

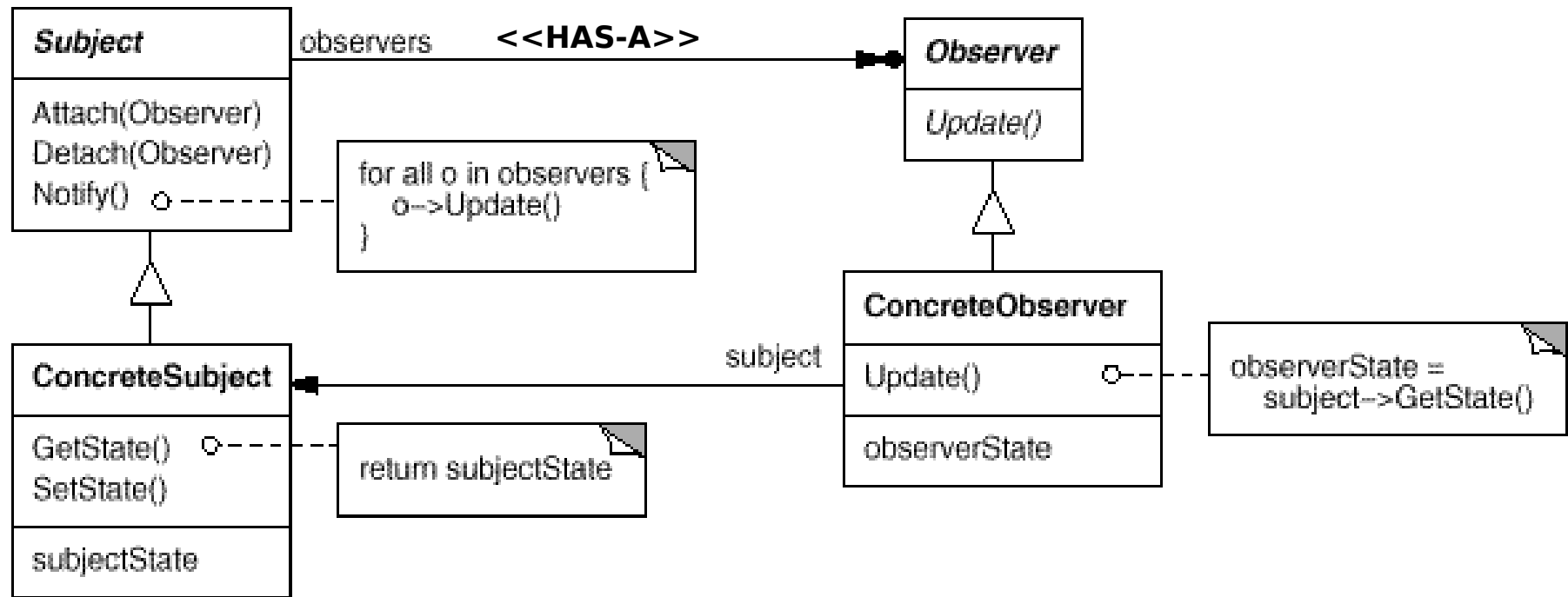
Observer Pattern

- Observer Pattern Overview
- Observer Pattern UML Diagram
- Observer Pattern Examples
- The observed list
- JList
- Servlet Session observer
- Advantages & Disadvantages

The Observer Pattern Overview

- **Allows an object to notify other objects when it changes.**
- Code pattern:
 - All observers implement a callback method for responding to change, e.g. *changeOccured()*.
 - Whenever the observed object changes, it notifies all observers by invoking that method.
- Observers ("**Listeners**") are common in JDK.
- In particular, note the swing MVC design.

Observer Pattern UML Diagram



Subject (the observed object) has methods for registering/deregistering observers, and a *notify()* method that goes through all observers and notifies them of a change, by calling their *update()* method. Subject is responsible to notify observers whenever its state changes.

The Observer Pattern Flow

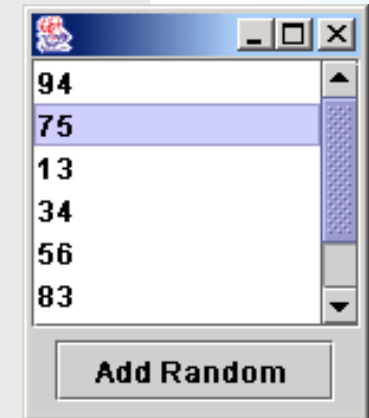
- The main framework instantiate the *ConcreteObservable* object.
- Then it instantiate and attaches the concrete observers to it using the methods defined in the *Observable* interface.
- Each time the state of the subject is changing it notifies all the attached Observers using the methods defined in the *Observer* interface.
- When a new *Observer* is added to the application, all we need to do is to instantiate it in the main framework and to add it to the *Observable* object.
- The classes already created will remain unchanged.

The Observer Pattern Use

- The observer pattern is used when:
 - The change of a state in one object must be reflected in another object without keeping the objects tight coupled.
 - Our framework needs to be enhanced in the future with new observers with minimal changes.
- Some Classical Examples:
 - MVC Pattern - The observer pattern is used in the model view controller architectural pattern. This pattern is used to decouple the model from the view. View represents the Observer and the model is the Observable object.
 - Event management - This is one of the domains where the Observer patterns is extensively used. Swing and .Net are extensively using the Observer pattern for implementing the events mechanism

Classic Swing Example:

```
// Whenever our list-model changes, it notifies the view,  
// which then automatically refreshes the display !  
public class MyListTest extends JFrame {  
    private DefaultListModel model = new DefaultListModel();  
    private JList view=new JList(model);  
    private JButton addBtn = new JButton("Add Random ");  
    public MyListTest(){  
        ...  
        addBtn.addActionListener(new ActionListener(){  
            public void actionPerformed(ActionEvent e){  
                int r= (int)(100 * Math.random());  
                model.addElement(new Integer(r));  
                // View magically updates !  
            }  
        });  
    }  
}
```



How is this done ?

// A slightly-simplified version of standard ListModel implementations:

```
class ListDataEvent {  
    private Object src;        // Object generating the event  
    private int code;          // Event code, e.g. ListDataEvent.INTERVAL_ADDED  
    private int index0;        // Changed occurred in interval [index0 ...index1]  
    private int index1;        //   or [index1 ...index0] ,   inclusive  
    ...  
}
```

// Callback (response) methods which every observer must implement:

```
interface ListDataListener {  
    void contentsChanged(ListDataEvent ev);  
    void intervalAdded(ListDataEvent ev);  
    void intervalRemoved(ListDataEvent ev);  
}
```


How is this done ? (cont.)

```
abstract class AbstractListModel{
    private ArrayList listeners = new ArrayList();    // todo: defer allocation
    public void addListener(ListDataListener listener){
        listeners.add(listener);
    }
    public void removeListener(ListDataListener listener){
        listeners.remove(listener);
    }
    protected void fireIntervalAdded(Object src, int index0, int index1){
        ListDataEvent ev=new ListDataEvent(src,
            ListDataEvent.INTERVAL_ADDED, index0, index1);
        for(Iterator it=listeners.iterator(); it.hasNext(); )
            ((ListDataListener)it.next()).intervalAdded(ev);
    }
    protected void fireIntervalRemoved(Object src, int index0, int index1) {... }
    protected void fireContentsChanged(Object src, int index0, int index1) {... }
    public abstract int getSize();
    public abstract Object getElementAt(int i);
}
```

The observed list:

```
class DefaultListModel extends AbstractListModel {
    private ArrayList impl=new ArrayList();
    public int getSize(){
        return impl.size();
    }
    public Object getElementAt(int i){
        return impl.get(i);
    }
    public void addElement(Object obj){
        int oldSize=getSize();
        impl.add(obj);
        fireIntervalAdded(this, oldSize-1, oldSize-1);
    }
    public void clear(){
        int oldSize=getSize();
        impl.clear();
        fireIntervalRemoved(this, 0, oldSize-1);
    }
}
```

JList (simplified):

```
class JList extends JComponent{
public JList(ListModel mod){
    mod.addListDataListener(new ListDataListener(){
        void contentsChanged(ListDataEvent ev){
            ...// refresh display
        }
        void intervalAdded(ListDataEvent ev){
            ...// refresh display
        }
        void intervalRemoved(ListDataEvent ev){
            ...// refresh display
        }
    });
}
}
```

Servlet Session observer:

```
// Servlet which dedicates one DB connection per session:
```

```
public class ShopServlet extends HttpServlet{  
    public void doGet(HttpServletRequest req, HttpServletResponse res) {  
        HttpSession session=req.getSession(true);  
        session.setMaxInactiveInterval(60*5); // 5 minutes  
        Connection dbConn =DriverManager.getConnection(url, user, pswd);  
        session.setAttribute("dbConnection", dbConn);  
        ...  
    }  
}
```

```
// Listener will close connection when session times-out:
```

```
public class MySessionListener implements HttpSessionListener {  
    public void sessionCreated(HttpSessionEvent ev){ }  
    public void sessionDestroyed(HttpSessionEvent ev){  
        Connection dbConn = (Connection)ev.getSession().getAttribute("dbConnection");  
        dbConn.close();  
        ...  
    }  
}
```

Many subjects to Many observers

- It's not a common situation but there are cases when a there are many observers that need to observe more than one subject.
- In this case the observer need to be notified not only about the change, but also which is the subject with the state changed.
- This can be realized very simple by adding to the subjects reference in the update notification method.
 - The subject will pass a reference to itself(this) to the when notify the observer.

Who triggers the update?

- The communication between the subject and its observers is done through the notify method declared in observer interface
- But it can be triggered from either subject or observer object. (Usually the notify method is triggered by the subject when it's state is changed)
- Sometimes when the updates are frequent the consecutive changes in the subject will determine many unnecessary refresh operations in the observer
 - To make this process more efficient the observer can be made responsible for starting the notify operation when it is considered necessary.

Specific Implementation Problems

Making sure Subject state is self-consistent before notification

- The subject state should be consistent when the notify operation is triggered
- If changes are made in the subject state after the observer is notified, it will be refreshed with an old state
- This seems hard to achieve but in practice this can be easily done when Subject subclass operations call inherited operations
- In the following example, the observer is notified when the subject is in an inconsistent state

Specific Implementation Problems

Making sure Subject state is self-consistent before notification (continued)

- In the following example, the observer is notified when the subject is in an inconsistent state

```
class Observable {
    int state = 0, additionalState = 0;
    public updateState(int increment) {
        state = state + increment;
        notifyObservers();
    }
}

class ConcreteObservable extends Observable{
    public updateState(int increment){
        super.updateState(increment); // the observers are notified
        // now the state changes (after notification)
        additionalState = additionalState + increment;
    }
}
```

Push and pull communication methods


- There are 2 methods of passing the data from the subject to the observer when the state is being changed in the subject side:
 - **Push model** - The subjects send detailed information about the change to the observer whether it uses it or not
 - Because the subject needs to send the detailed information to the observer this might be inefficient when a large amount of data needs to be sent and it is not used
 - Another approach would be to send only the information required by the observer
 - In this case the subject should be able to distinguish between different types of observers and to know the required data of each of them, meaning that the subject layer is more coupled to observer layer.

Push and pull communication methods (cont.)

➤ The second method is the pull model

- **Pull model** - The subject just notifies the observers when a change in it's state appears and it's the responsibility of each observer to pull the required data from the subject
- This can be inefficient because the communication is done in 2 steps and problems might appear in multithreading environments

Observer - advantages:

- Object notifies others when changes occur, without making much assumptions on who the observer is.
- Events are important in general-use tools
 - No other easy way to tell when a button is pressed, or when a servlet session expires.
- “Auto pilot”:
 - Observers will automatically update themselves to be consistent with changes (less details for programmer to remember).

Observer - disadvantages:

- Be careful not to end up with a **complex graph of object communication**, all listening to each other, hard to trace and maintain.
- Might be solved by combining the observation pattern with a Mediator
- Why is it not customary for Employee beans to fire events on salary change?

