References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 293-303


Java API

VisualWorks Smalltalk API
Observer

One-to-many dependency between objects

When one object changes state,
   all its dependents are notified and updated automatically
Structure

```
subject
  └── observer A
  └── observer B

subject
  ┌── GetState() ─── Update() ─── Notify() ─── Attach(Observer)
  │                           └── Detach(Observer)
  └── Detach(Observer)

observer A
  └── Update()

observer B
  └── Update()

ConcreteSubject
  └── GetState()

ConcreteObserver
  └── Update()

Observer
  └── Update()

observers ──── observers

subjectState ─── subjectState
```

```
Pseudo Java Example

public class Subject {
    Window display;
    public void someMethod() {
        this.modifyMyStateSomeHow();
        display.addText( this.text() );
    }
}

public class Subject {
    ArrayList observers = new ArrayList();
    public void someMethod() {
        this.modifyMyStateSomeHow();
        changed();
    }

    private void changed() {
        Iterator needsUpdate = observers.iterator();
        while (needsUpdate.hasNext() )
            needsUpdate.next().update( this );
    }
}

public class SampleWindow {
    public void update(Object subject) {
        text = ((Subject) subject).getText();
        Thread.sleep(10000).
    }
}

Abstract coupling - Subject and Observer

Broadcast communication

Updates can take too long
Smalltalk Implementation
   Object implements methods for both Observer and Subject.

   Actual Subjects should subclass Model
Counter Example

- Increase Detector
- Decrease Detector
- Change Detector

Counter
Smalltalk Example

Smalltalk.CS635 defineClass: #Counter
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'count '
  classInstanceVariableNames: ''
  imports: ''
  category: 'Observer Examples'

  CS635.Counter class methods
new
  ^super new initialize

  CS635.Counter instance methods
decrease
  count := count - 1.
  self changed: #decrease

  increase
  count := count + 1.
  self changed: #increase

  initialize
  count := 0

  printOn: aStream
    aStream
      nextPutAll: count printString
### Count Observer

Smalltalk-CS635 defineClass: #IncreaseDetector
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'model '
  classInstanceVariableNames: "
  imports: "
  category: 'Observer Examples'

#### class methods

on: aCounter
  | detector |
  detector := super new.
  aCounter addDependent: detector.
  ^detector

#### instance methods

update: anAspectSymbol with: aParameter from: aSender
  anAspectSymbol = #increase ifTrue:
    [Transcript
      show: 'Count is now:' , aSender printString;
      cr]

| counter |
| counter := Counter new.
IncreaseDetector on: counter.
IncreaseDetector on: counter.
counter
  increase;
  decrease;
  decrease;
  increase
Java's Observer

<table>
<thead>
<tr>
<th>Java</th>
<th>Observer Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Observer</td>
<td>Abstract Observer class</td>
</tr>
<tr>
<td>Observable class</td>
<td>Subject class</td>
</tr>
</tbody>
</table>

**Class java.util.Observable**

Observable object may have any number of Observers

Whenever the Observable instance changes, it notifies all of its observers

Notification is done by calling the update() method on all observers.

**Interface java.util.Observer**

Allows all classes to be observable by instances of class Observer
class Counter extends Observable {
    public static final String INCREASE = "increase";
    public static final String DECREASE = "decrease";
    private int count = 0;
    private String label;

    public Counter( String label ) { this.label = label; }

    public String label() { return label; }
    public int value() { return count; }
    public String toString() { return String.valueOf( count );}

    public void increase() {
        count++;
        setChanged();
        notifyObservers( INCREASE );
    }

    public void decrease() {
        count--;
        setChanged();
        notifyObservers( DECREASE );
    }
}
Java Observer

class IncreaseDetector implements Observer {
    public void update( java.util.Observable whatChanged,
                        java.lang.Object message) {
        if ( message.equals( Counter.INCREASE) ) {
            Counter increased = (Counter) whatChanged;
            System.out.println( increased.label() + " changed to " +
                                increased.value());
        }
    }
}

Implementation Issues
Mapping subjects (Observables) to observers

Use list in subject
Use hash table

```java
public class Observable {
    private boolean changed = false;
    private Vector obs;

    public Observable() {
        obs = new Vector();
    }

    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }
}
```
Observing more than one subject

If an observer has more than one subject how does it know which one changed?

Pass information in the update method

```plaintext
update: anAspectSymbol with: aParameter from: aSender
    anAspectSymbol = #increase ifTrue: [Transcript
        show: 'Count is now: ' , aSender printString;
        cr]
```
Deleting Subjects

In C++ the subject may no longer exist

Java/Smalltalk observer may prevent subject from garbage collection
Who Triggers the update?

Have methods that change the state trigger update

class Counter extends Observable {
    public void increase() {
        count++;
        setChanged();
        notifyObservers( INCREASE );
    }
}

Have clients call Notify at the right time

class Counter extends Observable {
    public void increase() {
        count++;
    }
}

Counter pageHits = new Counter();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.notifyObservers();
Make sure Subject is self-consistent before Notification

class ComplexObservable extends Observable {
    Widget frontPart = new Widget();
    Gadget internalPart = new Gadget();

    public void trickyChange() {
        frontPart.widgetChange();
        internalPart.anotherChange();
        setChanged();
        notifyObservers();
    }
}

class MySubclass extends ComplexObservable {
    Gear backEnd = new Gear();

    public void trickyChange() {
        super.trickyChange();
        backEnd.yetAnotherChange();
        setChanged();
        notifyObservers();
    }
}
Adding information about the change

push models - add parameters in the update method

class IncreaseDetector extends Counter implements Observer { // stuff not shown

    public void update( Observable whatChanged, Object message) {
        if ( message.equals( INCREASE) )
            increase();
    }
}

class Counter extends Observable { // some code removed
    public void increase() {
        count++;
        setChanged();
        notifyObservers( INCREASE );
    }
}
Adding information about the change

pull model - observer asks Subject what happened

class IncreaseDetector extends Counter implements Observer {
    public void update( Observable whatChanged ) {
        if ( whatChanged.didYouIncrease() )
            increase();
    }
}

class Counter extends Observable {
    public void increase() {
        count++;
        setChanged();
        notifyObservers();
    }
}
Scaling the Pattern
Java Event Model

AWT/Swing components broadcast events to Listeners

JDK1.0 AWT components broadcast an event to all its listeners

A listener normally not interested all events

Broadcasting to all listeners was too slow with many listeners
Java 1.1+ Event Model

Each component supports different types of events:

Component supports
  ComponentEvent  FocusEvent
  KeyEvent  MouseEvent

Each event type supports one or more listener types:

  MouseEvent
  MouseListener  MouseMotionListener

Each listener interface replaces update with multiple methods

  MouseListener
    mouseClicked()  mouseEntered()
    mousePressed()  mouseReleased()

Listeners
  Only register for events of interest
  Don't need case statements to determine what happened
Small Models

Often an object has a number of fields (aspects) of interest to observers.

Rather than make the object a subject, make the individual fields subjects.
- Simplifies the main object.
- Observers can register for only the data they are interested in.

VisualWorks ValueHolder

Subject for one value.

ValueHolder allows you to:

- Set/get the value
  - Setting the value notifies the observers of the change.

- Add/Remove dependents.