CS 683 Emerging Technologies
Spring Semester, 2003
Doc 7 Aspects & Observer Pattern
Contents

References

Design Pattern Implementation in Java and AspectJ, Hannemann & Kiczales, OOPSAL 2002, Seattle Washington Conference proceedings, Available at http://www.cs.ubc.ca/~jan/AODPs/

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

Source Code for examples in Design Pattern Implementation in Java and AspectJ, See http://www.cs.ubc.ca/~jan/AODPs/ do download the examples

Notice

This document contains source code written and copy written by Jan Hannemann and Gregor Kiczales. The code used is their Observer example from Design Pattern Implementation in Java and AspectJ, Hannemann & Kiczales, OOPSAL 2002, Seattle Washington Conference proceedings. The paper and source code is available at http://www.cs.ubc.ca/~jan/AODPs/.

The source code here as been modified to fit onto slides for display in a classroom. The edits include removing comments and some minor reformatting to conserve space.

Coupling & Cohesion

- Coupling

  Strength of interaction between objects in system

- Cohesion

  Degree to which the tasks performed by a single module are functionally related

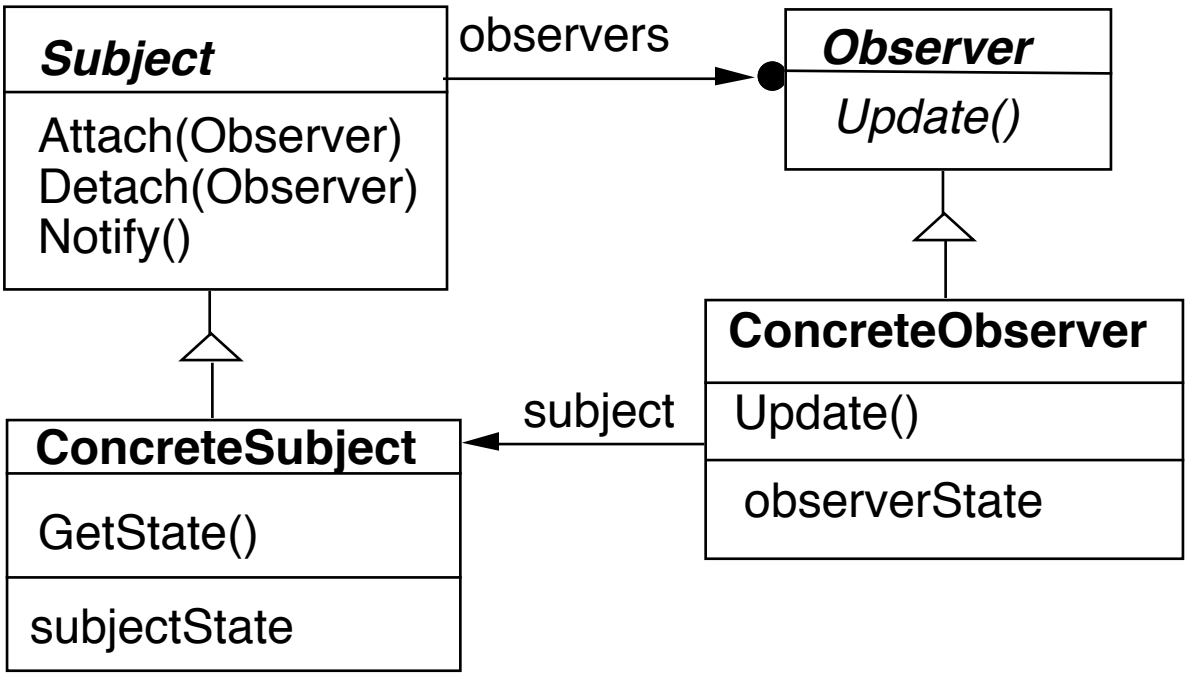
Observer Pattern

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

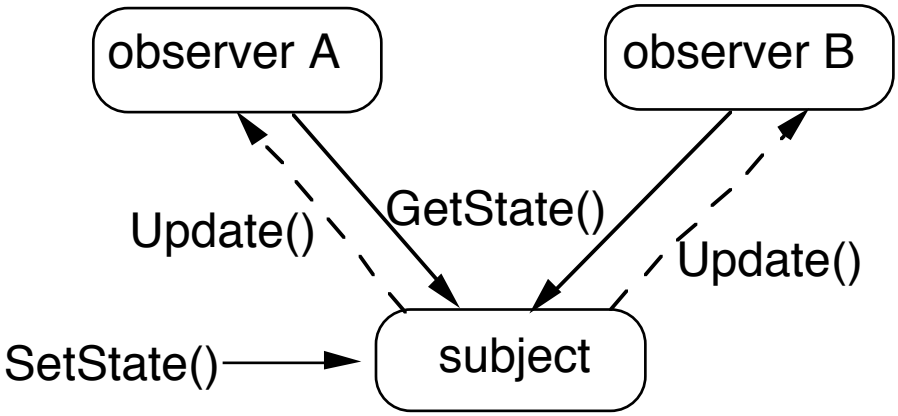
Use the Observer pattern:

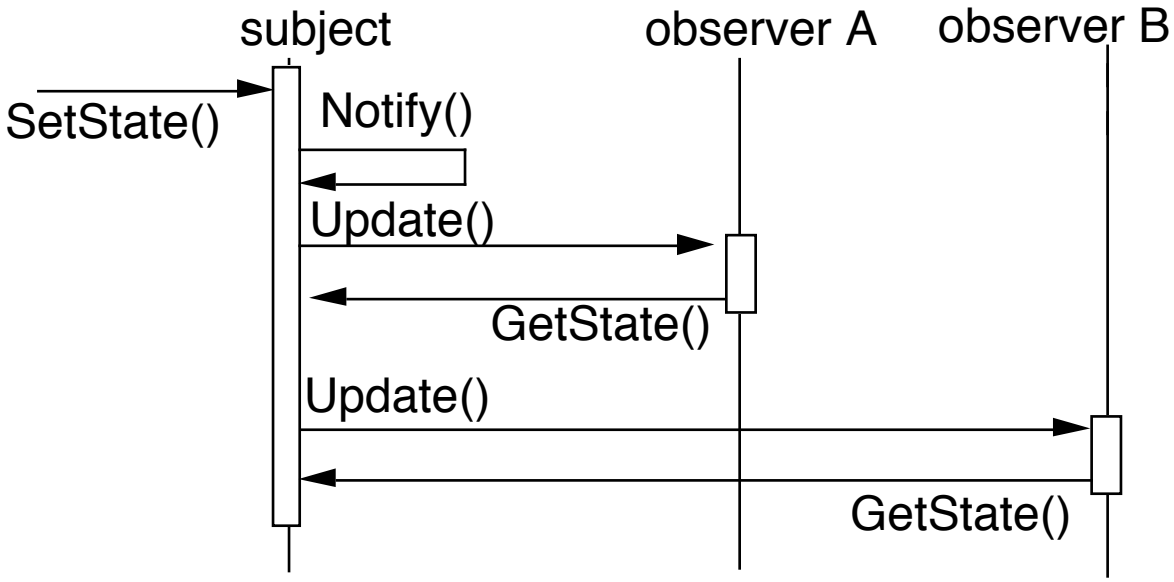
- When an abstraction has two aspects, one dependent on the other.
- When a change to one object requires changing others, and you don't how many objects need to be changed
- When an object should be able to notify other objects without making assumptions about who these objects are.

Structure


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
ConcreteObserver
Update()
observerState
subject
ConcreteSubject
GetState()
subjectState

Collaborations





Simple Example
Replace

Note example does not use legal Java

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

With

```
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();
      etc.
   }
}
```

Consequences

- Abstract coupling between Subject and Observer
- Support for broadcast communication
- Unexpected updates

  Simple change in subject can cause numerous updates, which can be expensive or distracting

- Updates can take too long

  Subject cannot perform any work until all observers are done

Point Example Without Aspects

Point that is to be displayed on the screen

Observer

```
public interface Observer {
   public void update(Subject s);
}
```

Subject

```
public interface Subject {

   public void attach(Observer o);
   public void detach(Observer o);
   public void notifyObservers();
}
```

Point

```
import java.awt.Color;
import java.util.HashSet;
import java.util.Iterator;

public class Point implements Subject {

    private HashSet observers;

    private int x;
    private int y;
    private Color color;

    public Point(int x, int y, Color color) {
        this.x=x;
        this.y=y;
        this.color=color;
        this.observers = new HashSet();
    }

    public int getX() { return x; }

    public int getY() { return y; }

    public void setX(int x) {
        this.x=x;
        notifyObservers();
    }
```

Point Continued

```
    public void setY(int y) {
        this.y=y;
        notifyObservers();
    }

    public Color getColor() { return color; }

    public void setColor(Color color) {
        this.color=color;
        notifyObservers();
    }

    public void attach(Observer o) {
        this.observers.add(o);
    }

    public void detach(Observer o) {
        this.observers.remove(o);
    }

    public void notifyObservers() {
        for (Iterator e = observers.iterator() ; e.hasNext() ;) {
            ((Observer)e.next()).update(this);
        }
    }
}
```

Screen

```
import java.util.HashSet;
import java.util.Iterator;

public class Screen implements Subject, Observer {

    private HashSet observers;

    private String name;

    public Screen(String s) {
        this.name = s;
        observers = new HashSet();
    }

    public void display (String s) {
       System.out.println(name + ": " + s);
       notifyObservers();
    }

    public void attach(Observer o) {
        this.observers.add(o);
    }

    public void detach(Observer o) {
        this.observers.remove(o);
    }
```

Screen Continued

```
    public void notifyObservers() {
        for (Iterator e = observers.iterator() ; e.hasNext() ;) {
            ((Observer)e.next()).update(this);
        }
    }

    public void update(Subject s) {

        display("update received from a "+s.getClass().getName()+
            " object");
    }
}
```

Sample Program

```
public static void main(String argv[]) {

   Point p = new Point(5, 5, Color.blue);

   System.out.println("Creating Screen s1,s2,s3,s4,s5 and Point p");

   Screen s1 = new Screen("s1");
   Screen s2 = new Screen("s2");

   Screen s3 = new Screen("s3");
   Screen s4 = new Screen("s4");

   Screen s5 = new Screen("s5");

   p.attach(s1);
   p.attach(s2);

   p.attach(s3);
   p.attach(s4);

   s2.attach(s5);
   s4.attach(s5);

   p.setColor(Color.red);

   p.setX(4);
}
```

Point Example Without Aspects

Point

```
import java.awt.Color;

public class Point implements Subject {

    private int x;
    private int y;
    private Color color;

    public Point(int x, int y, Color color) {
        this.x=x;
        this.y=y;
        this.color=color;
    }

    public int getX() { return x; }

    public int getY() { return y; }

    public void setX(int x) {       this.x=x;    }

    public void setY(int y) {      this.y=y;    }

    public Color getColor() { return color; }

    public void setColor(Color color) {     this.color=color;   }

}
```

Screen

```
public class Screen {

    private String name;


    public Screen(String s) {
        this.name = s;
    }

    public void display (String s) {
        System.out.println(name + ": " + s);
    }
}
```

ObserverProtocol

```
import java.util.WeakHashMap;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

public abstract aspect ObserverProtocol {

    protected interface Subject  { }

    protected interface Observer { }

    private WeakHashMap perSubjectObservers;

    protected List getObservers(Subject s) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers = (List)perSubjectObservers.get(s);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(s, observers);
        }
        return observers;
    }


    public void    addObserver(Subject s, Observer o) {
        getObservers(s).add(o);
    }
```

ObserverProtocol Continued

```
    public void removeObserver(Subject s, Observer o) {
        getObservers(s).remove(o);
    }

    protected abstract pointcut subjectChange(Subject s);
    protected abstract void updateObserver(Subject s, Observer o);

    after(Subject s): subjectChange(s) {
        Iterator iter = getObservers(s).iterator();
        while ( iter.hasNext() ) {
            updateObserver(s, ((Observer)iter.next()));
        }
    }
}
```

ColorObserver

```
import java.awt.Color;

public aspect ColorObserver extends ObserverProtocol{


  declare parents: Point  implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
      call(void Point.setColor(Color)) && target(s);

  protected void updateObserver(Subject s, Observer o) {
     ((Screen)o).display("Screen updated because color changed.");
  }
}
```

CoordinateObserver

```
public aspect CoordinateObserver extends ObserverProtocol{

  declare parents: Point  implements Subject;

  declare parents: Screen implements Observer;

  protected pointcut subjectChange(Subject s):
    ( call(void Point.setX(int)) ||
      call(void Point.setY(int)) ) && target(s);

  protected void updateObserver(Subject s, Observer o) {
      ((Screen)o).display("Screen updated as coordinates changed.");
  }
}
```

Sample Program

```
  public static void main(String argv[]) {

    Point p = new Point(5, 5, Color.blue);

    System.out.println("Creating Screen s1,s2,s3,s4,s5 and Point p");

    Screen s1 = new Screen("s1");
    Screen s2 = new Screen("s2");

    Screen s3 = new Screen("s3");
    Screen s4 = new Screen("s4");

    Screen s5 = new Screen("s5");

    ColorObserver.aspectOf().addObserver(p, s1);
    ColorObserver.aspectOf().addObserver(p, s2);

    CoordinateObserver.aspectOf().addObserver(p, s3);
    CoordinateObserver.aspectOf().addObserver(p, s4);

    ScreenObserver.aspectOf().addObserver(s2, s5);
    ScreenObserver.aspectOf().addObserver(s4, s5);

    p.setColor(Color.red);

    p.setX(4);

  }
}
```

Important Features

- Locality

  Point does not have any Observer/Subject code
  All Observer/Subject code is localized in the aspects

- Reusability

  ObserverProtocol can be used any Observer pattern instance

- Composition Transparency

  Subjects & Observers can be in multiple observer/subject relationships without their code becoming more complex

- Pluggability

  Subjects and observers can be used with or without the pattern