The Observer
Design Pattern

For details see Gamma et al. in “Design Patterns”
Observer Design Pattern

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

Alternative Implementation using AspectJ
The Observer Design Pattern
Alternative Implementation using AspectJ

• We want to...

• avoid the decision between Push or Pull mode observers

• better support observers interested only in specific events
The Observer Design Pattern

Alternative Implementation using AspectJ

Parts Common to Potential Instantiations of the Pattern

1. The existence of Subject and Observer roles (i.e. the fact that some classes act as Observers and some as Subjects)
2. Maintenance of a mapping from Subjects to Observers
3. The general update logic: Subject changes trigger Observer updates

Parts Specific to Each Instantiation of the Pattern

4. Which classes can be Subjects and which can be Observers
5. A set of changes of interest on the Subjects that trigger updates on the Observers
6. The specific means of updating each kind of Observer when the update logic requires it

Will be implemented in a reusable ObserverProtocol aspect.
The Observer Design Pattern
Alternative Implementation using AspectJ

public abstract aspect ObserverProtocol {
    // Realization of the Roles of the Observer Design Pattern
    protected interface Subject { }
    protected interface Observer { }

    ...
}
The Observer Design Pattern

Alternative Implementation using AspectJ

```java
public abstract aspect ObserverProtocol {
    ... 
    // Mapping and Managing Subjects and Observers
    private WeakHashMap<Subject, List<Observer>> perSubjectObservers;
    protected List<Observer> getObservers(Subject s) {
        if (perSubjectObservers == null)
            perSubjectObservers = new WeakHashMap<Subject, List<Observer>>();
        List<Observer> observers = perSubjectObservers.get(s);
        if (observers == null) {
            observers = new LinkedList<Observer>();
            perSubjectObservers.put(s, observers);
        }
        return observers;
    }
    public void addObserver(Subject s, Observer o) {
        getObservers(s).add(o);
    }
    public void removeObserver(Subject s, Observer o) {
        getObservers(s).remove(o);
    }
    ... 
}
The Observer Design Pattern
Alternative Implementation using AspectJ

```java
public abstract aspect ObserverProtocol {
    ...
    // Notification related functionality
    abstract protected pointcut subjectChange(Subject s);

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

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

The part common to instantiations of the pattern.
The Observer Design Pattern

Alternative Implementation using AspectJ - Example
The Observer Design Pattern

Alternative Implementation using AspectJ - Example

Task: Observe Changes of the Color

```java
public aspect ColorObserver extends ObserverProtocol {

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

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

    protected void updateObserver(Subject s, Observer o) {
        ((Screen)o).display("Color change.");
    }
}
```

To create a mapping between an Observer and a Subject:

ColorObserver.aspectOf().addObserver(P, S);
The Observer Design Pattern

Alternative Implementation using AspectJ - Assessment

• **Locality**
  All code that implements the Observer pattern is in the abstract and concrete observer aspects, none of it is in the participant classes; there is no coupling between the participants. Potential changes to each Observer pattern instance are confined to one place.

• **Reusability**
  The core pattern code is abstracted and reusable. The implementation of ObserverProtocol is generalizing the overall pattern behavior. The abstract aspect can be reused and shared across multiple Observer pattern instances.
The Observer Design Pattern
Alternative Implementation using AspectJ - Assessment

• **Composition transparency**
  Because a pattern participant’s implementation is not coupled to the pattern, if a Subject or Observer takes part in multiple observing relationships their code does not become more complicated and the pattern instances are not confused. Each instance of the pattern can be reasoned about independently.

• **(Un)pluggability**
  It is possible to switch between using a pattern and not using it in the system.
Observer Design Pattern

How it is implemented depends on the available programming language mechanisms; the consequences may also change!