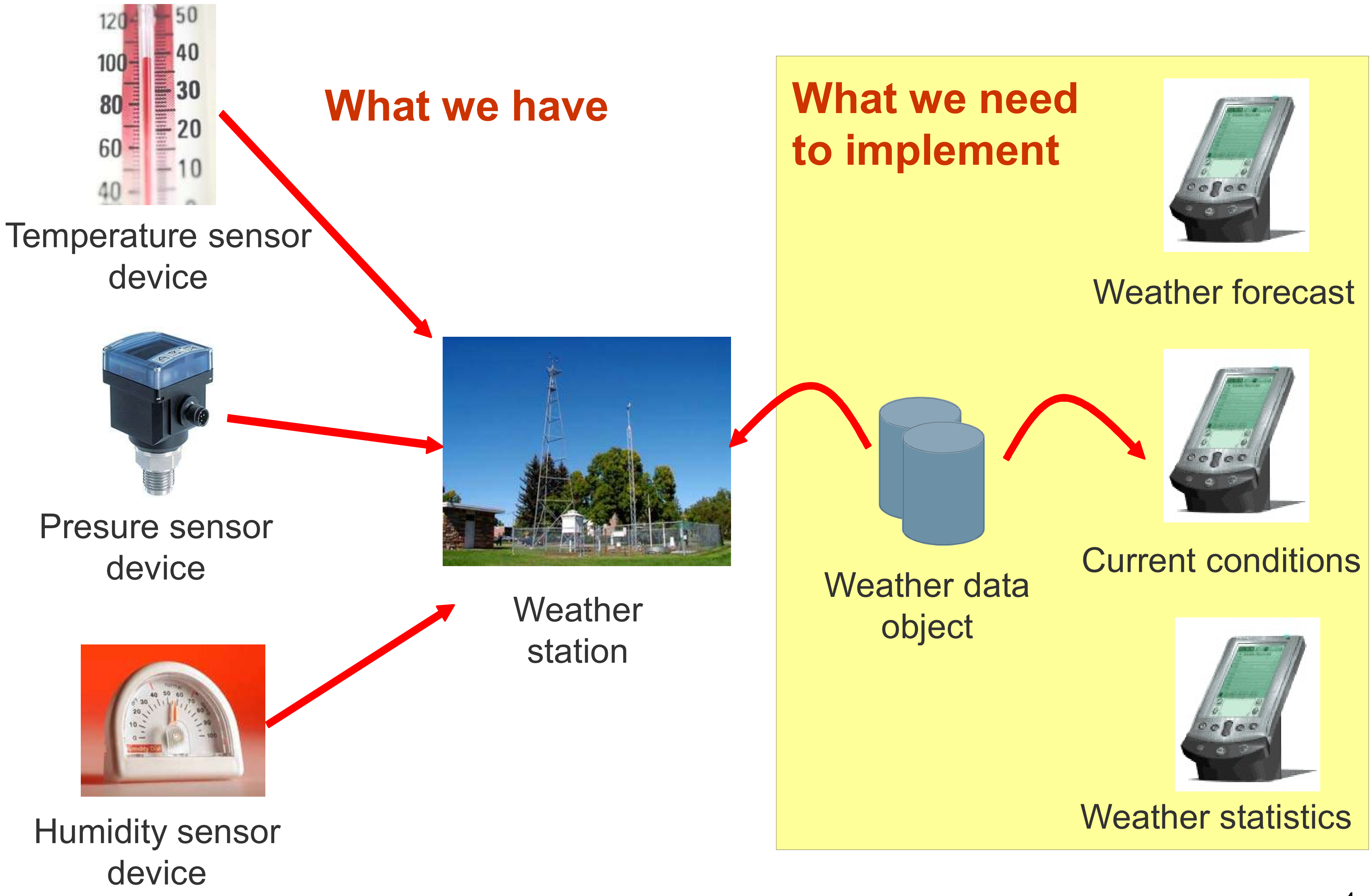
Observer – A Non Software Example



Motivation Weather Station

- Build a weather monitoring station that have
  - `Weather Station` – hardware device which collects data from various sensors (humidity, temperature, pressure)
  - `WeatherData` object which interfaces with the `Weather Station` hardware
- Need to implement three display elements that use the `WeatherData` and must be updated each time `WeatherData` has new measurements
  - a current conditions display Temp, Humidity, Pressure change
  - a weather statistics display Avg. temp, Min. temp, Max. temp
  - and a forecast display
- The system must be expandable
  - can create new custom display elements and can add or remove as many display elements as they want to the application.

A Weather Monitoring Application



What we have

Temperature sensor device

Presure sensor device

Humidity sensor device

Weather station

What we need to implement

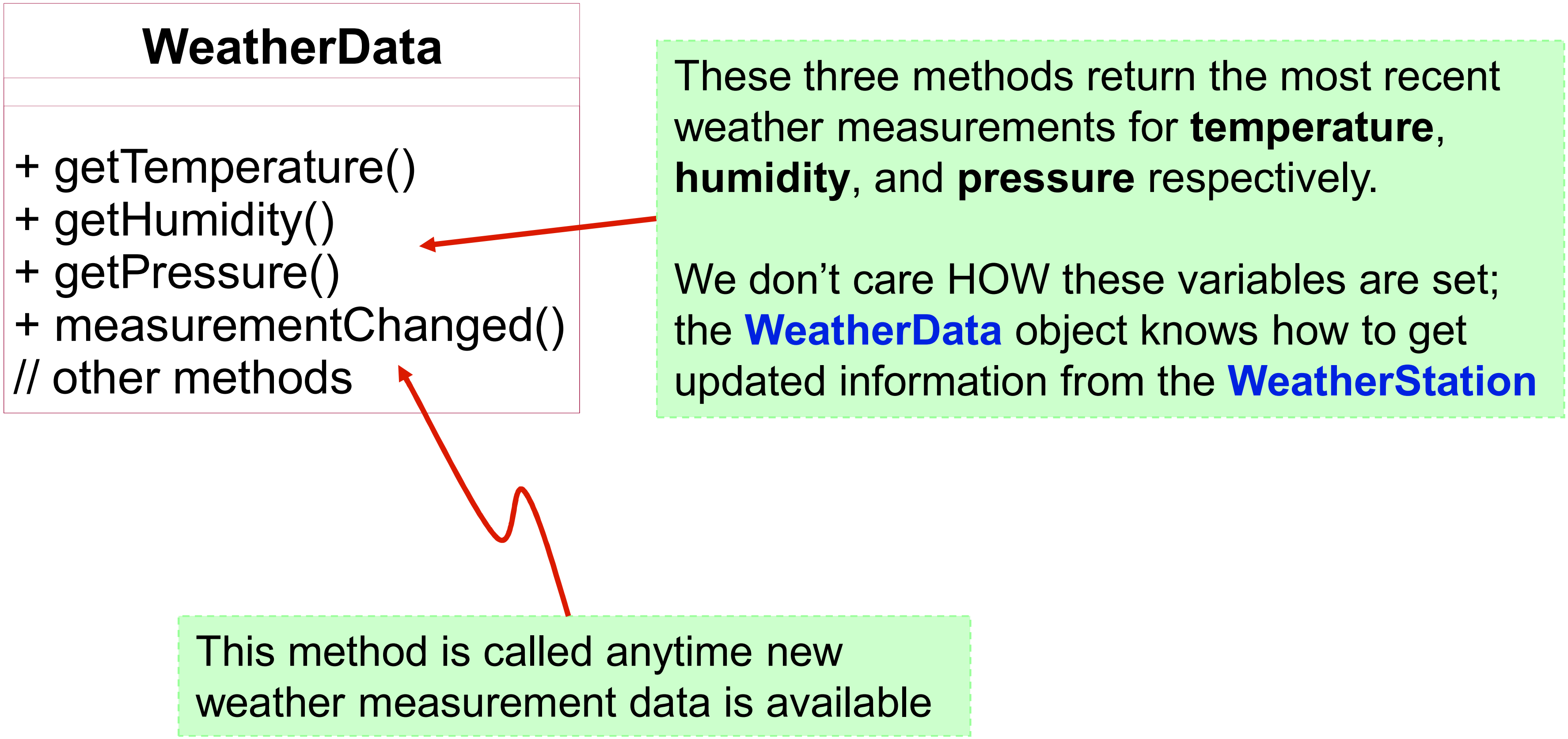
Weather data object

Weather forecast

Current conditions

Weather statistics

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 **WeatherStation**

This method is called anytime new weather measurement data is available

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?

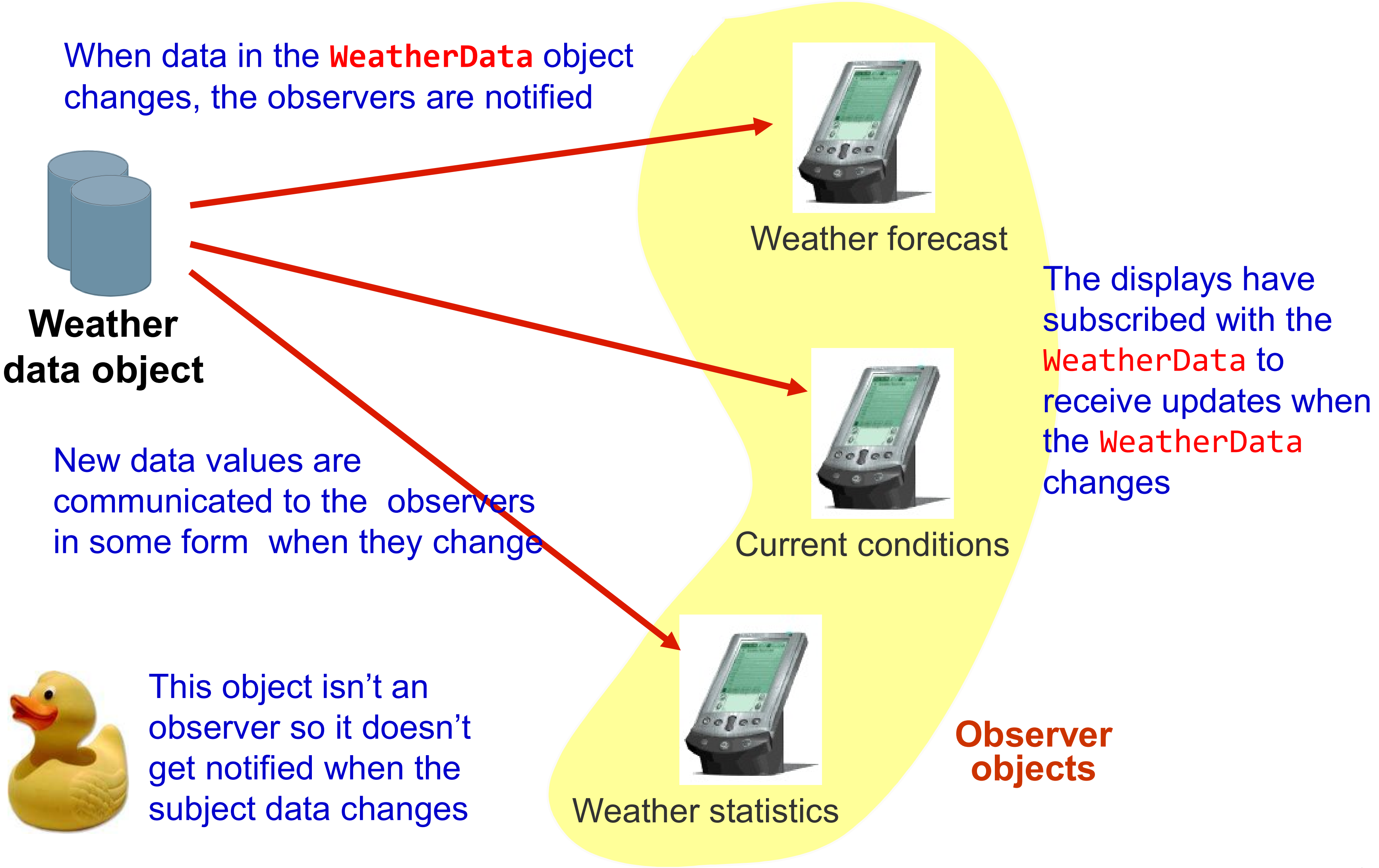
```
public class WeatherData {
   // instance variable declarations
   public void measurementsChanged() {
      float temp = getTemperature();
      float humidity = getHumidity();
      float pressure = getPressure();

      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.

Publisher + Subscriber = Observer



When data in the `WeatherData` object changes, the observers are notified

Weather data object

Weather forecast

The displays have subscribed with the `WeatherData` to receive updates when the `WeatherData` changes

Current conditions

New data values are communicated to the observers in some form when they change

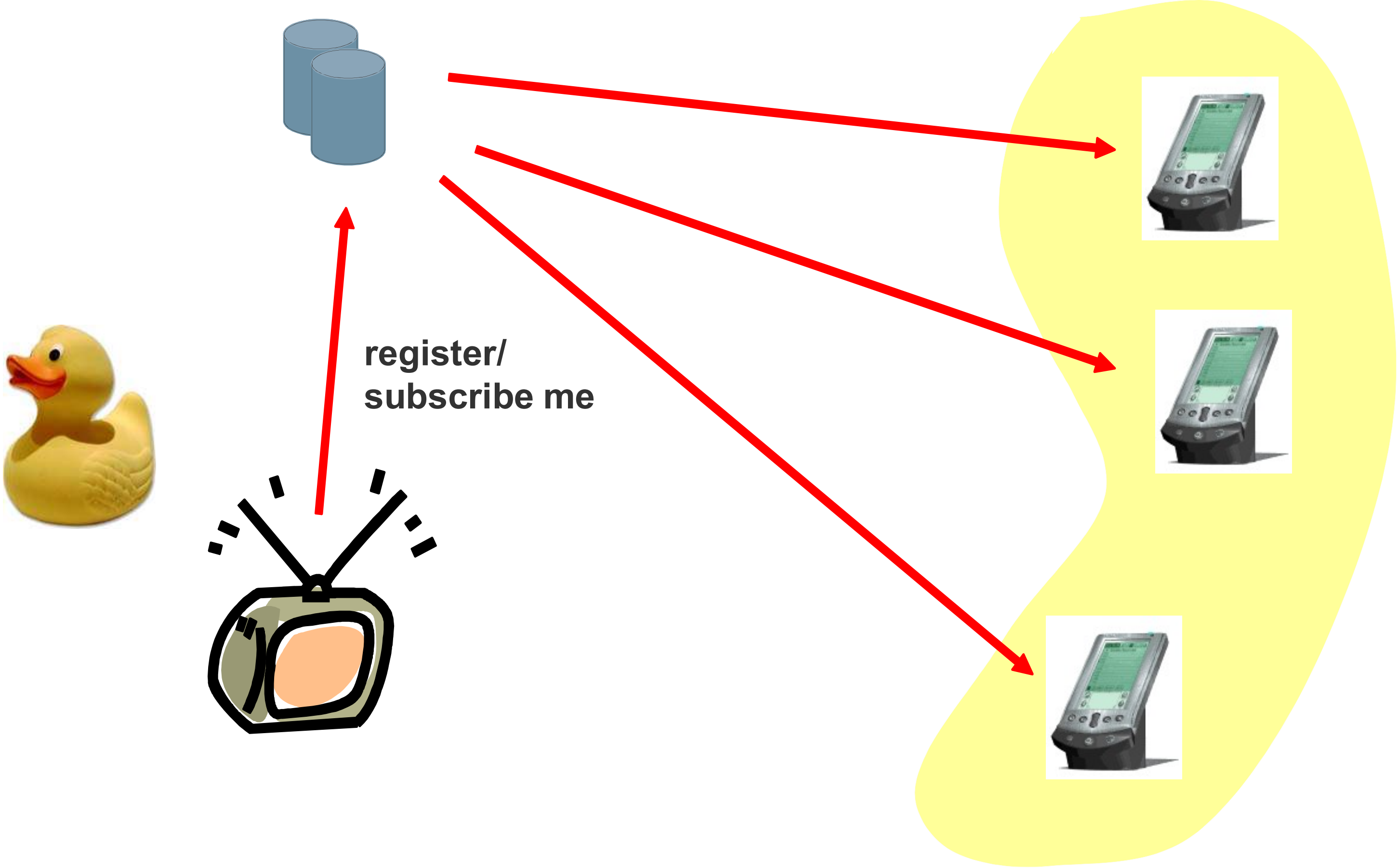
This object isn't an observer so it doesn't get notified when the subject data changes

Weather statistics

Observer objects

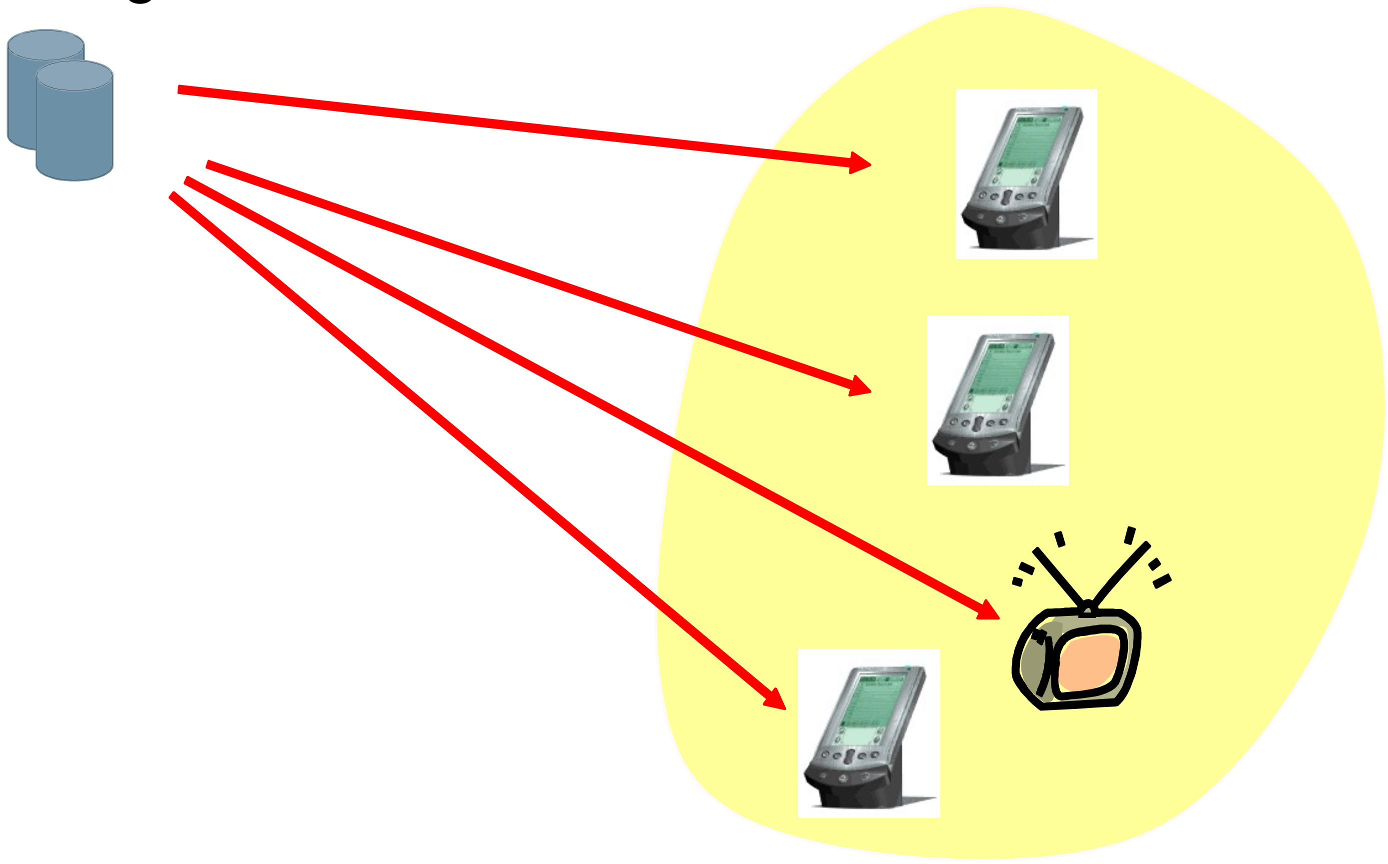
Adding Observers

- A TV station comes along and tells the Weather data that it wants to become an observer



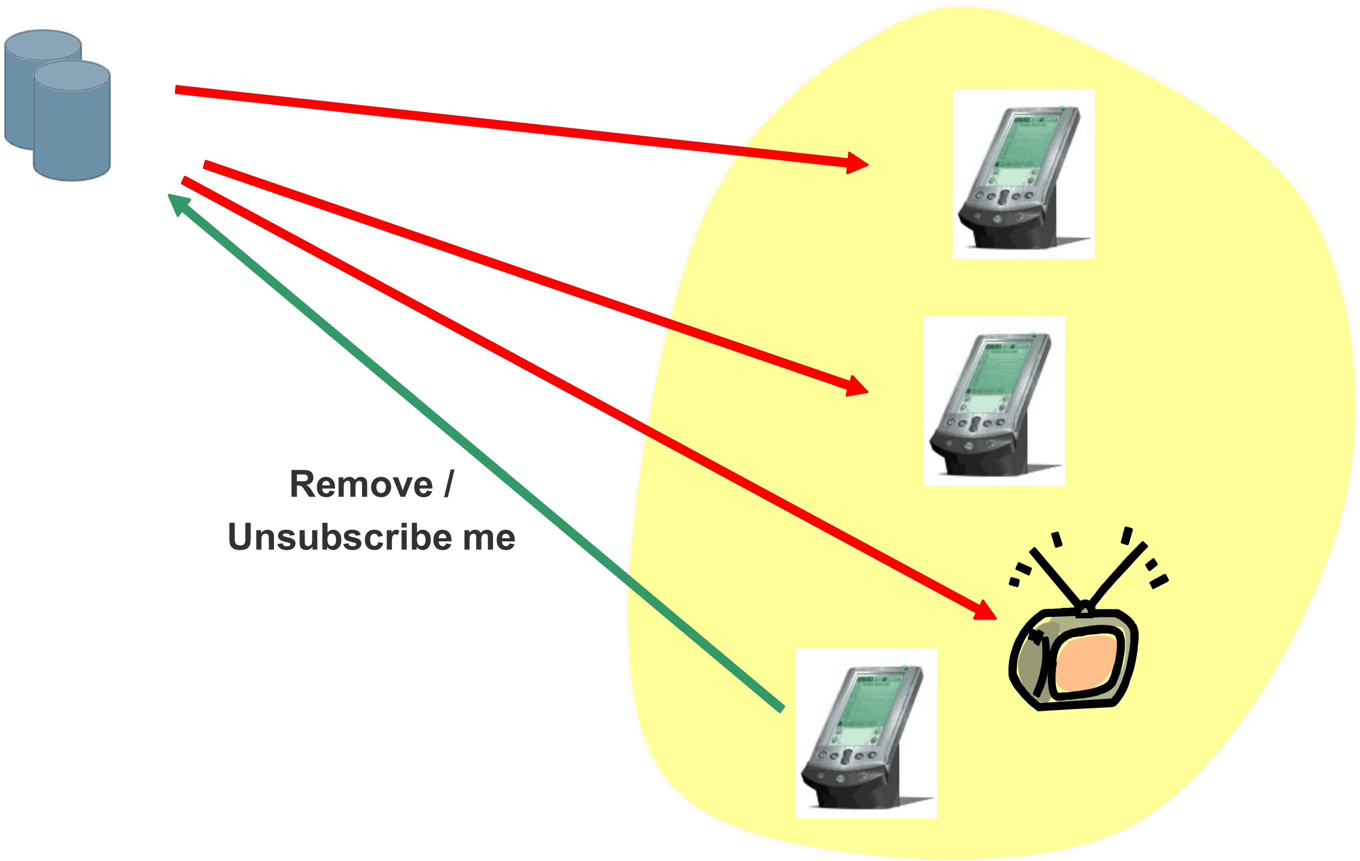
Adding Observers

- The TV station is now an official observer
  - It gets a notification when the Weather object has changed



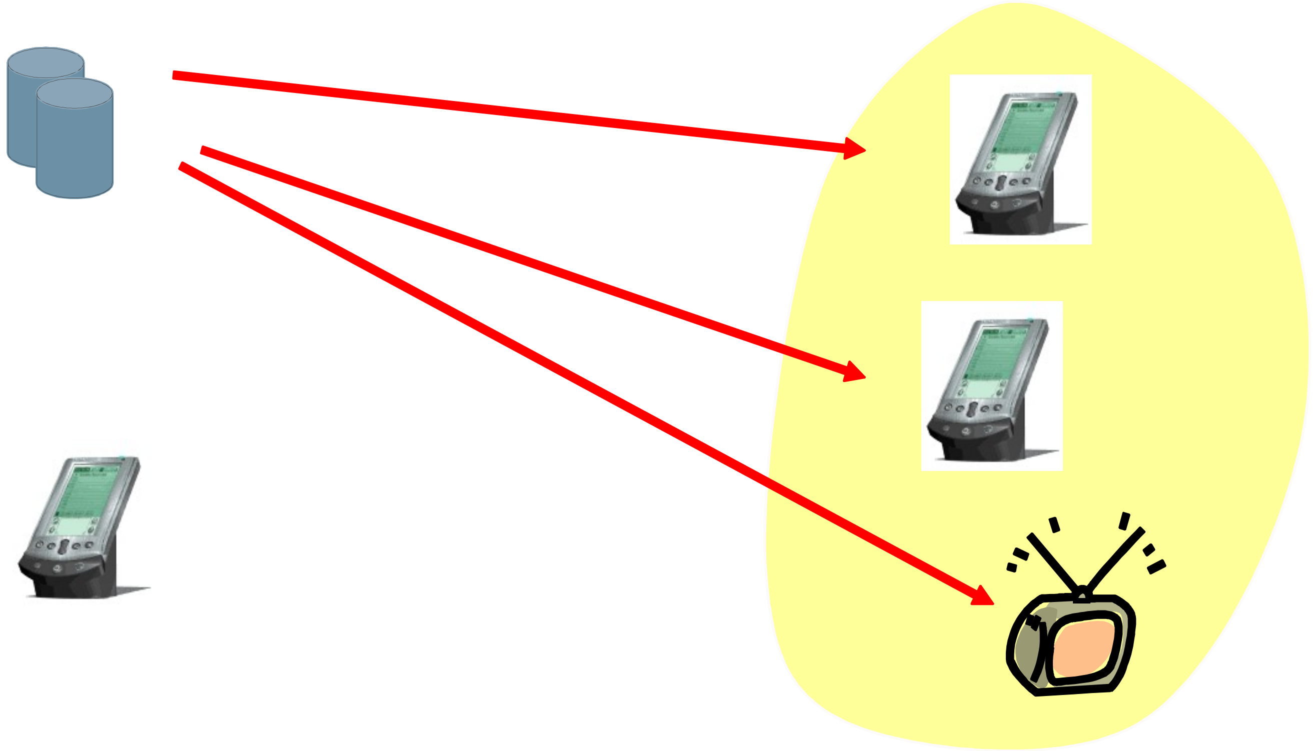
Removing Observers

- One of the displays asks to be removed as an observer



Removing Observers

- All the other observers can get another notification except the the display that has been recently removed from the set of observers



Design Principle

Strive for loosely coupled designs between objects that interact.

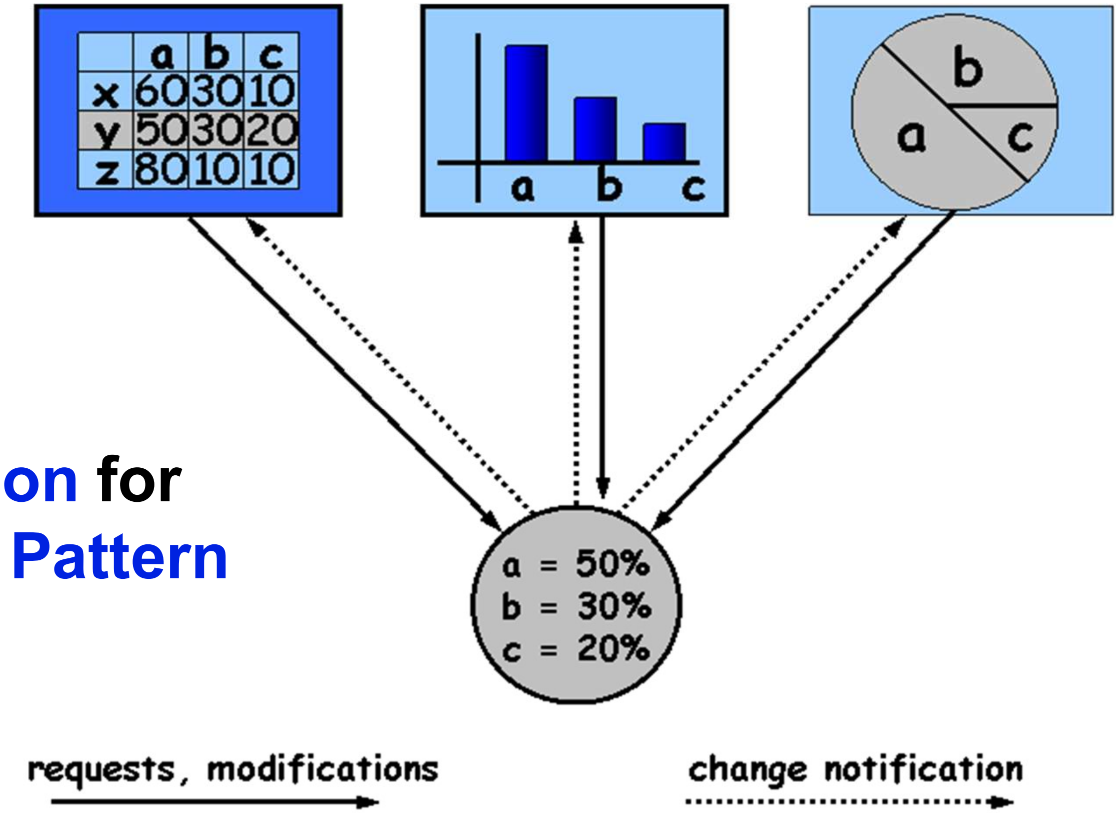
- Loosely coupled designs allow us to build flexible OO systems that can handle changes because they minimize the interdependency between objects.

The Constitution of Software Architects

- Encapsulate what varies
- Program through an interface not to an implementation
- Favor Composition over Inheritance
- Classes should be open for extension but closed for modification
- Strive for loosely coupled designs between objects that interact.
- ?????????
- ?????????
- ?????????
- ?????????

The 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.

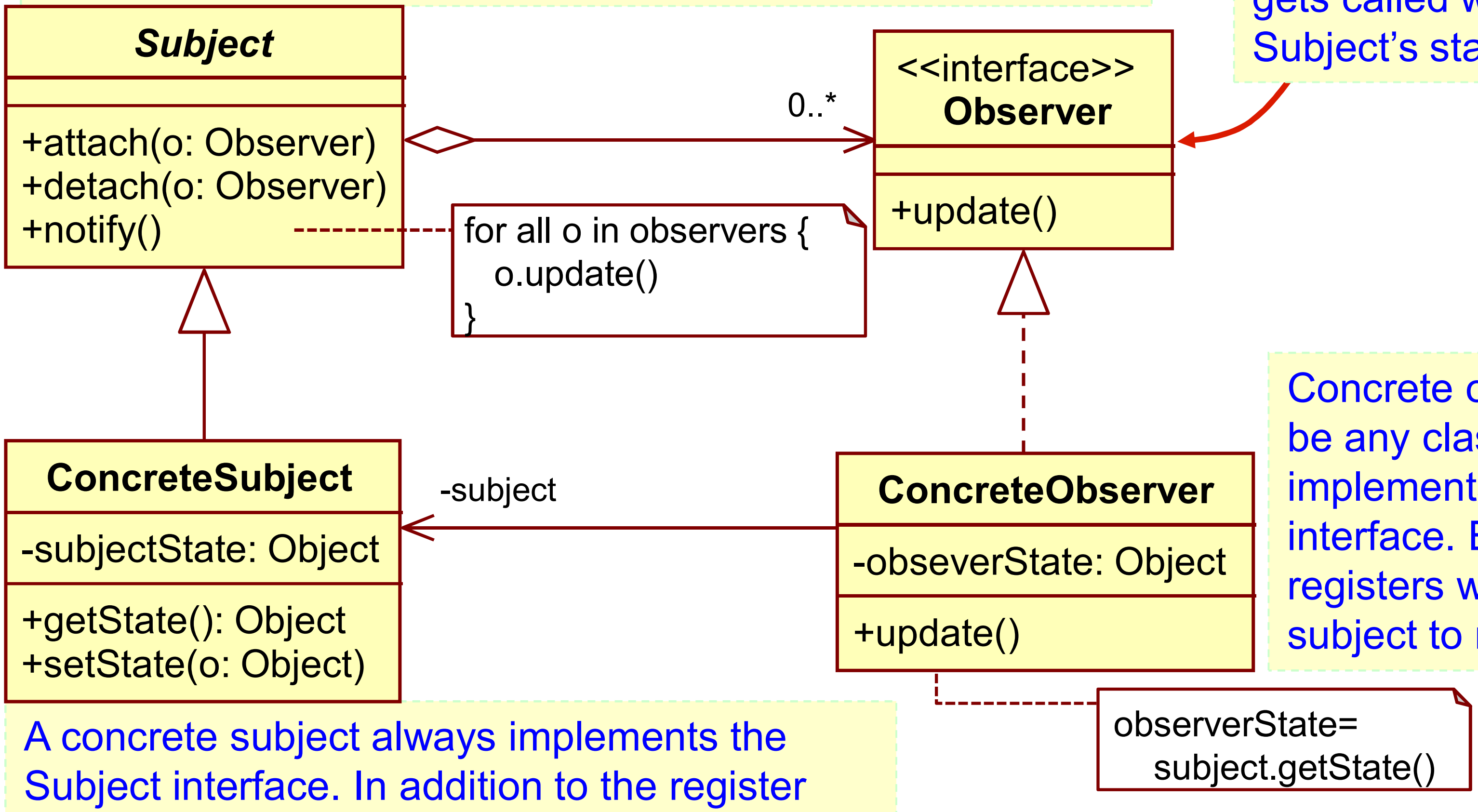


	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Motivation for Observer Pattern

Structure Observer Pattern

Knows its numerous observers. Provides an interface for attaching and detaching observer objects.
Sends a notification to its observers when its state changes.



Subject

+attach(o: Observer)
+detach(o: Observer)
+notify()

0..*

<<interface>>
Observer

+update()

```
for all o in observers {
   o.update()
}
```

ConcreteSubject

-subjectState: Object

+getState(): Object
+setState(o: Object)

-subject

ConcreteObserver

-obseverState: Object

+update()

```
observerState=
   subject.getState()
```

Defines an update interface for concrete observers, that gets called when the Subject’s state changes.

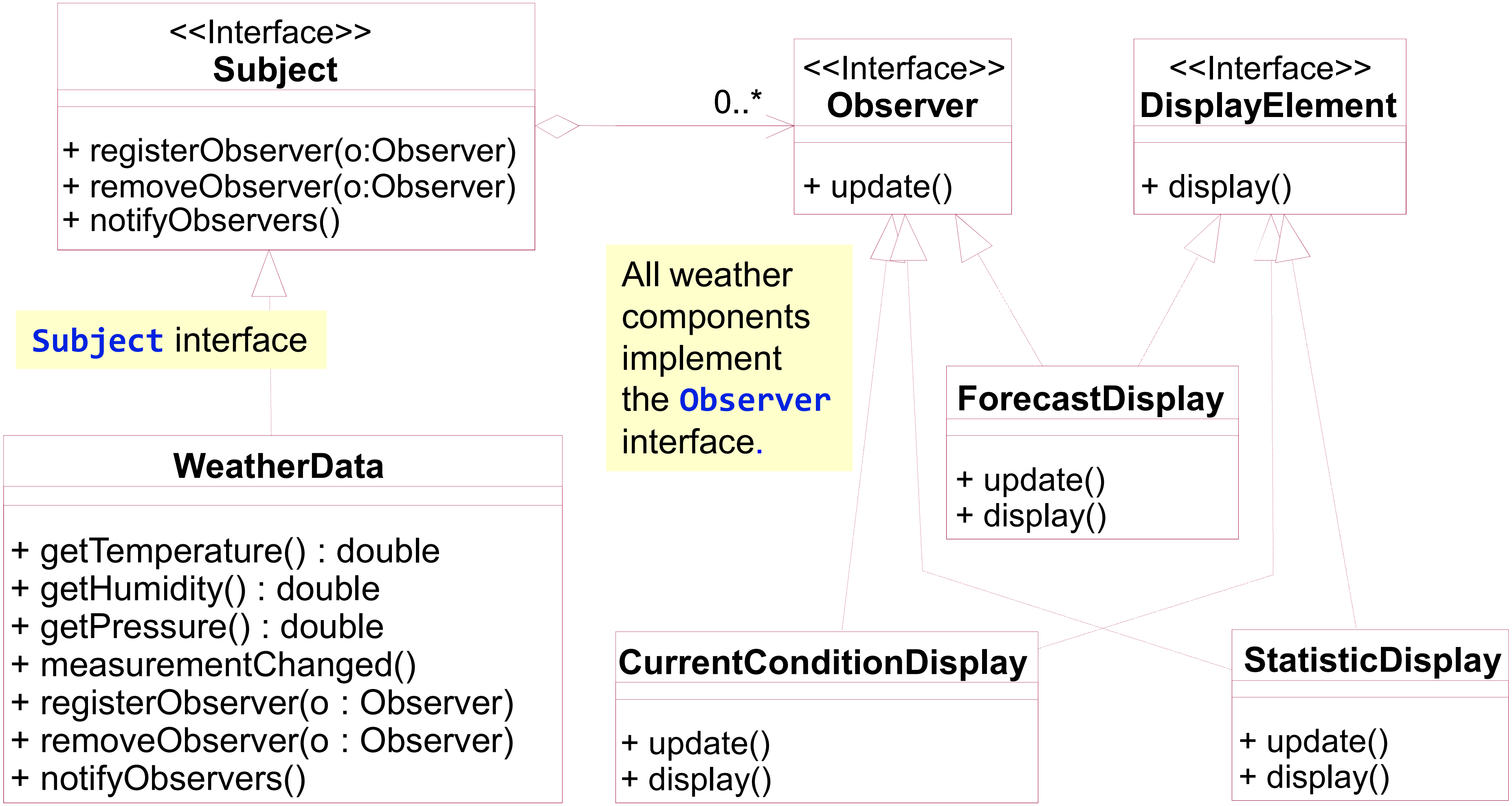
Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

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.

Consequences

- Abstract coupling between subject and observer
  - Coupling is abstract, thus minimal (concrete class isn’t known).
  - Can have multiple layers of abstraction.
- Support for broadcast communication
  - Subject doesn’t need to know its receivers.
- Unexpected updates
  - Can be blind to some changes in the subject (i.e., observer doesn’t know “what” has changed in the subject).

Designing the Weather Station



<<Interface>>
Subject

+ registerObserver(o:Observer)
+ removeObserver(o:Observer)
+ notifyObservers()

0..*

<<Interface>>
Observer

+ update()

<<Interface>>
DisplayElement

+ display()

Subject interface

All weather components implement the Observer interface.

ForecastDisplay

+ update()
+ display()

WeatherData

+ getTemperature() : double
+ getHumidity() : double
+ getPressure() : double
+ measurementChanged()
+ registerObserver(o : Observer)
+ removeObserver(o : Observer)
+ notifyObservers()

CurrentConditionDisplay

+ update()
+ display()

StatisticDisplay

+ update()
+ display()

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();
}
```

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

Implementing the Subject Interface

```
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 o : observers) {
         o.update(temperature, humidity, pressure);
      }
   }
   public void measurementsChanged() {
      notifyObservers();
  }
}
```

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

```
public class CurrentConditionsDisplay
       implements Observer, DisplayElement {
   private float temperature;
   private float humidity;
   public CurrentConditionsDisplay(Subject 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()**

TestDrive

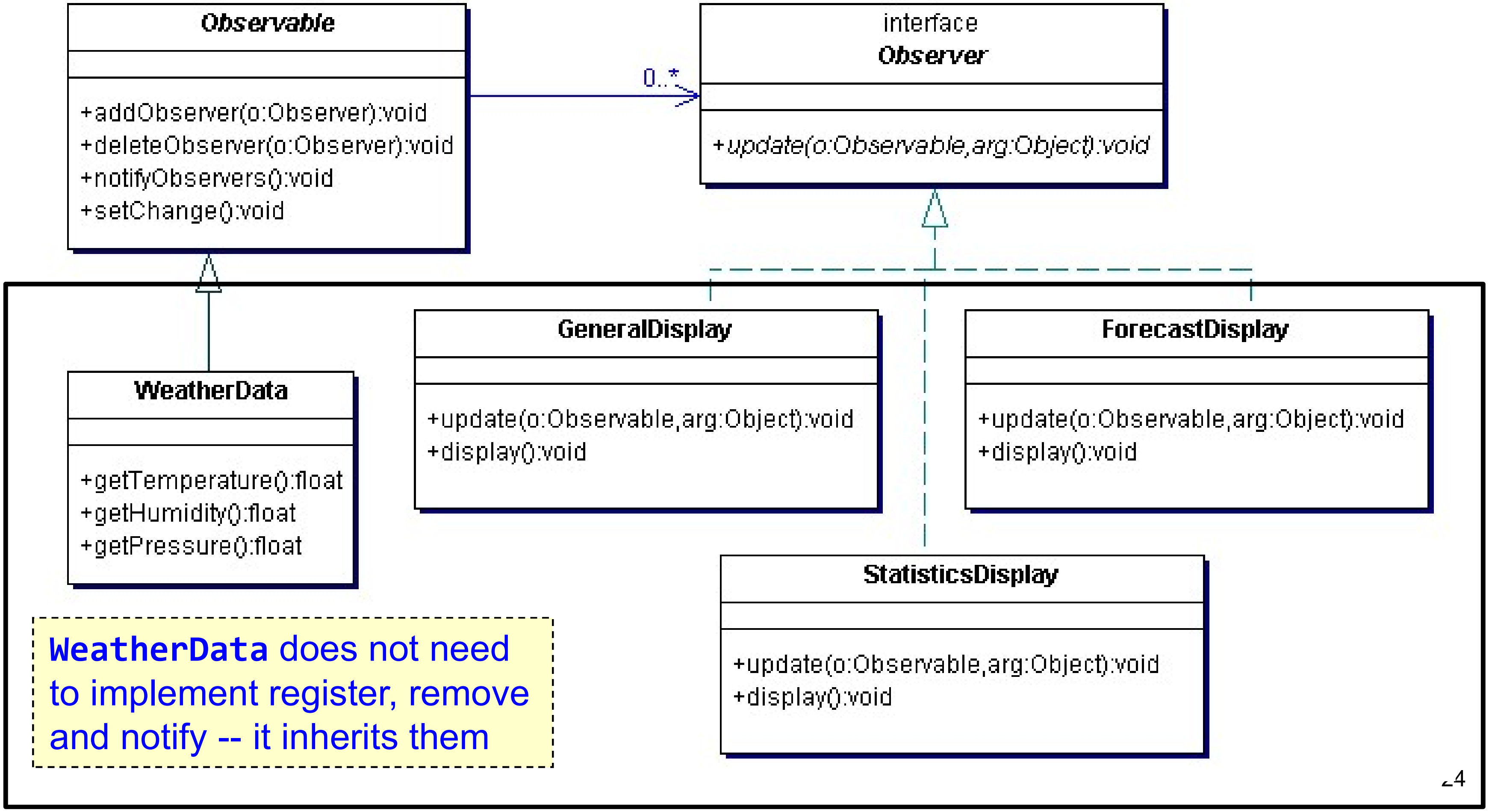
```
public class WeatherStation {

   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);
   }
}
```

Java's Built-in Observer Pattern

- For an Object to become an Observer:
  - Implement the **`java.util.Observer`** interface and call **`addObserver()`** on any **`Observable`** object. To remove use **`deleteObserver()`** method.
- For the Observable to send notifications
  - Extend **`java.util.Observable`** superclass
  - Then a 2 step process:
    1. First call the **`setChanged()`** method to signify that the state has changed in your object.
    2. call one of two methods: **`notifyObservers()`** or **`notifyObservers(Object arg)`**
- For the Observer to receive notifications
  - Implement the update() method as before **`update(Observable o, Object arg)`**

Java’s Built-in Observer Pattern

- Observer == Observer
- Observable == Subject



`WeatherData` does not need to implement register, remove and notify -- it inherits them

WeatherData extends the `Observable`

```
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();
   }
```

Rework the CurrentConditionsDisplay

```java
import java.util.Observable;
import java.util.Observer;
public class CurrentConditionsDisplay
          implements Observer, DisplayElement {
  Observable observable;
  private float temperature;
  private float humidity;

  public CurrentConditionsDisplay(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 conditions: " + temperature
               + "F degrees and " + humidity + "% humidity");
  }
}
```

implement **java.utils.Observer**

Implementation Issues: Updates

- The simple observer protocol does not specify what changed in the subject
- More fine-grained update protocols may specify the extent of the change:
  - `update(Object changedState)`
    or `cellUpdate (int x, int y, float value)`
- Some observers may observe more than one subject (many-to-many relation) E.g., a graph can depend on several different sheets
  - The update should specify which subject changed:
    `update(Subject changedSubject)`

Update Protocols: Push or Pull

- **Pull**: The subject should provide an interface that enables observers to query the subject for the required state information to update their state.
- **Push**: The subject should send the state information that the observers may be interested in.
- Pull assumes no knowledge of the subject about its observers, so it is more re-usable but less efficient.
- Push assumes that the subjects has some knowledge about what the observers need, so it is less re-usable but more efficient.
- Intermediate: when the observer registers using **attach()**, it specifies the kind of events it is interested in.

Update Protocols: Push or Pull

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

We first call the **`setChanged()`** to indicate that the state has changed.

We aren't sending a data object with the **`notifyObservers()`** call. The **`Observers`** are aware of the subject and they will use that to "pull" the latest information from the subject.

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

A "push" method -- the modified data is being pushed to the observers.

Other places to find Observer Pattern in Java

- Both JavaBeans and Swing also provide implementations of the **Observer** pattern
- Look under the hood of **`JButton`**'s super class **`AbstractButton`**
  - Has many add/remove listener methods: allow you to add and remove observers (called listeners in Swing)
  - These "listeners" are available for various types of events that occur on the Swing component
  - Example: **`ActionListener`** lets you "listen in" on any types of actions that might occur on a button -- like a button press.
- Also check out the **`PropertyChangeListener`** in JavaBeans

A Simple Example:
A “Life-Changing” Application

- Background:
  - Simple application with one button that says “Should I do it”.
  - When you click on the button the “listeners” get to answer the question in any way they want.

  - Code sample: implements 2 such listeners: `AngelListener` and `DevilListener`

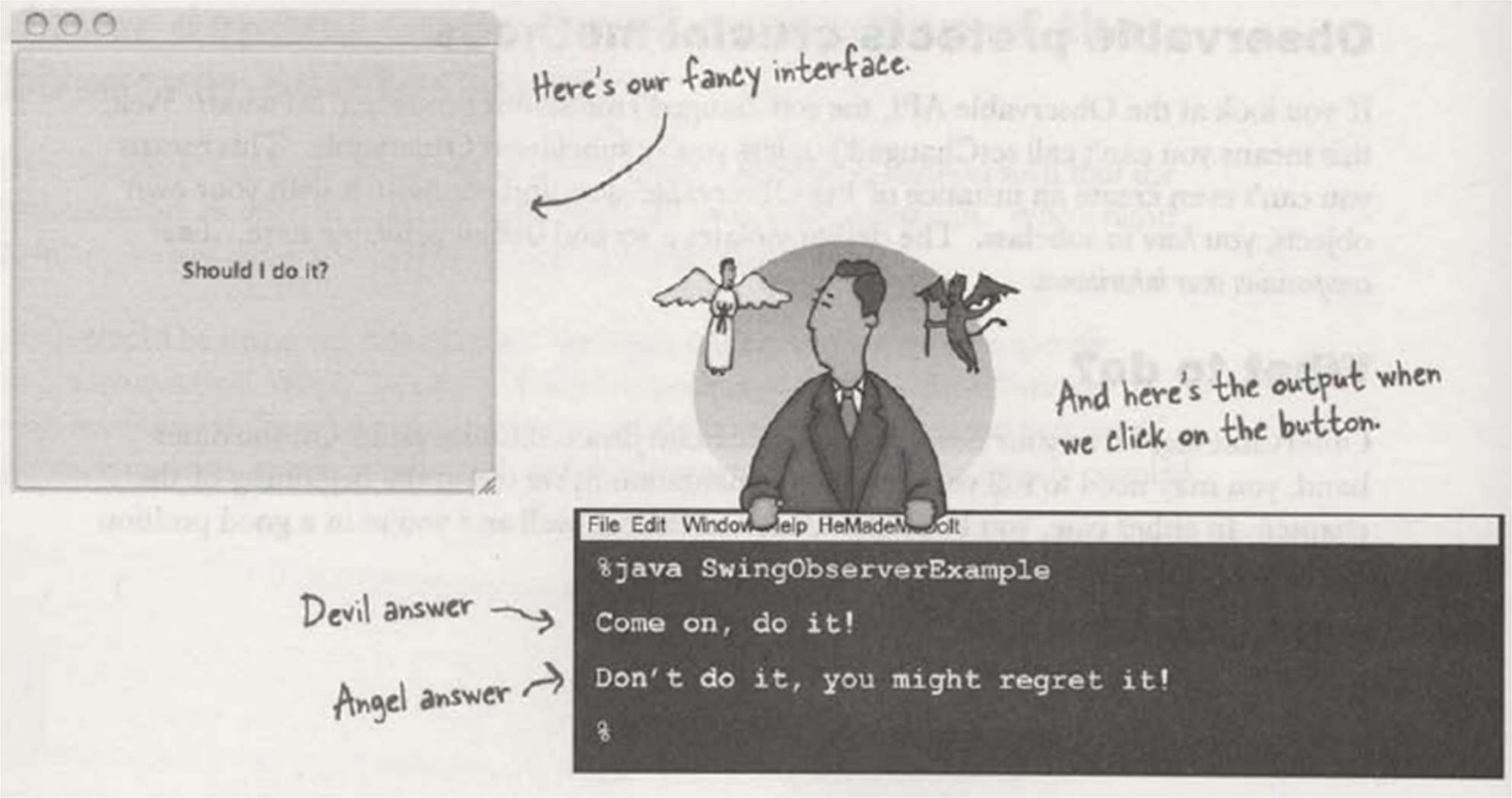
Implementing the Subject Interface

```
public class SwingObserverExample {
  JFrame frame;
  public static void main(String[] args) {
    SwingObserverExample example = new SwingObserverExample();
    example.go();
  }

  public void go() {
    frame = new JFrame();
    JButton button = new JButton("Should I do it");
    button.addActionListener(new AngelListener());
    button.addActionListener(new DevilListener());
    frame.getContentPane().add(BorderLayout.CENTER, button);
    // set frame properties here
  }

  class AngelListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      System.out.println("Don’t do it, you might regr
    }
  }
  class DevilListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      System.out.println("Come on, do it!");
    }
  }
}
```

Setting up the listeners/observers of the button.

Here are the class definitions for the observers, that implement the **`actionPerformed()`** method and gets called when the state in the subject (button) changes.

A Simple Example:
A "Life-Changing" Application



Here's our fancy interface.

And here's the output when we click on the button.

```
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

Devil answer

Angel answer

Summary

- OO Principle in play: "Strive for loosely coupled designs between objects that interact."
- Main points:
  - The Observer pattern defines a one to many relationship between objects
  - Subjects (observables), update Observers using a common interface
  - Observers are loosely coupled in that the Observable knows nothing about them, other than they implement the Observer interface.
  - You can push or pull data from the Observable when using the pattern (“pull” is considered more correct)

Summary

- Don’t depend on a specific order of notification for your Observers
- Java has several implementations of the Observer Pattern including the general purpose java.util.Observable
- Watch out for issues with java.util.Observable
- Don’t be afraid to create our own version of the Observable if needed
- Swing makes heavy use of the Observer pattern, as do many GUI frameworks
- You find this pattern in other places as well including JavaBeans and RMI.