Observer Design Pattern

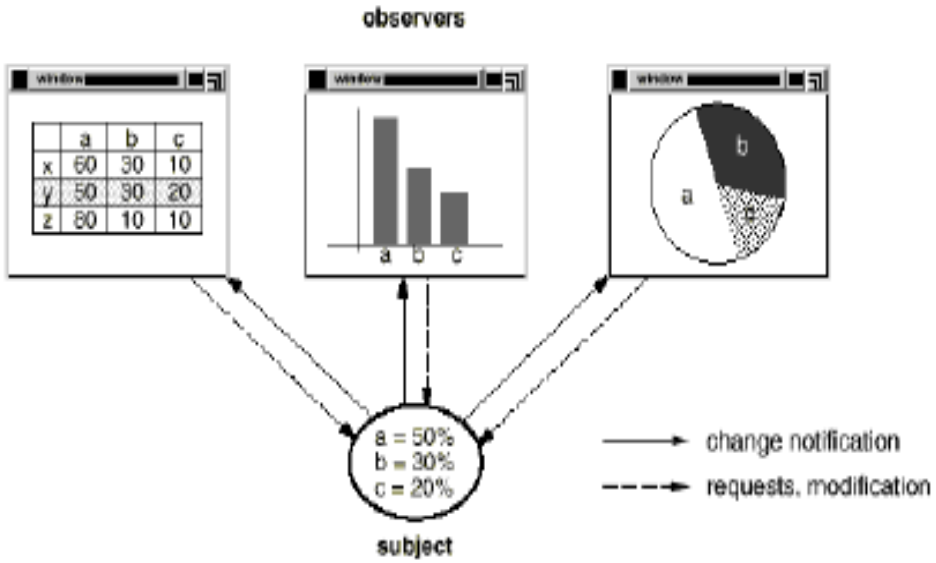
Also Known As:
Dependents,
Publish-Subscribe,
and confusingly, Model-View

Introduction to Observer

- A *Gang of Four* design pattern, one of the patterns discussed in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
- Observer defines a one-to-many or many-to-many dependency between objects.
- When the state of one object changes, then all the other objects that are dependent on that object are updated automatically.
- Used for event handling where consistency between objects is necessary, e.g. Swing Framework for GUI development.

General Example

- Suppose you have some data that can be displayed by a table, a bar graph or a pie chart.
- Changes to the underlying data should be reflected in al three of the displays
- This is where the Observer Design Pattern comes in handy.



Motivation for Using

- Maintaining consistency between related objects is necessary when a system contains a collection of cooperating classes.
- This consistency shouldn't be accomplished through tightly coupling the classes since this reduces the reusability of those tightly coupled classes.
- Needs to be scalable. There should also be no limit on the number of objects that depend on one or more other objects.

Example of the Problem:

- You are coding an app in which a weather station updates three objects, one that displays current conditions, one that calcs statistics over time (up to date), and one that makes a forecast.

- Here is the obvious approach:

```
public class WeatherData {
        [declarations and getters/setters omitted]

        public void measurements changed(){
                float temp = getTemperature();
                float humidity = getHumidity();
                float pressure = getPressure();

                currentConditionsDisplay.update(temp, humidity, pressure);
                statisticsDisplay.update(temp, humidity, pressure);
                forecastDisplay.update(temp, humidity, pressure);
    }
}
```

Freeman, Freeman, Sierra, and Bates, Head First Design Patterns.O'Reilly 2004

Problems With The Obvious Approach

```
public void measurementsChanged(){
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
}
```

Problems:

- Area that is likely to change is mixed with area that is not likely to change
- update() calls are coded to concrete objects, not types
- Need to change code if the subscribers change

Observer addresses these problems

Three Major Aspects of Observer

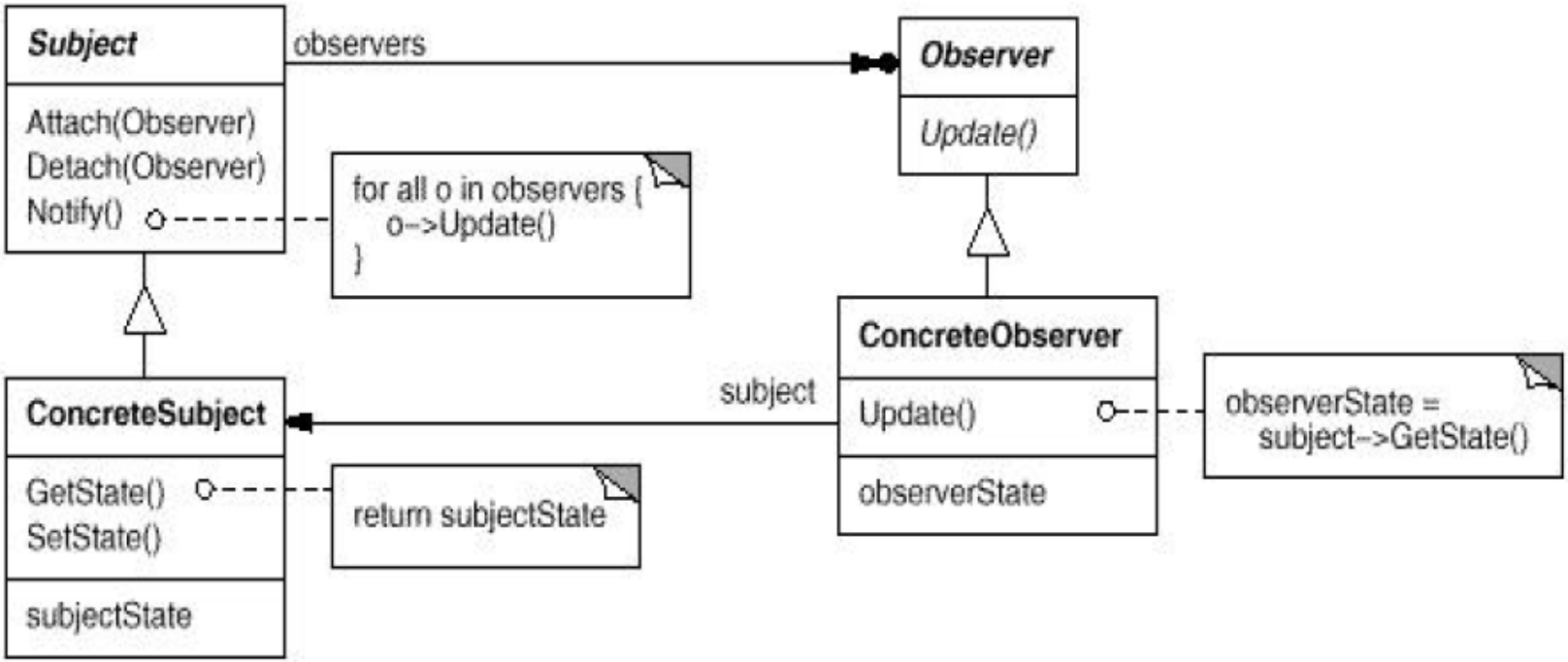
The Subject, which is the object being observed

The Observer, which observes a Subject

Relationship between 1 and 2:

attach/detach (or subscribe / unsubscribe) and update

Generalized Structure



Generalized Structure (cont.)

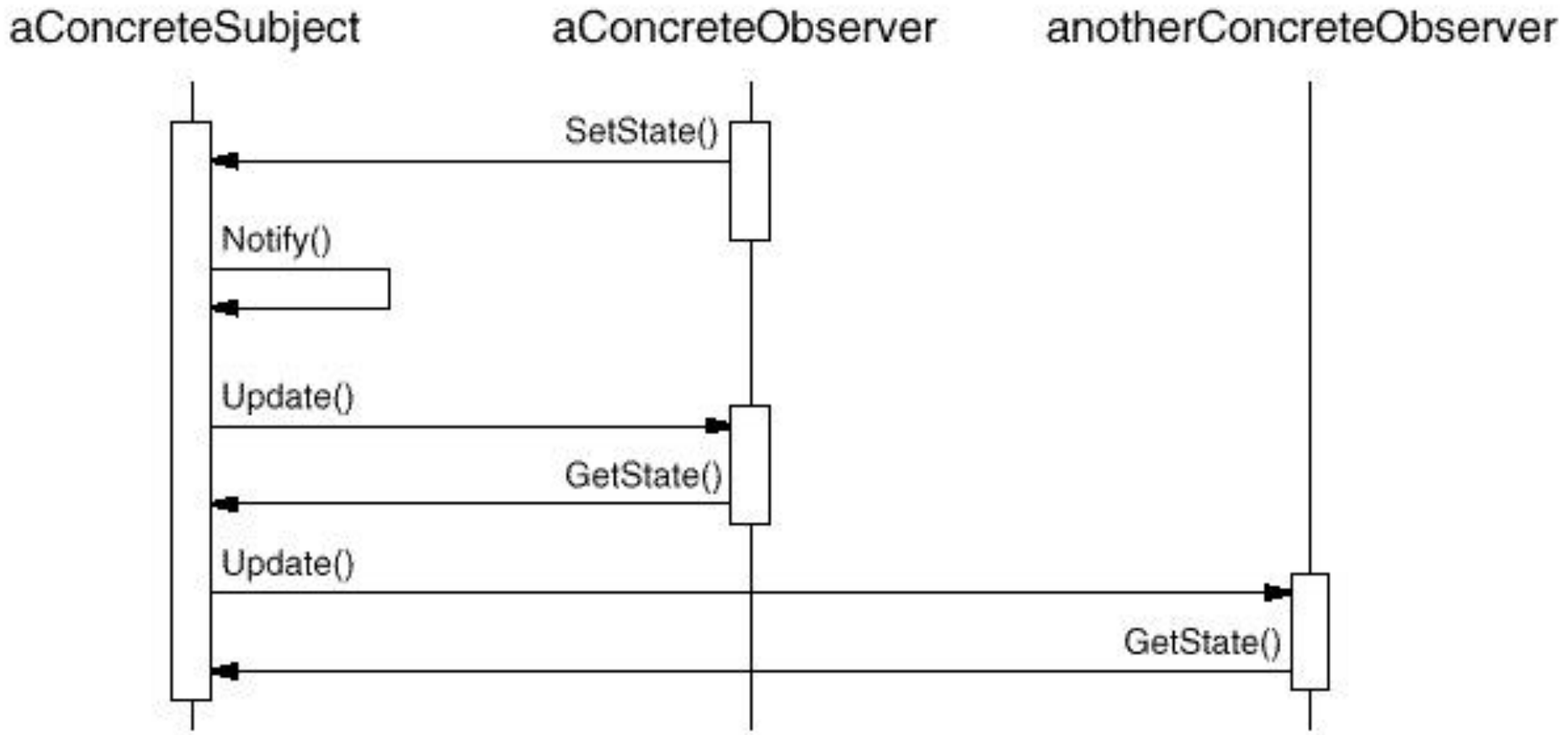
- Subject
  - Interface for ConcreteSubjects
  - Requires implementations to provide at least the following methods:
    - subscribe / attach
    - unsubscribe / detach
    - notify all observers of state changes

- ConcreteSubject
  - Implements the Subject interface
  - Maintains direct or indirect references to one or more ConcreteObservers
  - Keeps track of its own state
  - When its state changes it sends a notification to all of its Observers by calling their update() methods

Generalized Structure (cont.)

- Observer
  - Interface for ConcreteObserver objects
  - Requires an update method

- ConcreteObserver
  - This is the actual object that is observing the state of the ConcreteSubject.
  - The state that it maintains should always be consistent with the state of its Subject.
  - Implements update() method.


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

Two Ways to Implement Updates

- **The Push Model**
  - Subject sends all of the necessary information about any of its changes to all the Observers.
  - *Pushes* information to the Observer as parameter with the update() method.
  - Requires assumptions about what the Observers need to know.
  - May need to allow for subscription to relevant changes only, but this adds complexity

- **The Pull Model**
  - The Subject sends an indication to the Observer that a change has occurred.
  - Observers use public methods of Subject to query information they want
  - It is up to the Observer to *pull* all of the necessary information from the Subject in order to effect any relevant changes.
  - Subject requires fewer assumptions about what the observers want to know

General Implementation

- Subjects can track Observers through ArrayLists or other data structures.
- Observers can track multiple Subjects and get different data from each.
- Pull model uses an update method that takes a reference to Subject as a parameter.
- The Subject should trigger updates when its state changes.

General Implementation

Methods that change state may call a stateChanged() method:

```
public void notifyObservers(){
    for(Observer o: observers)
        o.update();
}

public void stateChanged(){
    // do other things

    notifyObservers();

    // do whatever else still needs doing
}

public void setMeasurements(arguments....) {
    // set instance variables first

    stateChanged();
}
```

Simple Example: Swing Button With Listeners

- Swing JButtons are Subjects; Listeners are Observers
- JButton extends AbstractButton, an abstract class that requires methods to add and remove listeners, as well as several types of notify() methods
- ActionListener requires that implementers have actionPerformed() method (update())
- Can add as many listeners as you like to JButton, as long as they implement ActionListener

Familiar Example: Swing Button With Listeners

```
public class SwingObserverExample{
        Jframe frame;
        [stuff omitted]

        public void go() {
                frame = new JFrame();
                JButton button = new JButton("Should I do it?");
                button.addActionListener(new AngelListener());
                button.addActionListener(new DevilListener());
                frame.getContentPane().add(BorderLayout.CENTER, button);
                [frame property code omitted]
        }

// using inner classes in this very simple example
        class AngelListener implements ActionListener {
                public void actionPerformed(ActionEvent event) {
                        System.out.println("Don't do it");
                }
        }

        class DevilListener implements ActionListener {
                public void actionPerformed(ActionEvent event) {
                        System.out.println("Come on, do it!");
                }
        }
}
```

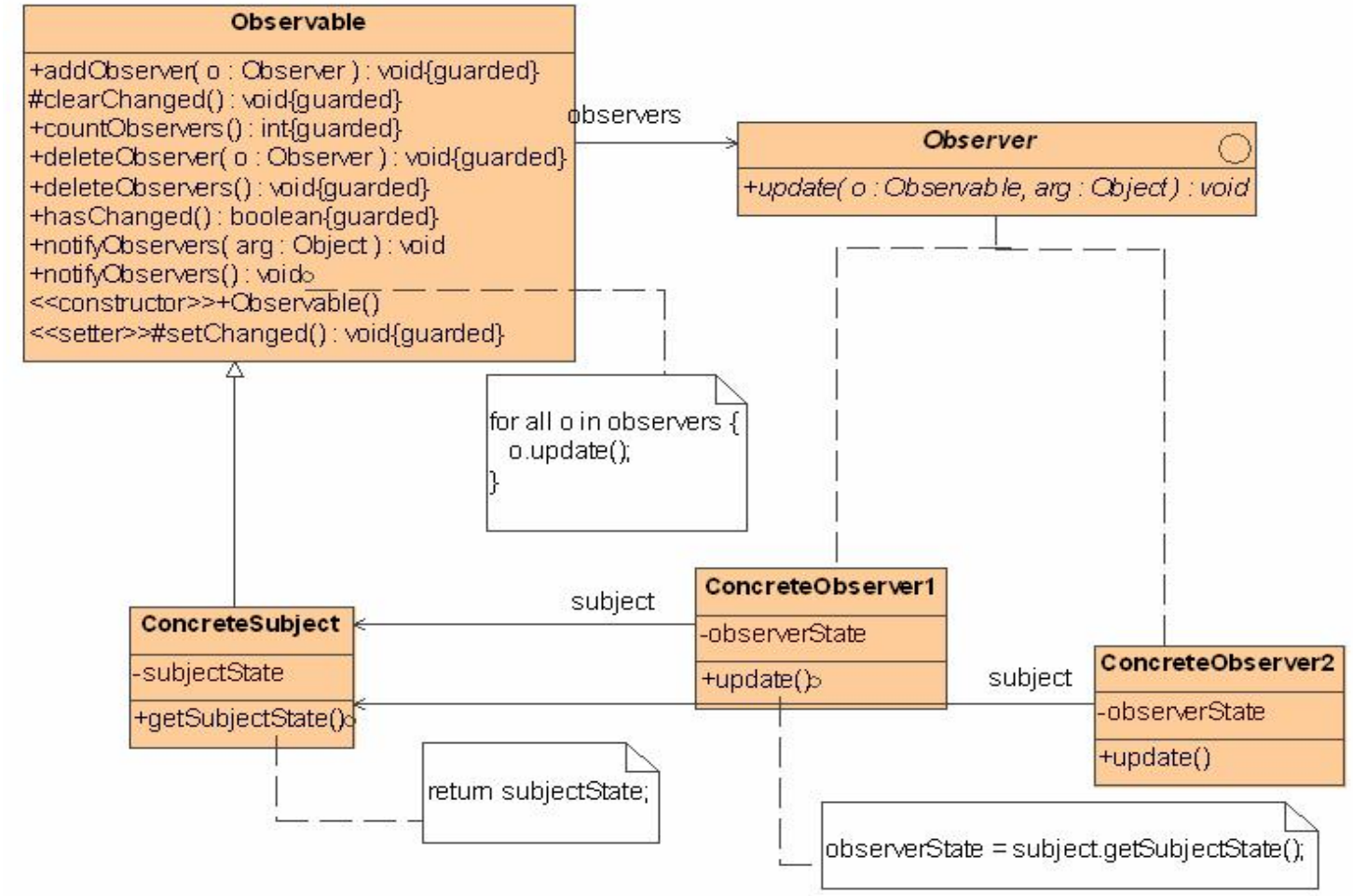
When we click the button, both listeners are notified and take action.

Freeman, Freeman, Sierra, and Bates, Head First Design Patterns,O'Reilly 2004, p. 73

Implementation in Java

- Java has built-in support for Observer
- **java.util.Observable** class can be extended by a Subject
- **java.util.Observer** interface can be implemented by a class that wants to observe a Subject

UML Diagram for Observable/Observer Classes



Methods in java.util.Observable

- Observable()
  - Creates an Observable object (Subject) with no Observers initially
- setChanged()
  - Indicates that this Subject has changed in some way.
- hasChanged()
  - Returns True if the setChanged() method has been called more recently than the clearChanged() method. Returns False if otherwise.
- clearChanged()
  - Indicates that this object is done notifying all of its observers of its most recent changes. It is called automatically by notifyObservers() method.
- countObservers()
  - Returns the number of objects that are Observing this Subject.

Methods in java.util.Observable (cont.)

- addObserver(Observer o)
  - Adds the passed Observer object to the list of Observers kept by the Subject
- deleteObserver(Observer o) / deleteObservers()
  - Removes the passed Observer object or all of the Observer objects respectively from the list of Observers kept by the Subject

Methods in java.util.Observable (cont.)

- notifyObservers(Object arg) / notifyObservers()
  - If this Subject has changed, this method notifies all of its Observers and then calls the clearChanged() method. When given an arg as a parameter in the function call, the Observer knows which attribute of the Subject has changed otherwise the Observer can be notified without specifying an arg.

Methods in java.util.Observer

- update(Observable o, Object arg)
  - Called when the Subject has changed. o is the Subject in question, and arg is an argument that can be passed to tell the Observer which attribute of the Subject has changed.

Limitations of Built-In Implementation

- Observable is a class, not an interface
  - Can't add its behavior to a concrete class that subclasses something else.
  - Since there is no Observable interface, you can't create an impl that works with Observer but doesn't subclass Observable.
- Can't compose another class that has an Observable since setChanged() is protected

Benefits of the Observer Pattern

- Minimal coupling between the Subject and Observer Objects.
- Many Observers can be added to a Subject without having to modify the Subject.
- Reuse of Subjects without needing to also reuse any of their Observers. The opposite also holds true.
- The only thing a Subject needs to keep track of is its list of Observers.
- The Subject does not need to know the concrete classes of its Observers, only that each one implements the Observer interface.

Trouble Spots

- Cascading notifications if Observers update their own clients or if they can also make changes to the Subject.
- Repeated notifications when sequences of changes occur.
- “Dangling references” to Subjects or Observers when either type are manually deleted in non-garbage collected environments. Need to notify Observers when Subjects are deleted and vice-versa.

Trouble Spots

- Subject state must be self-consistent before calling notify(), especially with pull model.
- Careful not to push irrelevant information on observers with push model.
- If update() fails, the Observer won't know that it missed potentially important information

References

Books:

- Freeman, Freeman, Sierra, and Bates, Head First Design Patterns,O'Reilly 2004, p. 73
- Gamma, Helm, Johnson, and Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley 1995.

Websites:

Activity Diagrams:

- http://www.agilemodeling.com/artifacts/activityDiagram.htm
- http://computersciencesource.wordpress.com/2010/03/15/software-engineering-activity-diagrams/
- http://www.cse.unt.edu/~rgoodrum/Teaching/2009/fall/4910/website%20files/handouts/umlbasicsp2_actdg.pdf
- http://www.devx.com/ibm/Article/21615
- http://edutechwiki.unige.ch/en/UML_activity_diagram
- http://www.sa-depot.com/?p=158
- http://sourcemaking.com/uml/modeling-business-systems/external-view/activity-diagrams
- http://www.uml-diagrams.org/activity-diagrams.html

Observer:

- http://www.blackwasp.co.uk/Observer.aspx
- http://www.cs.clemson.edu/~malloy/courses/patterns/observer.html
- http://www.codeproject.com/KB/architecture/Observer_Design_Pattern.aspx
- http://java.dzone.com/articles/observer-pattern?utm_source=am6_feedtweet&utm_medium=twitter&utm_campaign=toya256ForRSS
- http://www.patterndepot.com/put/8/observer.pdf
- http://sourcemaking.com/design_patterns/observer
- http://userpages.umbc.edu/~tarr/dp/lectures/Observer.pdf
- http://en.wikibooks.org/wiki/Java_Programming/Design_Patterns#Observer
- http://www.wohnklo.de/patterns/observer.html