

CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 12 Observer & Prototype
March 9, 2009

Copyright ©, All rights reserved. 2009 SDSU & Roger Whitney, 5500
Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent ([http://
www.opencontent.org/opl.shtml](http://www.opencontent.org/opl.shtml)) license defines the copyright on this
document.

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 293-303, 117-126

Java API

VisualWorks Smalltalk API

Prototype-based Languages

http://en.wikipedia.org/wiki/Prototype-based_programming

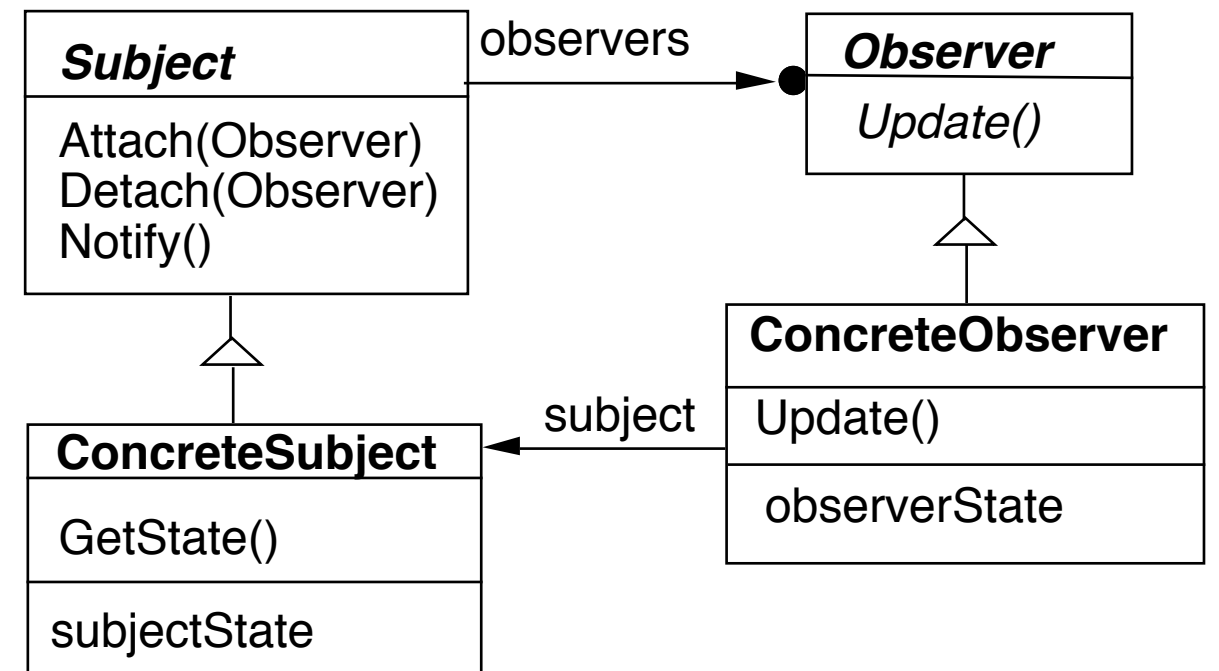
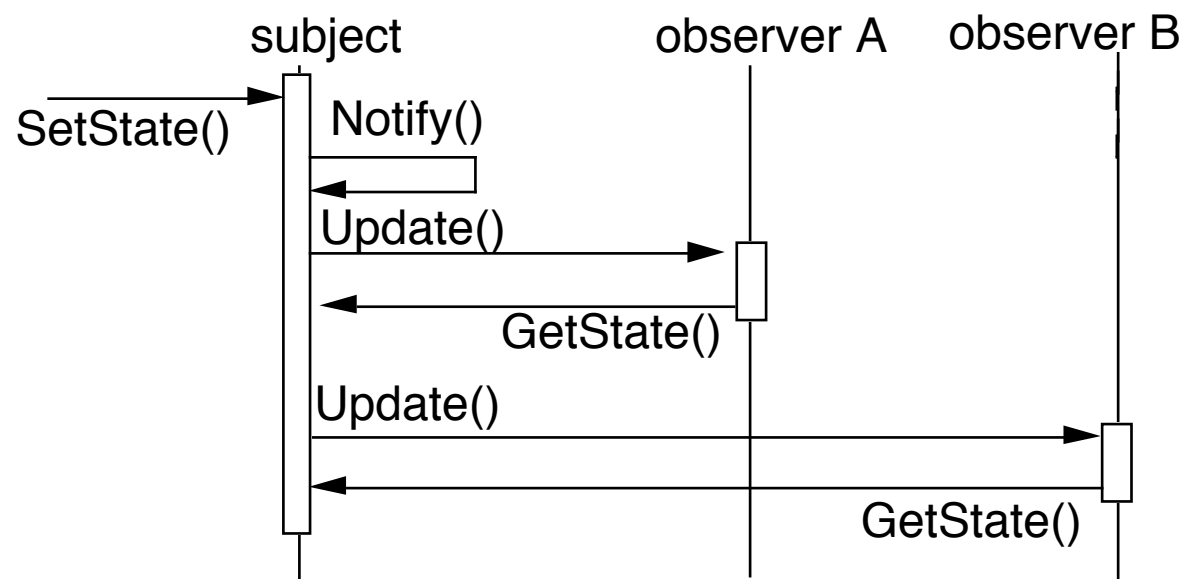
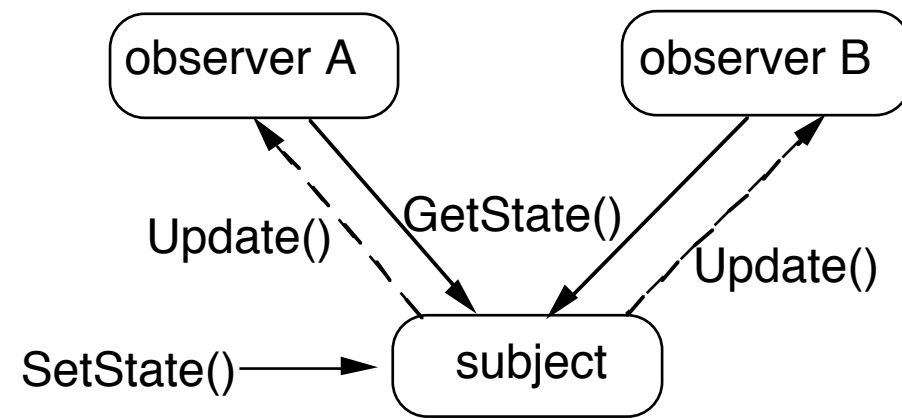
JavaScript The Definite Guide 4'th Ed, Flanagan, O'Reilly Press, 2002

Observer

One-to-many dependency between objects

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

Structure



Pseudo Java Example

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

Abstract coupling - Subject and Observer

Broadcast communication

Updates can take too long

```
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).  
    }  
}
```

Some Language Support

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

Smalltalk Implementation

Object implements methods for both Observer and Subject.

Actual Subjects should subclass Model

Java's Observer

Class java.util.Observable

Java	Observer Pattern
Interface Observer	Abstract Observer class
void addObserver(Observer o)	Subject class
void clearChanged()	
int countObservers()	
void deleteObserver(Observer o)	
void deleteObservers()	
boolean hasChanged()	
void notifyObservers()	
void notifyObservers(Object arg)	
void setChanged()	

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

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 );
    }
}
```


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());
        }
    }

    public static void main(String[] args) {
        Counter test = new Counter();
        IncreaseDetector adding = new IncreaseDetector();
        test.addObserver(adding);
        test.increase();
    }
}
```

Ruby Example

```
require 'observer'
```

```
class Counter
```

```
  include Observable
```

```
  attr_reader :count
```

```
  def initialize
```

```
    @count = 0
```

```
  end
```

```
  def increase
```

```
    @count += 1
```

```
    changed
```

```
    notify_observers(:INCREASE)
```

```
  end
```

```
  def decrease
```

```
    @count -= 1
```

```
    changed
```

```
    notify_observers(:DECREASE)
```

```
  end
```

```
end
```

```
class IncreaseDetector
```

```
  def update(type)
```

```
    if type == :INCREASE
```

```
      puts('Increase')
```

```
    end
```

```
  end
```

```
end
```

```
count = Counter.new()
```

```
puts count.count
```

```
count.add_observer(IncreaseDetector.new)
```

```
count.increase
```

```
count.increase
```

```
puts count.count
```

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

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

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();
```

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 {           // some code removed  
    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

KeyEvent

FocusEvent

MouseEvent

Each event type supports one or more listener types:

MouseEvent

MouseListener

MouseMotionListener

Each listener interface replaces update with multiple methods

MouseListener

mouseClicked()

mousePressed()

mouseEntered()

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

Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

Applicability

Use the Prototype pattern when

- A system should be independent of how its products are created, composed, and represented; and

- When the classes to instantiate are specified at run-time; or

- To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

- When instances of a class can have one of only a few different combinations of state.

Insurance Example

Insurance agents start with a standard policy and customize it

Two basic strategies:

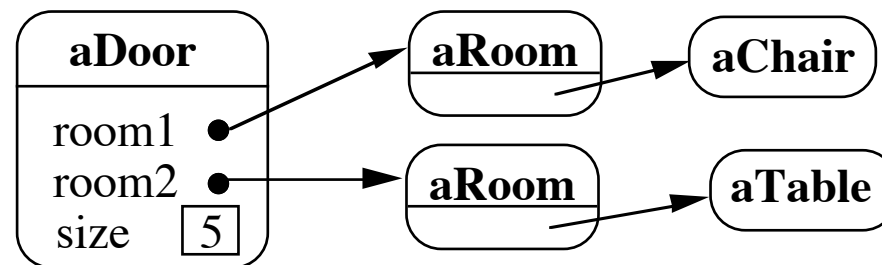
Copy the original and edit the copy

Store only the differences between original and the customize version in a decorator

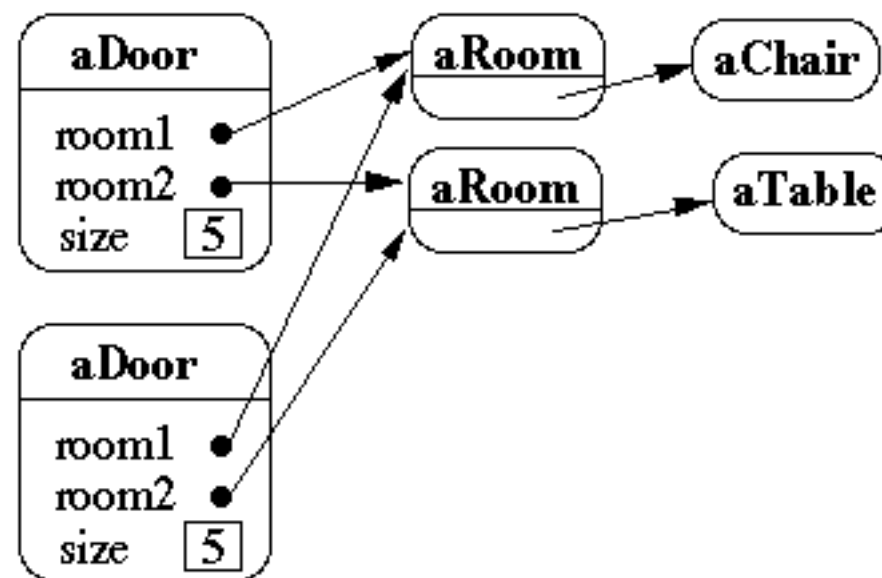
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

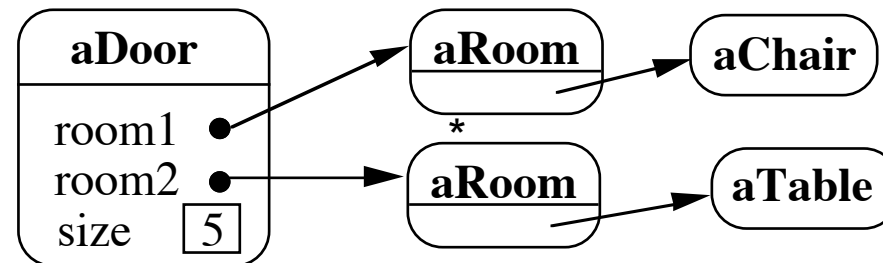


Shallow Copy

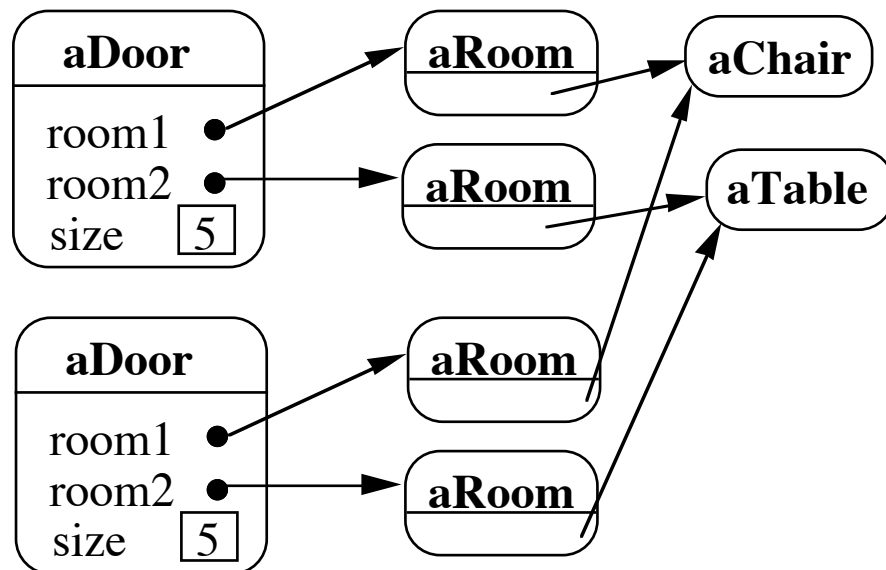


Shallow Copy Verse Deep Copy

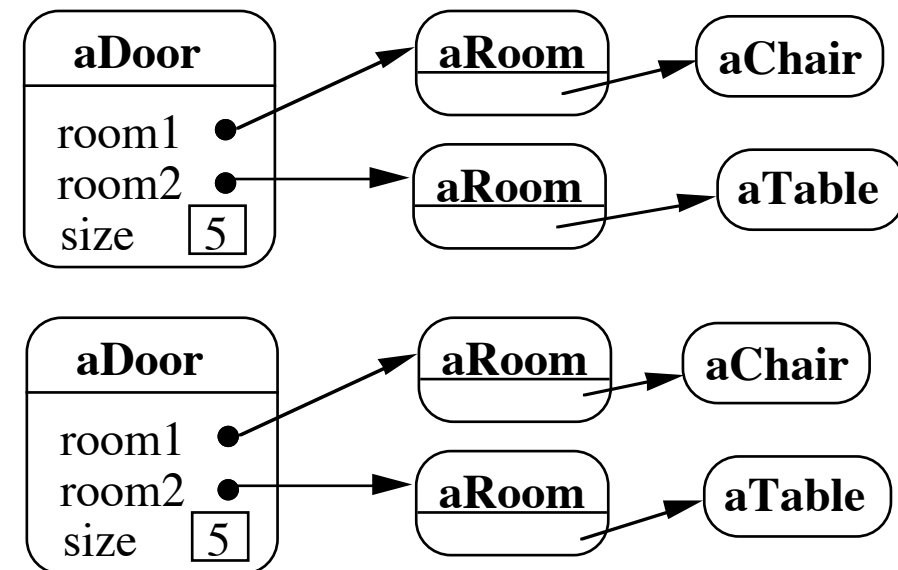
Original Objects



Deep Copy



Deeper Copy



Cloning Issues - C++ Copy Constructors

```
class Door {  
    public:  
        Door();  
        Door( const Door&);  
        virtual Door* clone() const;  
  
        virtual void Initialize( Room*, Room* );  
        // stuff not shown  
    private:  
        Room* room1;  
        Room* room2;  
}  
  
Door::Door ( const Door& other ) //Copy constructor {  
    room1 = other.room1;  
    room2 = other.room2;  
}  
  
Door* Door::clone() const {  
    return new Door( *this );  
}
```

Cloning Issues - Java Clone

Shallow Copy

```
class Door implements Cloneable {  
    private Room room1;  
    private Room room2;  
  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

Deep Copy

```
public class Door implements Cloneable {  
    private Room room1;  
    private Room room2;  
  
    public Object clone() throws CloneNotSupportedException {  
        Door thisCloned =(Door) super.clone();  
        thisCloned.room1 = (Room)room1.clone();  
        thisCloned.room2 = (Room)room2.clone();  
        return thisCloned;  
    }  
}
```

Prototype-based Languages

No classes

Behaviour reuse (inheritance)

Cloning existing objects which serve as prototypes

Some Prototype-based languages

Self

JavaScript

Squeak (eToys)

Perl with Class::Prototyped module