CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005
Doc 10 Observer

Contents

References

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

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 305-326

Java API

VisualWorks Smalltalk API

Observer

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

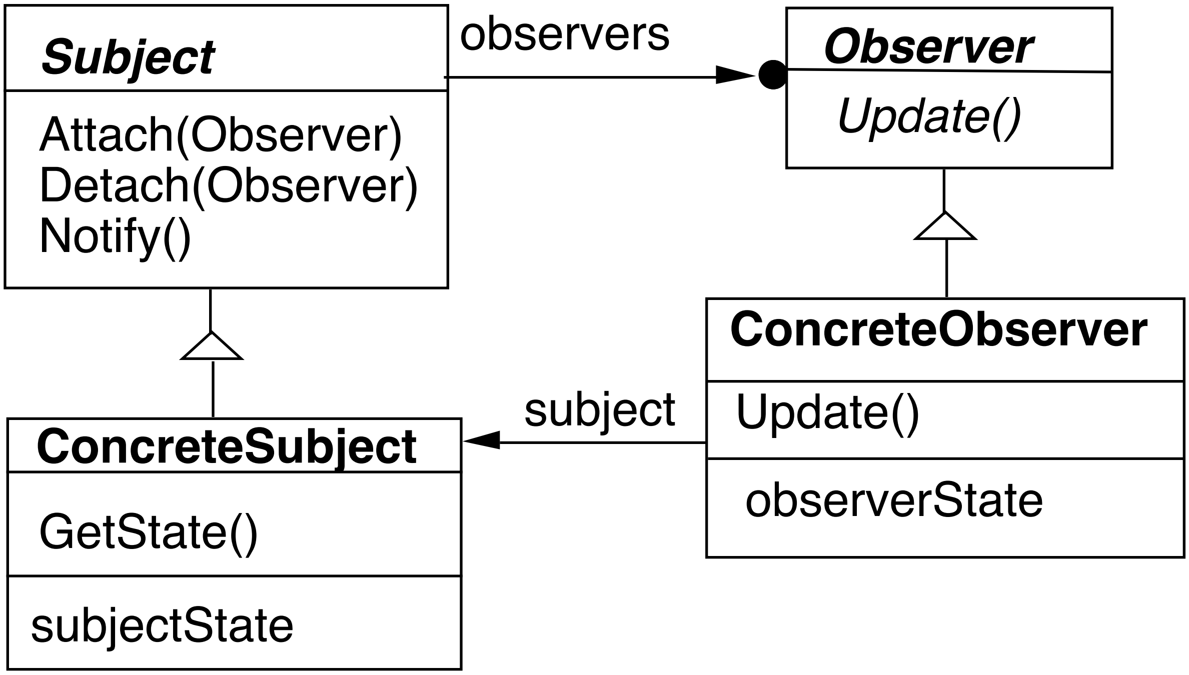
Use the Observer pattern:

- When an abstraction has two aspects, one dependent on the other.

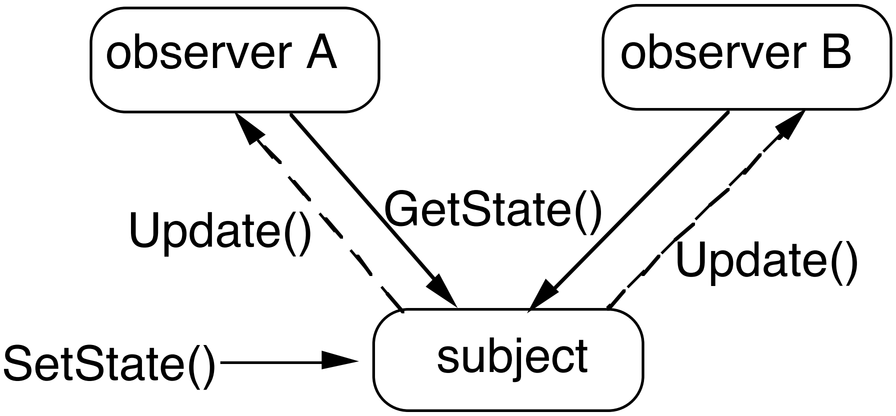
- When a change to one object requires changing others, and you don't how many objects need to be changed

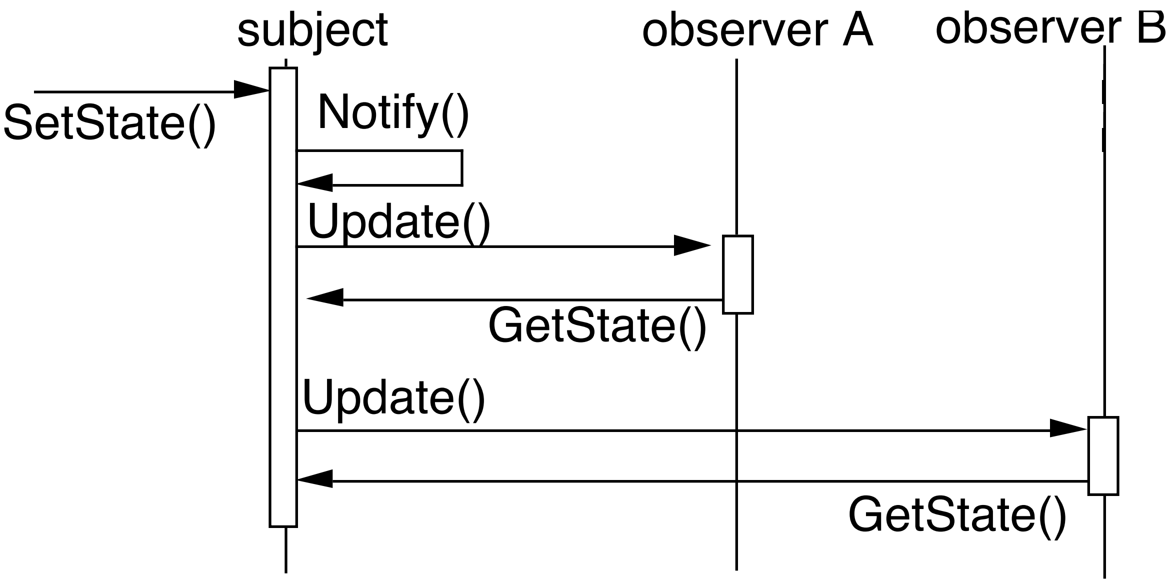
- When an object should be able to notify other objects without making assumptions about who these objects are.

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 work until all observers are done

Smalltalk Implementation

Smalltalk	Java	Observer Pattern
Object	Observer	Abstract Observer class
Object & Model	Observable	Subject class

Object implements methods for both Observer and Subject.

Actual Subjects should subclass Model

- Model handles dependents better

Object methods

update: anAspectSymbol
update: anAspectSymbol with: aParameter
update: anAspectSymbol with: aParameter from: aSender
   Receive an update message from a Model(Subject)

changed
changed: anAspectSymbol
changed: anAspectSymbol with: aParameter
   Receiver changed.

addDependent: anObject

removeDependent: anObject

dependents
   return collection of all dependents

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: #IncreaseDectector
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'model '
   classInstanceVariableNames: ''
   imports: ''
   category: 'Observer Examples'
```

CS635.IncreaseDectector class methods

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

CS635.IncreaseDectector instance methods

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

Smalltalk Example - Continued

```
| counter |
counter := Counter new.
IncreaseDetector on: counter.
counter
   increase;
   decrease;
   decrease;
   increase
```

Java’s Implementation

Java API implements a framework for this pattern

Java	Observer Pattern
Interface Observer	Abstract Observer class
Observable class	Subject class

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

When implemented, this interface allows all classes to be observable by instances of class Observer

java.util.Observable Methods

addObserver(Observer)
Adds an observer to the observer list.

clearChanged()
Clears an observable change.

countObservers()
Counts the number of observers.

deleteObserver(Observer)
Deletes an observer from the observer list.

deleteObservers()
Deletes observers from the observer list.

hasChanged()
Returns a true boolean if an observable change has occurred.

notifyObservers()
Notifies all observers if an observable change occurs.

notifyObservers(Object)
Notifies all observers of the specified observable change which occurred.

setChanged()
Sets a flag to note an observable change.

Interface java.util.Observer

update
Called when observers in the observable list need to be updated

A Java Example

```
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 );
      }
   }
```

```
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());
           }
       }
   }
```

```
abstract class CounterButton extends Button
  {
  protected Counter count;

  public CounterButton( String buttonName, Counter count )
     {
     super( buttonName );
     this.count = count;
     }

  public boolean action( Event processNow, Object argument )
     {
     changeCounter();
     return true;
     }

  abstract protected void changeCounter();
  }
```

```
class IncreaseButton extends CounterButton
  {
  public IncreaseButton( Counter count )
     {
     super( "Increase", count );
     }

  protected void changeCounter()
     {
     count.increase();
     }
  }
```

```
class DecreaseButton extends CounterButton
  {
  public DecreaseButton( Counter count )
     {
     super( "Decrease", count );
     }

  protected void changeCounter()
     {
     count.decrease();
     }
  }
```

```
class ButtonController extends Frame
  {
  public ButtonController( Counter model, int x, int y,
                        int width, int height )
     {
     setTitle( model.label() );
     reshape(x, y,  width, height );
     setLayout( new FlowLayout() );

     // buttons to change counter
     add( new IncreaseButton( model ));
     add( new DecreaseButton( model ));
     show();
     }
  }
```

Sample Program

```
class TestButton
  {
  public  static  void  main( String  args[] ){
     Counter x = new Counter( "x" );
     Counter y = new Counter( "y" );

     IncreaseDetector plus = new IncreaseDetector(  );
     x.addObserver( plus );
     y.addObserver( plus );

     new ButtonController( x,  30, 30, 150, 50 );
     new ButtonController( y,  30, 100, 150, 50 );
     }
```

Implementation Issues
Mapping subjects(Observables) to observers

Use list in subject
Use hash table

```
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

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

Dangling references to Deleted Subjects

In C++ the subject may no longer exist

In Java/Smalltalk the subject will exists as long as reference exists

> Observer reference to Subject may keep Subject around after Subject is not needed

Who Triggers the update?

- Have methods that change the state trigger update

```
class Counter extends Observable
   {     // some code removed
   public void increase()
      {
      count++;
      setChanged();
      notifyObservers( INCREASE );
      }
   }
```

If there are several of changes at once, you may not want each change to trigger an update

It might be inefficient or cause too many screen updates

- Have clients call Notify at the right time

```
class Counter extends Observable
      { // some code removed
      public void increase()  {    count++;   }
      }

Counter pageHits = new Counter();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.notifyObservers();
```

Make sure Subject is self-consistent before Notification

Here is an example of the problem

```
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( );
    }
  }
```

A Template Method Solution to the Problem

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

  public void trickyChange()
     {
     doThisChangeWithFactoryMethod();
     setChanged();
     notifyObservers( );
     }

  private void doThisChangeWithTemplateMethod()
     {
     frontPart.widgetChange();
     internalpart.anotherChange();
     }
  }

class MySubclass extends ComplexObservable
  {
  Gear backEnd = new Gear();
  private void doThisChangeWithTemplateMethod()
     {
     super. DoThisChangeWithTemplateMethod();
     backEnd.yetAnotherChange();
     }
  }
```

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
  { // stuff not shown

  public void update( Observable whatChanged )
    {
    if ( whatChanged.didYouIncrease() )
      increase();
    }
  }

  class Counter extends Observable
    {    // some code removed
    public void increase()
      {
      count++;
      setChanged();
      notifyObservers( );
      }
    }
```

Scaling the Pattern

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 supports
MouseListener
MouseMotionListener

Each listener interface replaces update with multiple methods

MouseListener interface has:
mouseClicked()
mouseEntered()
mousePressed()
mouseReleased()

A mouse listener (observer) has to implement all 4 methods

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

Adapting Observers

An observer implements an update method

A concrete observer represents an abstraction

Update() may be out of place in this abstraction

Use an adapter to map update() method to a different method in the concrete observer

VisualWorks Smalltalk has a built-in adapter
DependencyTransformer