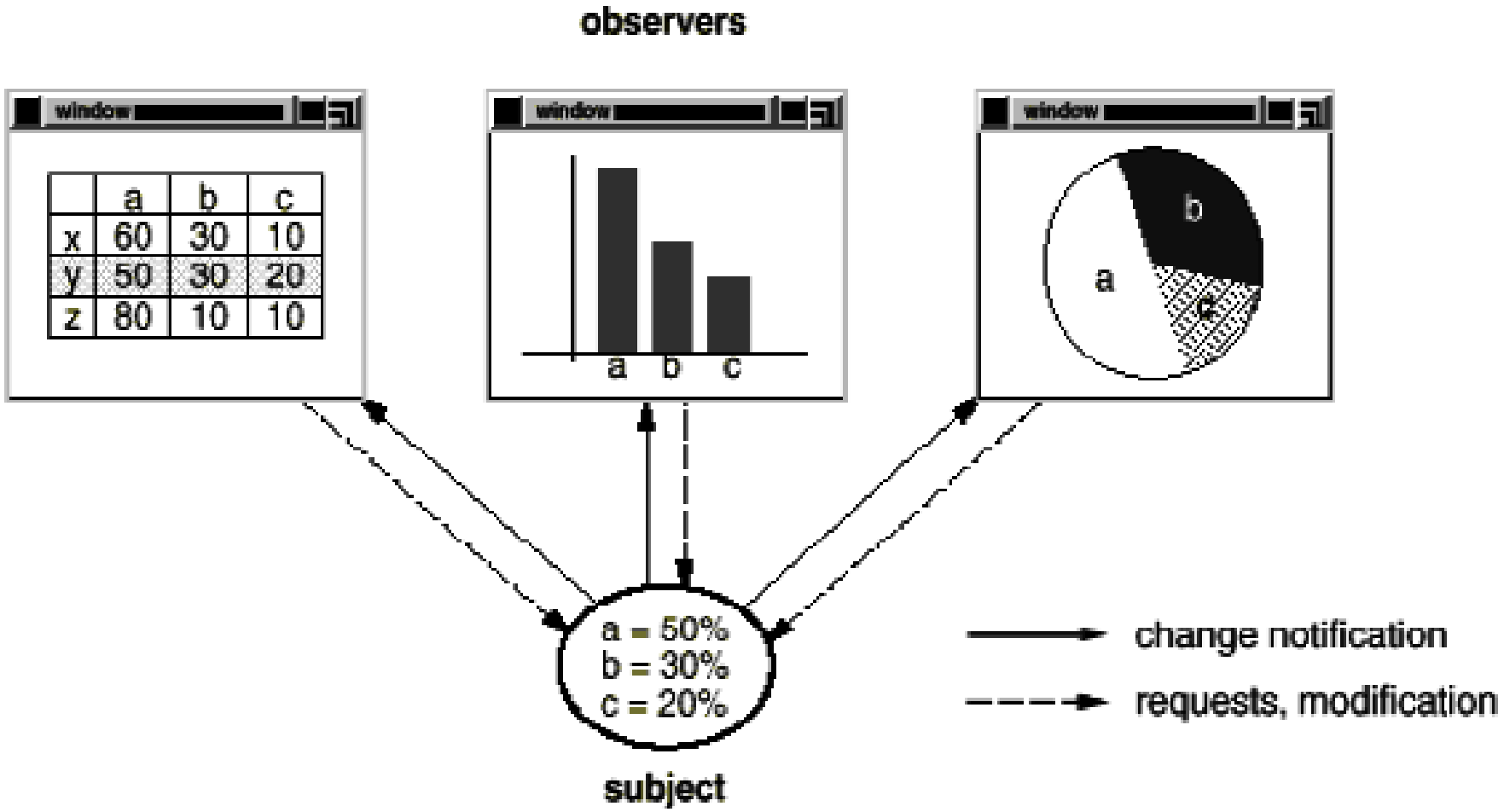
Tecniche di Progettazione: Design Patterns

GoF: Observer

The Observer Pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- AKA
  - Dependents, Publish-Subscribe, Model-View
- Motivation
  - The need to maintain consistency between related objects without making classes tightly coupled

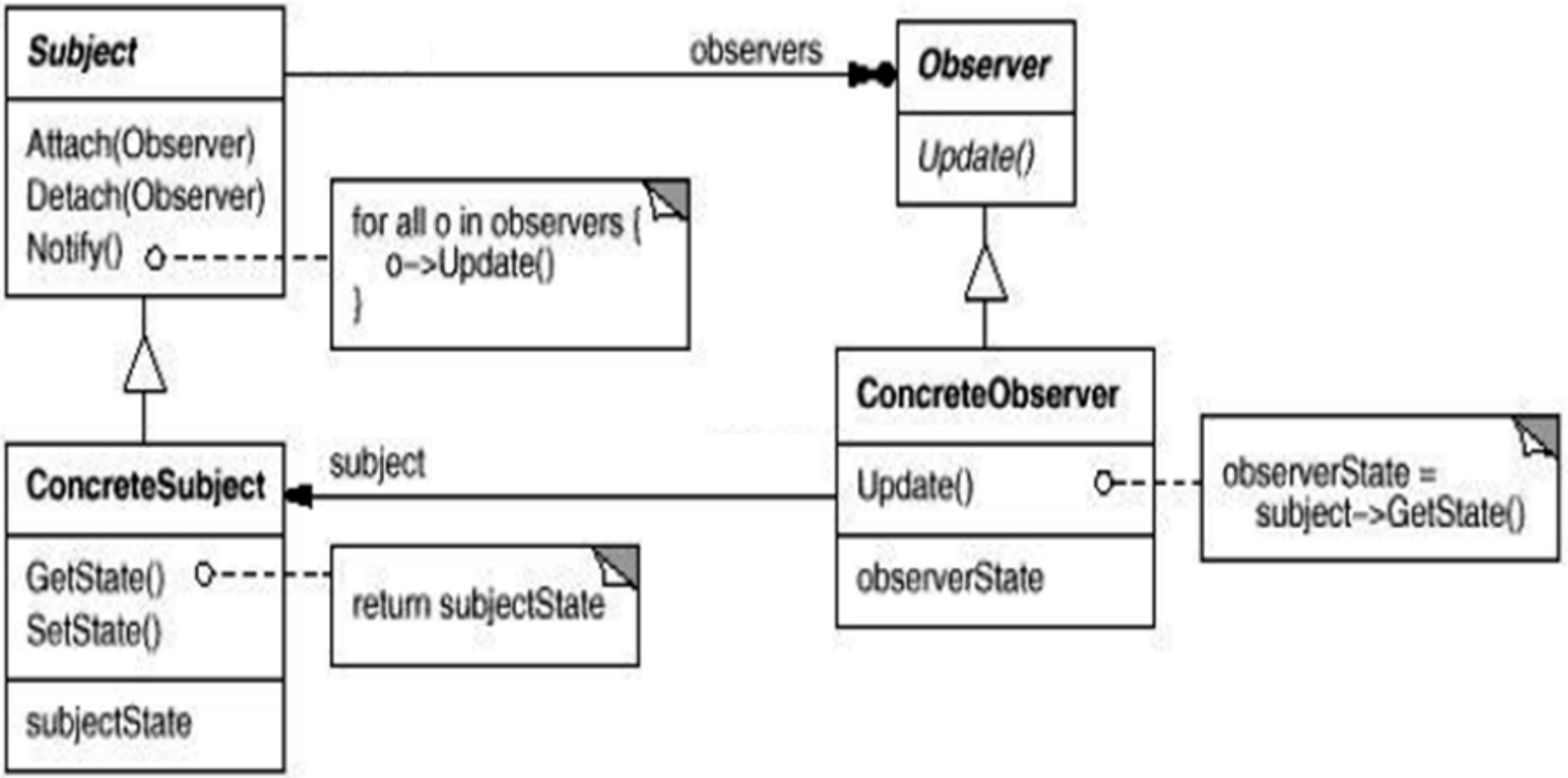
Example


observers
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a b c
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

Applicability

- Use the Observer pattern in any of the following situations:
  - When a change to one object requires changing others
  - When an object should be able to notify other objects without making assumptions about those objects

Structure


Subject
Attach(Observer)
Detach(Observer)
Notify()
observers
Observer
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
ConcreteSubject
GetState()
SetState()
subjectState
subject
return subjectState

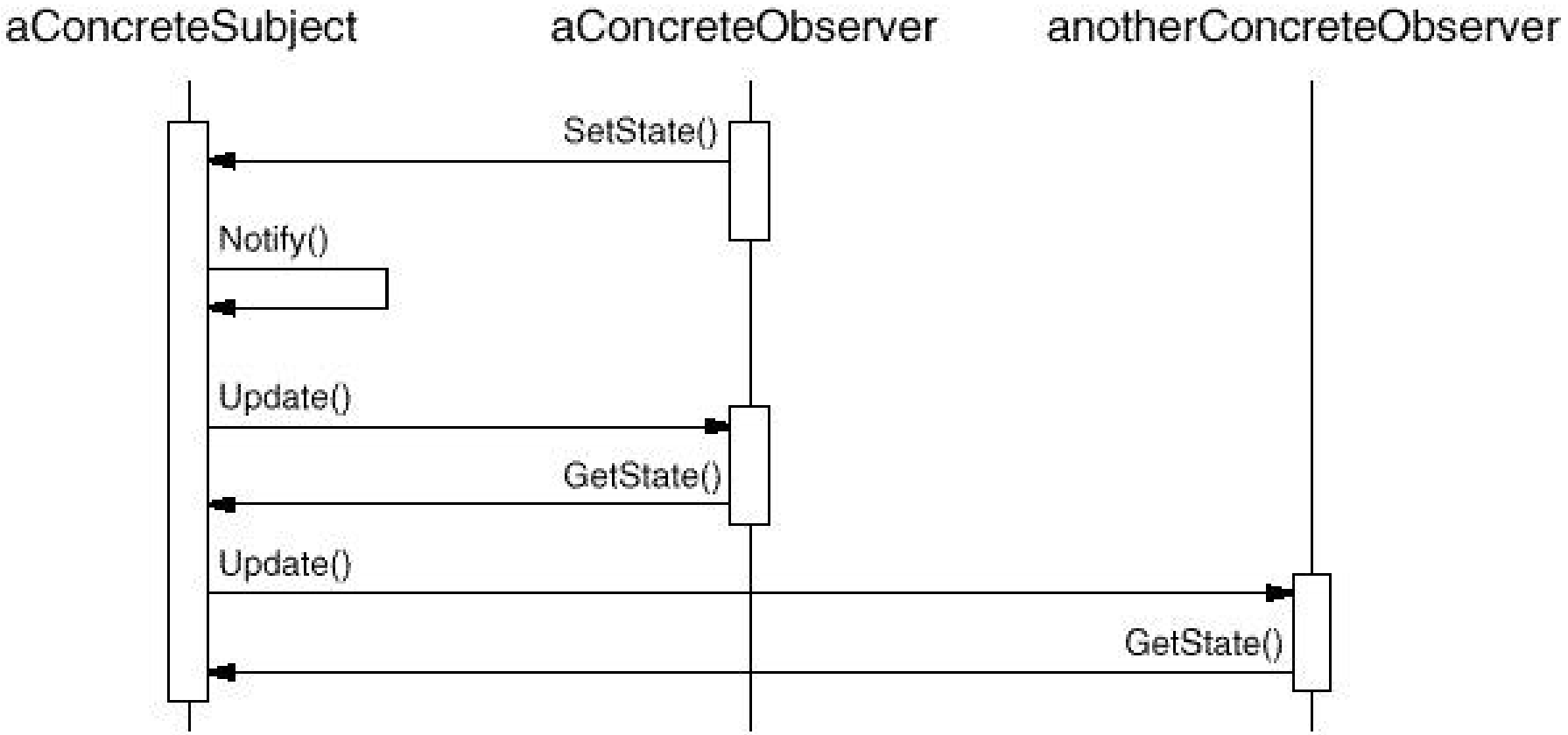
Participants

- Subject
  - Keeps track of its observers
  - Provides an interface for attaching and detaching Observer objects
- Observer
  - Defines an interface for update notification
- ConcreteSubject
  - The object being observed
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to its observers when its state changes

Participants (cont'd)

- ConcreteObserver
  - The observing object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer update interface to keep its state consistent with the subject's

A scenario


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()

Benefits

- Minimal coupling between the Subject and the Observer
- Can reuse subjects without reusing their observers and vice versa
- Observers can be added without modifying the subject
- Each subject knows is its list of observers
- Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
- Subject and observer can belong to different abstraction layers

Support for event multicasting

- Subject sends notification to all subscribed observers
- Observers can be added/removed at any time

Liabilities

- Possible cascading of notifications
- Observers are not necessarily aware of each other and must be careful about triggering updates
- Simple update interface requires observers to deduce changed item

Known Uses

- Smalltalk Model/View/Controller user interface framework
  - Model = Subject
  - View = Observer
  - Controller is whatever object changes the state of the subject
- Java 1.1 AWT/Swing Event Model

Implementation Issues

- How does the subject keep track of its observers?
  - Array, linked list
- What if an observer wants to observe more than one subject?
  - Have the subject tell the observer who it is via the update interface
- Who triggers the update?
  - The subject whenever its state changes
  - The observers after they cause one or more state changes
  - Some third party object(s)
- Make sure the subject updates its state before sending out notifications

Implementation Issues(cont'd)

- How much info about the change should the subject send to the observers?
  - Push Model – Lots (the subject push all data to the observers)
  - Pull Model - Very Little (obsetvers extract the data they need)
- Can the observers subscribe to specific events of interest?
  - If so, it's publish-subscribe
- Can an observer also be a subject?
  - Yes!

Implementation Issues(cont'd)

- What if an observer wants to be notified only after several subjects have changed state?
  - Use an intermediary object which acts as a mediator
  - Subjects send notifications to the mediator object which performs any necessary processing before notifying the observers

Java Implementation Of Observer

- Java provides the Observable/Observer classes as built-in support for the Observer pattern
- The java.util.Observable class is the base Subject class. Any class that wants to be observed extends this class.
  - Provides methods to add/delete observers
  - Provides methods to notify all observers
  - A subclass only needs to ensure that its observers are notified in the appropriate mutator methods
  - Uses a Vector for storing the observer references
- The java.util.Observer interface is the Observer interface. It must be implemented by any observer class.

java.util

Interface Observer

- public interface Observer
  - A class can implement the Observer interface when it wants to be informed of changes in observable objects.
  - Since: JDK1.0
- void update(Observable o, Object arg)
  - This method is called whenever the observed object is changed. An application calls an Observable object's notifyObservers method to have all the object's observers notified of the change.
  - Parameters:
    - o - the observable object.
    - arg - an argument passed to the notifyObservers method.

java.util

Class Observable

- public class **Observable**
  - This class represents an observable object, or "data" in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed.
  - An observable object can have one or more observers. An observer may be any object that implements interface Observer. After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by a call to their update method.

java.util

Class Observable

- The order in which notifications will be delivered is unspecified. The default implementation provided in the Observable class will notify Observers in the order in which they registered interest, but subclasses may change this order, use no guaranteed order, deliver notifications on separate threads, or may guarantee that their subclass follows this order, as they choose.
- Note that this notification mechanism has nothing to do with threads and is completely separate from the wait and notify mechanism of class Object.
- When an observable object is newly created, its set of observers is empty. Two observers are considered the same if and only if the equals method returns true for them.

- **Since:** JDK1.0

java.util

Class Observable: Observers list

- public void **addObserver**(Observer o)
  - Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set. The order in which notifications will be delivered to multiple observers is not specified.
- public void **deleteObserver**(Observer o)
  - Deletes an observer from the set of observers of this object. Passing null to this method will have no effect.
- public void **deleteObservers**()
  - Clears the observer list: this object no longer has any observers
- public int **countObservers**()
  - Returns the number of observers of this Observable object

java.util

Class Observable: notifyObservers

- public void **notifyObservers**(Object arg)
  - If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed. Each observer has its update method called with two arguments: this observable object and the arg argument.
- public void **notifyObservers**()
  - equivalent to: notifyObservers(null)

java.util

Class Observable: changes

- protected void **setChanged**()
  - Marks this Observable object as having been changed; the hasChanged method will now return true.
- protected void **clearChanged**()
  - Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the hasChanged method will now return false. This method is called automatically by the notifyObservers methods.
- public boolean **hasChanged**()
  - Returns true if and only if the setChanged method has been called more recently than the clearChanged method on this object; false otherwise.

Observable/Observer Example

```
/**
* A subject to observe!
*/
public class ConcreteSubject extends Observable {
  private String name;
  private float price;
  public ConcreteSubject(String name, float price) {
       this.name = name;
       this.price = price;
       System.out.println("ConcreteSubject created: " +
              name + " at " + price);
}
```

Observable/Observer Example (Cont'd)

```
  public String getName() {return name;}
  public float getPrice() {return price;}
  public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }
  public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
  }
}
```

Observable/Observer Example (Cont'd)

```
// An observer (of name changes).
public class NameObserver implements Observer {
  private String name;
  public NameObserver() {
    name = null;
    System.out.println("NameObserver created");
}
public void update(Observable obj, Object arg) {
  if (arg instanceof String) {
       name = (String)arg;
       System.out.println("NameObserver: Name changed to " +
       name);
  } else {
  System.out.println("NameObserver: Some other change to
       subject!");
}}}
```

Observable/Observer Example (Cont'd)

```
// An observer of price changes.
public class PriceObserver implements Observer {
  private float price;
  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }
  public void update(Observable obj, Object arg) {
       if (arg instanceof Float) {
              price = ((Float)arg).floatValue();
              System.out.println("PriceObserver: Price changed
              to " + price);
       } else {
              System.out.println(”PriceObserver: Some other
              change tosubject!");
}}}
```

Observable/Observer Example (Cont'd)

```
// Test program for ConcreteSubject, NameObserver and
   PriceObserver
public class TestObservers {
   public static void main(String args[]) {
   // Create the Subject and Observers.
   ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
   NameObserver nameObs = new NameObserver();
   PriceObserver priceObs = new PriceObserver();
   // Add those Observers!
   s.addObserver(nameObs);
   s.addObserver(priceObs);
   // Make changes to the Subject.
   s.setName("Frosted Flakes");
   s.setPrice(4.57f);
   s.setPrice(9.22f);
   s.setName("Sugar Crispies");
}}
```

Observable/Observer Example (Cont'd)

Test program output

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22
NameObserver: Some other change to subject!
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```

A Problem With Observable/Observer

- Problem:
  - Suppose the class which we want to be an observable is already part of an inheritance hierarchy:
  - **class SpecialSubject extends ParentClass**
  - Since Java does not support multiple inheritance, how can we have ConcreteSubject extend both Observable and ParentClass?

A Problem With Observable/Observer

- Solution:
  - Use Delegation
  - We will have SpecialSubject contain an Observable object
  - We will delegate the observable behavior that SpecialSubject needs to this contained Observable object

Delegated Observable

```
/**
* A subject to observe!
* Use a contained DelegatedObservable object.
* Note that in this version of SpecialSubject we do
* not duplicate any of the interface of Observable.
* Clients must get a reference to our contained
* Observable object using the getObservable()
  method.
*/
public class SpecialSubject extends ParentClass {
  private String name;
  private float price;
  private DelegatedObservable obs;
```

Delegated Observable (cont'd)

```
public SpecialSubject(String name, float price) {
  this.name = name;
  this.price = price;
  obs = new DelegatedObservable();
}
public String getName() {return name;}
public float getPrice() {return price;}
public Observable getObservable() {return obs;}
```

Delegated Observable (cont'd)

```
  public void setName(String name) {
      this.name = name;
      obs.setChanged();
      obs.notifyObservers(name);
  }
  public void setPrice(float price) {
      this.price = price;
      obs.setChanged();
      obs.notifyObservers(new Float(price));
  }
}
```

Delegated Observable (cont'd)

- What's this DelegatedObservable class?
- Two methods of java.util.Observable are protected methods: setChanged() and clearChanged()
- Apparently, the designers of Observable felt that only subclasses of Observable (that is, "true" observable subjects) should be able to modify the state-changed flag
- If SpecialSubject contains an Observable object, it could not invoke the setChanged() and clearChanged() methods on it
- So we have DelegatedObservable extend Observable and override these two methods making them have public visibility
- Java rule: A subclass can change the visibility of an overridden method of its superclass, but only if it provides more access

Delegated Observable (cont'd)

```
// A subclass of Observable that allows delegation.
public class DelegatedObservable extends Observable
  {
      public void clearChanged() {
            super.clearChanged();
      }
      public void setChanged() {
            super.setChanged();
      }
  }
```

Delegated Observable (cont'd)

```
// Test program for SpecialSubject with a Delegated Observable.
public class TestSpecial {
  public static void main(String args[]) {
        // Create the Subject and Observers.
        SpecialSubject s = new SpecialSubject("Corn Pops", 1.29f);
        NameObserver nameObs = new NameObserver();
        PriceObserver priceObs = new PriceObserver();
        // Add those Observers!
        s.getObservable().addObserver(nameObs);
        s.getObservable().addObserver(priceObs);
        // Make changes to the Subject.
        s.setName("Frosted Flakes");
        s.setPrice(4.57f);
        s.setPrice(9.22f);
        s.setName("Sugar Crispies");
}}
```

Delegated Observable (cont'd)

- This version of SpecialSubject did not provide implementations of any of the methods of Observable. As a result, it had to allow its clients to get a reference to its contained Observable object using the getObservable() method. This may have dangerous consequences. A rogue client could, for example, call the deleteObservers() method on the Observable object, and delete all the observers!
- Let's have SpecialSubject not expose its contained Observable object, but instead provide "wrapper" implementations of the addObserver() and deleteObserver() methods which simply pass on the request to the contained Observable object.

Delegated Observable 2

```
/**
* A subject to observe!  But this subject already extends another class.
* So use a contained DelegatedObservable object.
* Note that in this version of SpecialSubject we provide implementations of two of the methods
* of Observable: addObserver() and deleteObserver().
* These implementations simply pass the request on to our contained DelegatedObservable reference.
* Now clients can use the normal Observable semantics to add themselves as observers of this object.
*/
public class SpecialSubject2 extends ParentClass {
    private String name;
    private float price;
    private DelegatedObservable obs;
```

Delegated Observable 2 (cont'd)

```
public SpecialSubject2(String name, float price) {
    this.name = name;
    this.price = price;
    obs = new DelegatedObservable();
}
public String getName() {return name;}
public float getPrice() {return price;}
public void addObserver(Observer o) {
    obs.addObserver(o);
}
public void deleteObserver(Observer o) {
    obs.deleteObserver(o);
```

Delegated Observable 2 (cont'd)

```
public void setName(String name) {
    this.name = name;
    obs.setChanged();
    obs.notifyObservers(name);
}
public void setPrice(float price) {
    this.price = price;
    obs.setChanged();
    obs.notifyObservers(new Float(price));
}
}
```

Delegated Observable 2 (cont'd)

```
// Test program for SpecialSubject2 with a Delegated Observable.
public class TestSpecial2 {
  public static void main(String args[]) {
  // Create the Subject and Observers.
  SpecialSubject2 s = new SpecialSubject2("Corn Pops", 1.29f);
  NameObserver nameObs = new NameObserver();
  PriceObserver priceObs = new PriceObserver();
  // Add those Observers!
  s.addObserver(nameObs);
  s.addObserver(priceObs);
  // Make changes to the Subject.
  s.setName("Frosted Flakes");
  s.setPrice(4.57f);
  s.setPrice(9.22f);
  s.setName("Sugar Crispies");
}}
```

Java 1.1 Event Model

- Java 1.1 introduced a new GUI event model based on the Observer Pattern
- GUI components which can generate GUI events are called *event sources*
- Objects that want to be notified of GUI events are called *event listeners*
- Event generation is also called *firing the event*
- Comparison to the Observer Pattern:
  - ConcreteSubject => event source
  - ConcreteObserver => event listener
- For an event listener to be notified of an event, it must first register with the event source

Java 1.1 Event Model (cont'd)

- An event listener must implement an interface which provides the method to be called by the event source when the event occurs
- Unlike the Observer Pattern which defines just the one simple Observer interface, the Java 1.1 AWT event model has 11 different listener interfaces, each tailored to a different type of GUI event:
  - Listeners For Semantic Events
    - ActionListener
    - AdjustmentListener
    - ItemListener
    - TextListener

Java 1.1 Event Model (cont'd)

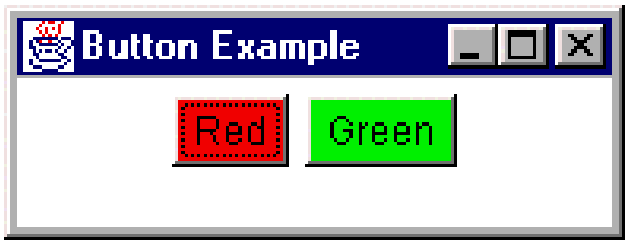
- Listeners For Low-Level Events
  - ComponentListener
  - ContainerListener
  - FocusListener
  - KeyListener
  - MouseListener
  - MouseMotionListener
  - WindowListener
- Some of these listener interfaces have several methods which must be implemented by an event listener. For example, the WindowListener interface has seven such methods. In many cases, an event listener is really only interested in one specific event, such as the Window Closing event

Java 1.1 Event Model (cont'd)

- Java provides "adapter" classes as a convenience in this situation.
- For example, the WindowAdapter class implements the WindowListener interface, providing "do nothing" implementation of all seven required methods. An event listener class can extend WindowAdapter and override only those methods of interest.

Java AWT

```
import java.awt.*;
import java.awt.event.*;
/**
* An example of the Java 1.1 AWT event model.
* This class not only builds the GUI, but it is the
* listener for button events.
*/
public class ButtonExample1
extends WindowAdapter
implements ActionListener {
Frame frame;
Panel buttonPanel;
Button redButton, greenButton;
```



Java AWT (cont'd)

```
// Build the GUI and display it.
public ButtonExample1(String title) {
  frame = new Frame(title);
  buttonPanel = new Panel(new FlowLayout());
  redButton = new Button("Red");
  redButton.setBackground(Color.red);
  redButton.setActionCommand("Change To Red");
  redButton.addActionListener(this);
  buttonPanel.add(redButton);
  greenButton = new Button("Green");
  greenButton.setBackground(Color.green);
  greenButton.setActionCommand("Change To Green");
  greenButton.addActionListener(this);
  buttonPanel.add(greenButton);
```

Java AWT (cont'd)

```
frame.add("Center", buttonPanel);
frame.addWindowListener(this);
frame.pack();
frame.setVisible(true);
}
// Since we are a WindowAdapter, we already implement the
// WindowListener interface.
// So only override those methods
// we are interested in.
public void windowClosing(WindowEvent e) {
  System.exit(0);
}
```

Java AWT (cont'd)

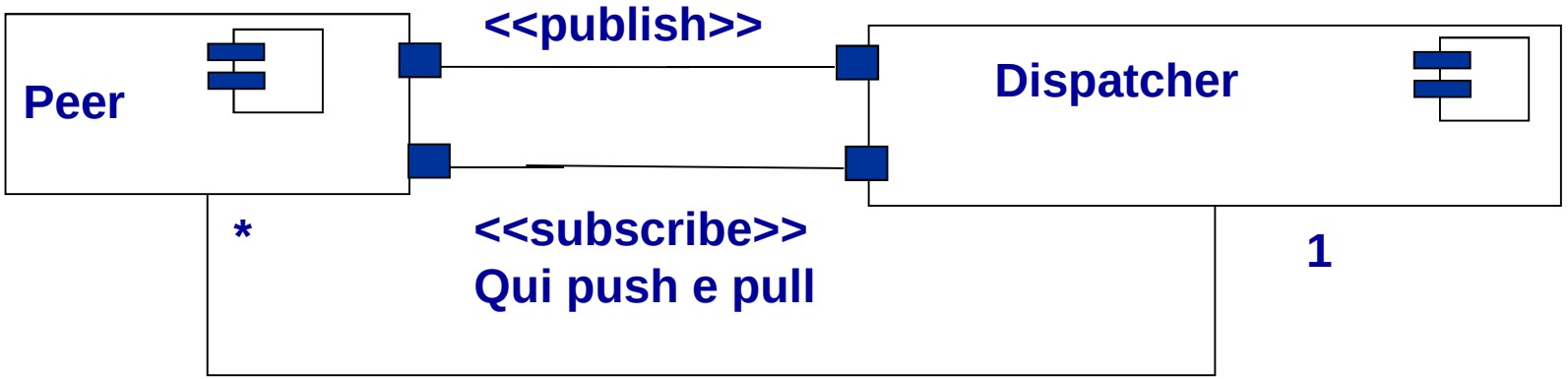
```
// Since we handle the button events, we must implement
// the ActionListener interface.
public void actionPerformed(ActionEvent e) {
  String cmd = e.getActionCommand();
  if (cmd.equals("Change To Red")) {
       System.out.println("Red pressed");
       buttonPanel.setBackground(Color.red);
  }
  else if (cmd.equals("Change To Green")) {
       System.out.println("Green pressed");
       buttonPanel.setBackground(Color.green);
  }}
public static void main(String args[]) {
  new ButtonExample1("Button Example");
}}
```

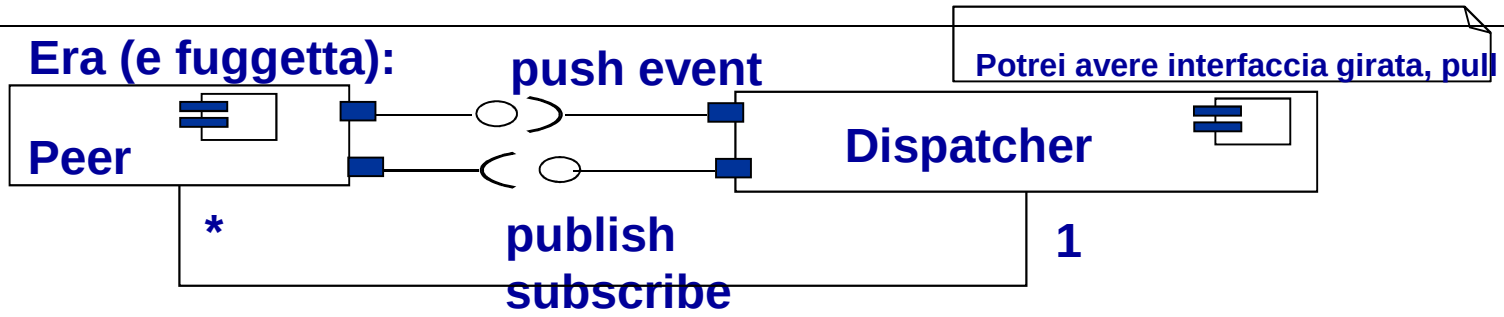
Publish-subscribe

- Le componenti interagiscono annunciando eventi: ciascuna componente si "abbona" a classi di eventi rilevanti per il suo scopo
- Ciascuna componente, volendo, può essere sia produttore che consumatore di eventi
- Disaccoppia produttori e consumatori di eventi e favorisce le modifiche dinamiche del sistema

Publish-subscribe

Operazioni: subscribe, unsubscribe, publish, notify (push), letMeKnow (pull)





Publish-subscribe

- In questo stile, mittenti e destinatari di messaggi dialogano attraverso un tramite, detto dispatcher o broker.
- Il mittente di un messaggio (detto publisher) non deve essere consapevole dell'identità dei destinatari (detti subscriber); esso si limita a "pubblicare" (to publish) il proprio messaggio al dispatcher.
- I destinatari si rivolgono a loro volta al dispatcher "abbonandosi" (to subscribe) alla ricezione di messaggi.
- Il dispatcher quindi inoltra ogni messaggio inviato da un publisher a tutti i subscriber interessati a quel messaggio.

Publish-subscribe

- In genere, il meccanismo di sottoscrizione consente ai subscriber di precisare nel modo più specifico possibile a quali messaggi sono interessati. Per esempio, un subscriber potrebbe "abbonarsi" solo alla ricezione di messaggi da determinati publisher, oppure aventi certe caratteristiche.
- Questo schema implica che ai publisher non sia noto quanti e quali sono i subscriber e viceversa. Questo può contribuire alla scalabilità del sistema.

Publish-subscribe: application ex

- Instant messaging
  - Implementazione peer-to-peer (WASTE)
  - Implementazione centralizzata (IRC, Jabber)
- HTTP streaming

Publish-subscribe: middleware e reti

- Reti: multicast con algoritmi di flooding

- Il Data Distribution Service for Real Time Systems (DDS) è uno standard emanato dall'Object Management Group (OMG) che definisce un middleware per la distribuzione di dati in tempo reale secondo il paradigma publish/subscribe.

Publish-subscribe e web services

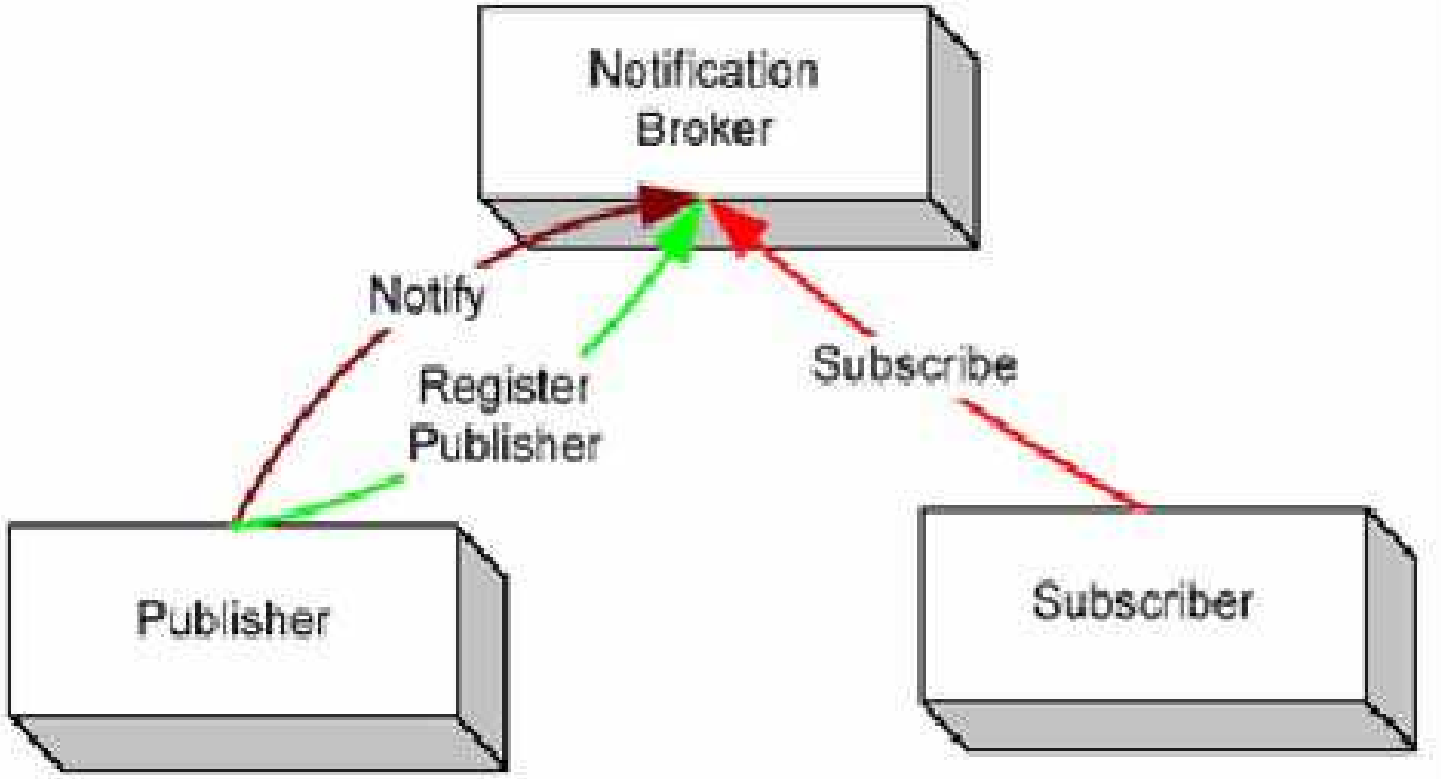


Figura 6. interfaccia del broker

Publish-subscribe e web services

- il publisher preliminarmente deve aver contattato il broker mediante il metodo RegisterPublisher dell'interfaccia del broker dichiarando di essere disposto a pubblicare notifiche di eventi relativi a specifici topics.
- Una volta registratosi presso il broker, il publisher invierà le notifiche degli eventi che verranno prodotti soltanto al broker presso cui si è registrato. Il broker sarà in grado di capire verso quali consumer dirigere le notifiche di evento poiché manterrà la lista di tutte le sottoscrizioni.
- Spetterà poi ai brokers contattare i consumers sottoscritti inviando copie del messaggio di notifica ricevuto dal publisher. La presenza di un broker non modifica la semantica comportamentale del subscription manager.