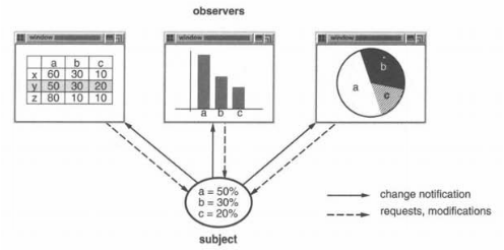
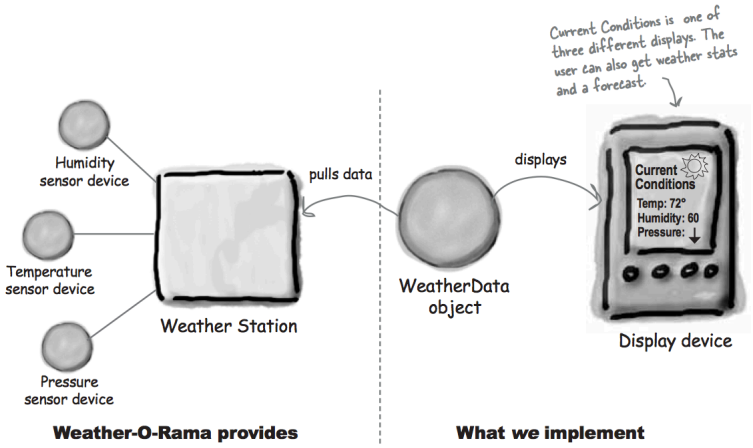
Observer

Observer

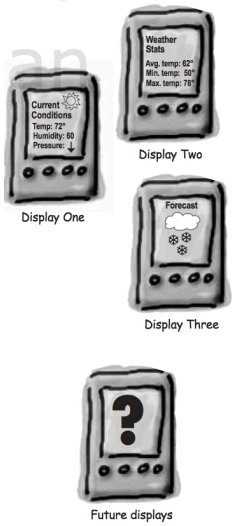
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.






```
getTemperature()
getHumidity()
getPressure()

measurementsChanged()
```



- Coding to concrete implementations, not interfaces.
- For every new display, we need to alter code.
- No way to add displays at runtime.

```
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 measuremets 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.

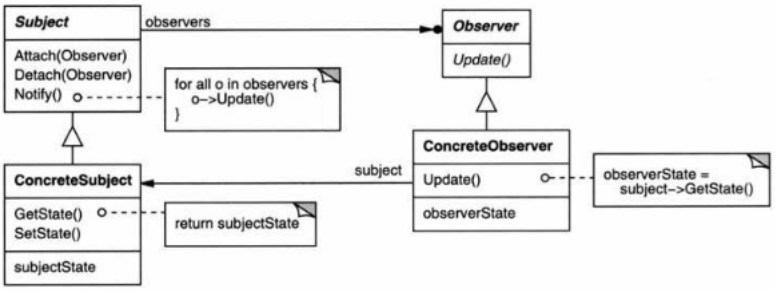
Observer

- A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state.
- The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are

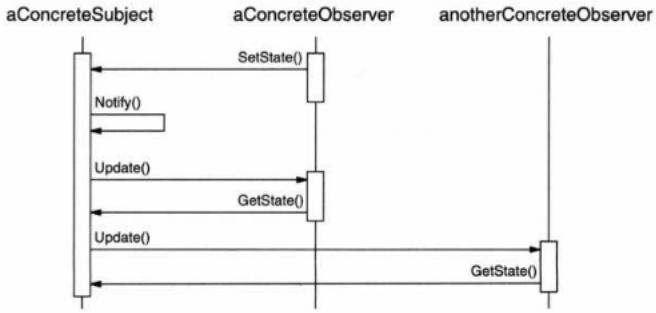
Applicability

- Use the Observer pattern when
  - 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.

Structure



Collaborations




```
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();
    }
```

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.

```
    ,

    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);
        }
    }
```

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.

```
    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.


```
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);
    }

    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.

Consequences

- Lets you add observers without modifying the subject or other observers.
- Abstract coupling between Subject and Observer.
- Support for broadcast communication.
- Unexpected/spurious updates.

Implementation

- Dangling references to deleted subjects.
  - Notify the observers that the subject has been deleted
- Updates:
  - **Push** model: the subject sends observers detailed information about the change, whether they want it or not.
  - **Pull** model: the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.

Implementation

- Observing more than one subject.
  - To distinguish between subjects, a subject sends its reference together with the update message
- Who triggers the update?
  - Have state-setting operations on Subject call Notify after they change the subject's state.
    - clients don't have to remember to call Notify on the subject.
    - several consecutive operations will cause several consecutive updates, which may be inefficient.
  - Make clients responsible for calling Notify at the right time.
    - avoids needless intermediate updates.
    - clients have an added responsibility

Implementation

- Specifying modifications of interest explicitly
  - allow registering observers only for specific events of interest.

```
void Subject::Attach(Observer*, Aspect& interest);

void Observer::Update(Subject*, Aspect& interest);
```

Change Manager

- Handle complex update semantics to avoid notifying observers more than once.