**Observer Design Pattern**

**1, Name and Classification:**

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

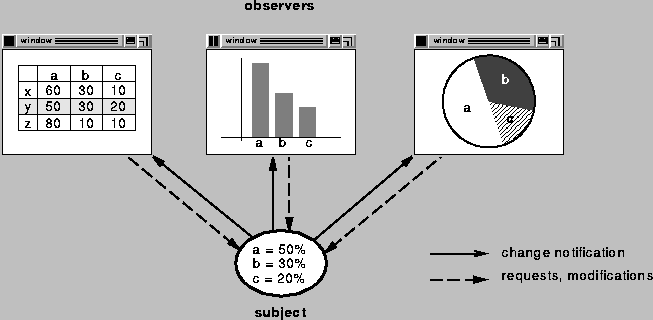
**2, Also Known as:**

Dependents, Publish-Subscribe

**3, Motivation:**

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.

This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there’s no reason to limit the number of different user interfaces to the same data. The Observer pattern describes how to establish these relationships.

**4, Application:**

Use the Observer pattern in any of the following situations:

* When an abstraction has two aspects, one dependent on the

other.Encapsulating these aspects in separate objects lets you vary

andreuse them independently.

* When a change to one object requires changing others, and youdon't know

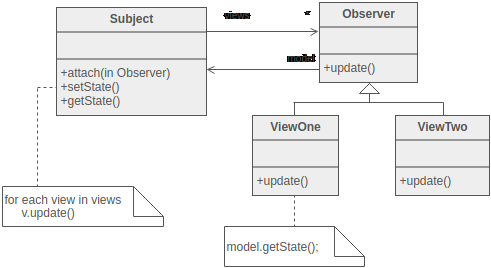
how many objects need to be changed.

* When an object should be able to notify other objects without

makingassumptions about who these objects are. In other words, you don'twant

these objects tightly coupled.

**5, Structure:**



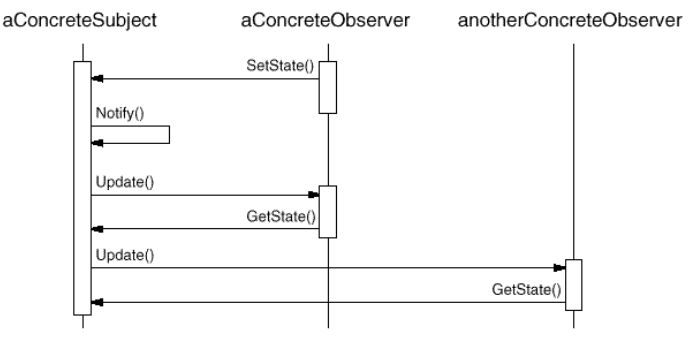
Subject represents the core (or independent or common or engine) abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction. The Subject prompts the Observer objects to do their thing. Each Observer can call back to the Subject as needed.

**6, Participants:**

* Subject: knows its Observer(s), provides operations for attaching and detacing Observer objects
* Observer: defines an updating interface for supporting notification about changes in Subject.
* ConcreteSubject: stores state of interest to ConcreteObserver objects, sends a notification to its observers upon state change.
* ConcreteObserver: maintains a reference to a ConcreteSubject object, stores state that should stat consistent with the subject, implements the Observer updating interface.

**7, Collaboration:**

* ConcreteSubject notifies its observers whenever a changeoccurs that couldmake its observers' state inconsistent with its own.
* After being informed of a change in the concrete subject, aConcreteObserver object may query the subject for information.ConcreteObserver uses this information to reconcile its state with thatof the subject.
* The following interaction diagram illustrates the collaborationsbetween a subject and two observers:

****

Note how the Observer object that initiates the change requestpostpones its update until it gets a notification from the subject.Notify is not always called by the subject. It can be called by anobserver or by another kind of object entirely. The Implementationsection discusses some common variations.

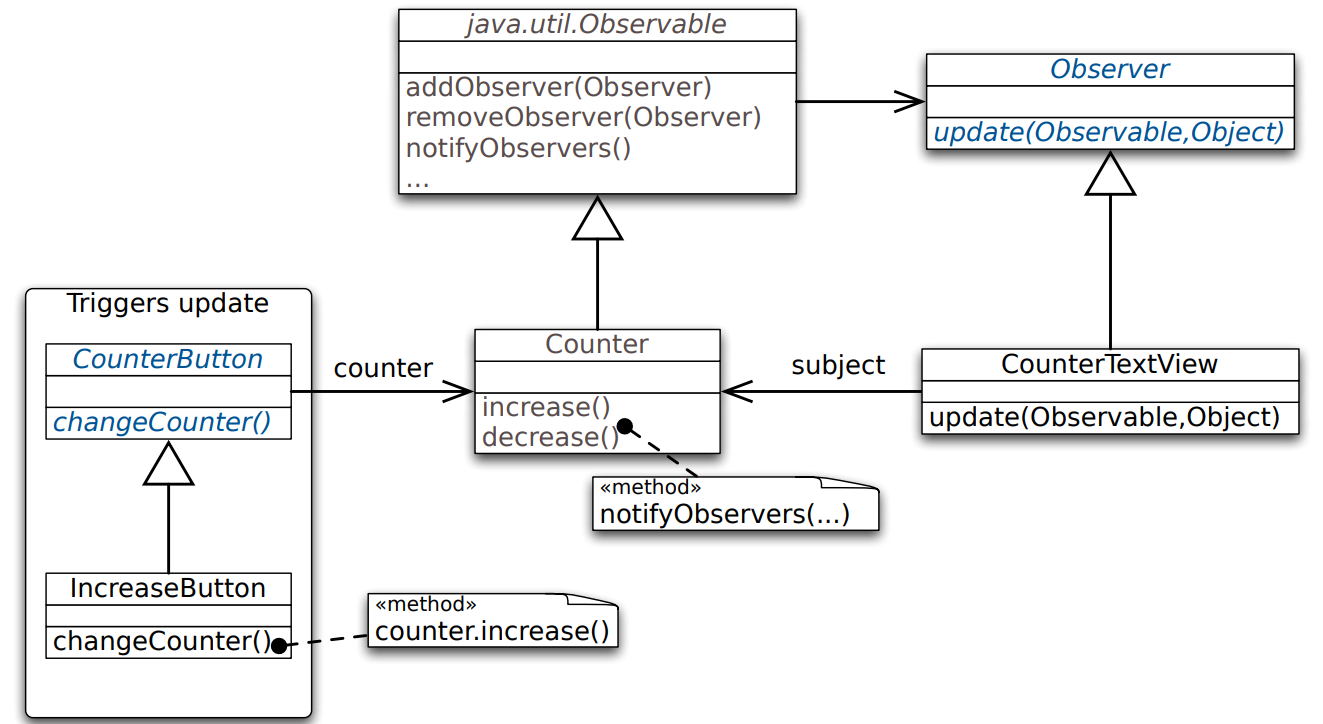
**8, Consequences:**

* Abstract coupling between Subject and Observer
* Support for broadcast communication
* notify doesn’t specify its receiver
* the sender doesn’t know the type of the receiver
* Unexpected/Uncontrolled updates:
* Danger of update cascades to observers and their dependent objects.
* Update sent to all observers, even though some of them may not be interested in the particular change
* No detail of what changed in the subjectl; observers may need to work hard to figure out what changed
* A common update interface for all observes limits the communication interface: Subject cannot send optional parameters to Observers

**9, Implementation:**

* addObserver(Observer): adds an observer to the observer list
* clearChanged(): clears an observable change
* countObservers(): counts the number of observer
* deleteObserver(Observer): deletes an observer form the observer list
* hasChanged(): returns a true Boolean if an observable change has occurred
* notifyObservers(): notifies all observers about an observable change
* notiffyObservers(Object) : notifies all observers of the specified obserable change occured
* setChanged(): sets a flag to note an observable change

**10, Sample Code:**



class Counter extends java.util.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);

} }

abstract class CounterButton extends Button {

protected Counter counter;

public CounterButton(String buttonName, Counter counter) {

super(buttonName);

this.counter = counter;

}

public boolean action(Event processNow, Object argument) {

changeCounter();

return true;

}

abstract protected void changeCounter();

}

abstract class CounterButton extends Button {

protected Counter counter;

public CounterButton(String buttonName, Counter counter) {

super(buttonName);

this.counter = counter;

}

public boolean action(Event processNow, Object argument) {

changeCounter();

return true;

}

abstract protected void changeCounter();

}

class IncreaseButton extends CounterButton{

public IncreaseButton(Counter counter) {

super("Increase", counter);

}

protected void changeCounter() { counter.increase(); }

}

class DecreaseButton extends CounterButton{/\* correspondingly… \*/}

class CounterTextView implements Observer{

Counter model;

public CounterTextView(Counter model) {

this.model= model;

model.addObserver(this);

}

public void paint(Graphics display) {

display.drawString(

"The value of "+model.label()+" is"+model,1,1

);

}

public void update(Observable counter, Object argument) {

repaint();

}

}

class Counter extends Observable {

private boolean increased = false;

boolean isIncreased() { return increased; }

void increase() {

count++;

increased=true;

setChanged();

notifyObservers();

}

}

class IncreaseDetector extends Counter implements Observer {

void update(Observable subject) {

if(((Counter)subject).isIncreased()) increase();

}

}

class Counter extends Observable {

void increase() {

count++;

setChanged();

notifyObservers(INCREASE);

}

}

class IncreaseDetector extends Counter implements Observer {

void update(Observable whatChanged, Object message) {

if(message.equals(INCREASE)) increase();

}

}

**11, Known Uses:**

The first and perhaps best-known example of the Observer pattern appearsin

Smalltalk Model/View/Controller (MVC), the user interface framework in the

Smalltalkenvironment [KP88]. MVC's Model class plays the role ofSubject, while

View is the base class for observers. Smalltalk,ET++ [WGM88], and the THINK class

library [Sym93b] provide ageneral dependency mechanism by putting Subject and

Observer interfacesin the parent class for all other classes in the system.

Other user interface toolkits that employ this pattern areInterViews [LVC89],

the AndrewToolkit [P+88], and Unidraw [VL90]. InterViewsdefines Observer and

Observable (for subjects) classes explicitly.Andrew calls them "view" and "data

object," respectively. Unidrawsplits graphical editor objects into View (for

observers) and Subjectparts.

**12, Related Patterns:**

* Mediator (305): Byencapsulating complex update semantics, the ChangeManager acts asmediator between subjects and observers.
* Singleton (144):The ChangeManager may use the Singleton pattern to make it uniqueand globally accessible.