The Observer Pattern

Updated: 2/25/2012

Based on Head-First Design Patterns, Chapter 2
Overview

- Why observers are necessary
- Who generates and observes events?
- Weather Monitoring Application
- Why we use interfaces
- Java Swing Example
Why Observers are Necessary

We need to have classes that watch other classes and get notified when something interesting happens. These interesting happenings are collectively known as events.

Event-driven graphical user interfaces generate lots of events.
Who Generates Events?

We need to have classes that watch other classes and get notified when something interesting happens.

Some common event generators are:

- Device controllers
- Background services
- Network controllers
- User interface events
- Database engines
Who Can Observe an Event?

• An observer class registers itself with another class that generates events.
  – could be multiple observers for the same event
  – one class can observe multiple types of events

• Example: the .NET `Windows.Form` class
  – receives events from the MS-Windows event queue.
  – developers can write specific event handlers.
  – a Form can relay events to its child controls.
Sample *Windows.Form* events

- Load, Activated, Deactivate
- Click
- FormClosing, FormClosed
- Validating, Validated
Weather Monitoring Application

Initial design of a sample legacy system

- Weather station (physical device, acquires the weather data)
- WeatherData object – tracks the data coming from the weather station, updates the displays.
- Display device
  - we want it to output to three different displays
Weather-O-Rama provides

What we implement

Weather Station

pulls data

WeatherData object

displays

Display device

Humidity sensor device

Temperature sensor device

Pressure sensor device

Current Conditions
Temp: 72°
Humidity: 60
Pressure: ↓
A naive approach codes directly to concrete objects:

```java
public class WeatherData{
    private float temperature;
    private float humidity;
    private float pressure;
    private CurrentConditionsDisplay currentConditionsDisplay;
    private StatisticsDisplay statisticsDisplay;
    private ForecastDisplay forecastDisplay;

    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);
    }
}
```
Subjects and Observers

The Observer pattern is like subscribing to a newspaper:

• A publisher class makes data available. We call its class the *subject* class.

• A subscriber asks the publisher to send it data. It is the *observer* class.
Sending Notifications

• When the subject detects a change in its data, it sends a notification to all of its registered observers.

• Observers can unsubscribe at any time while the program is running.
Technical Definition

• 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.

You have mail!
Interfaces Are Observers

- Don't define subject and observers as concrete classes.
Why We Use Interfaces

• A subject class only knows that an observer class implements a certain interface.
• You never need to modify a subject class when adding new types of observers.
• You can reuse subjects or observers independently of each other.
• Changes to subject or observer classes will not affect each other.
Redesigning the WeatherStation
Java Swing Example

- JFrame object is a container for GUI objects such as text boxes and buttons.
- An class that observes (receives) messages implements the `ActionListener` interface.
- A Button object sends notifications to its observers.
- The `Button.addActionListener` method registers an observer.
- The `actionPerformed` method (in the ActionListener interface) executes when the button sends a click message.