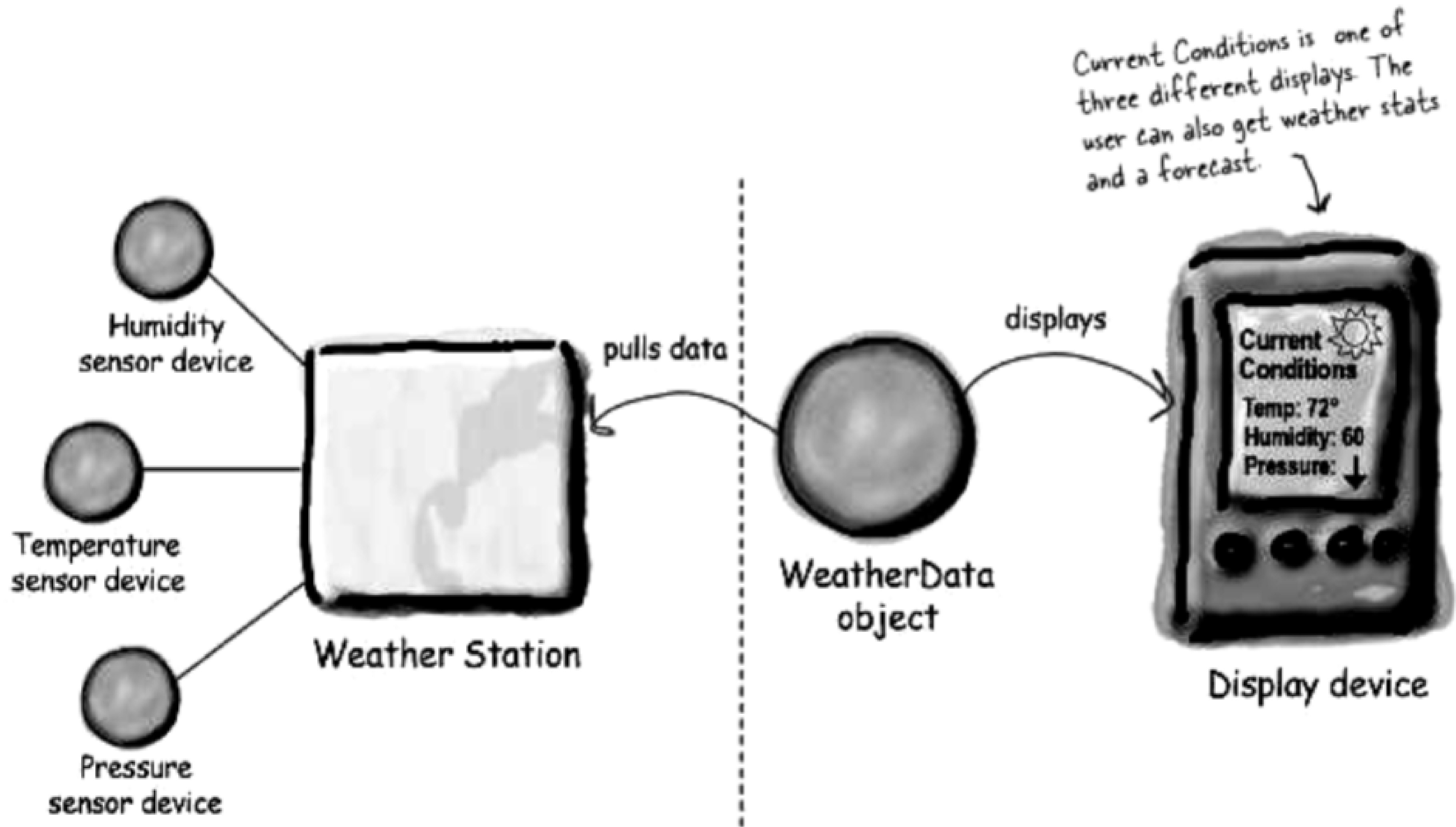
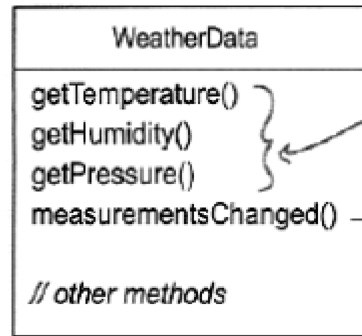
Tecniche di Progettazione: Design Patterns

GoF: Observer

Weather Monitoring



Weather Data class



These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

The developers of the WeatherData object left us a clue about what we need to add...

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

WeatherData.java

member, this Current Conditions is just
NE of three different display screens.

Display device

Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.

Naif solution

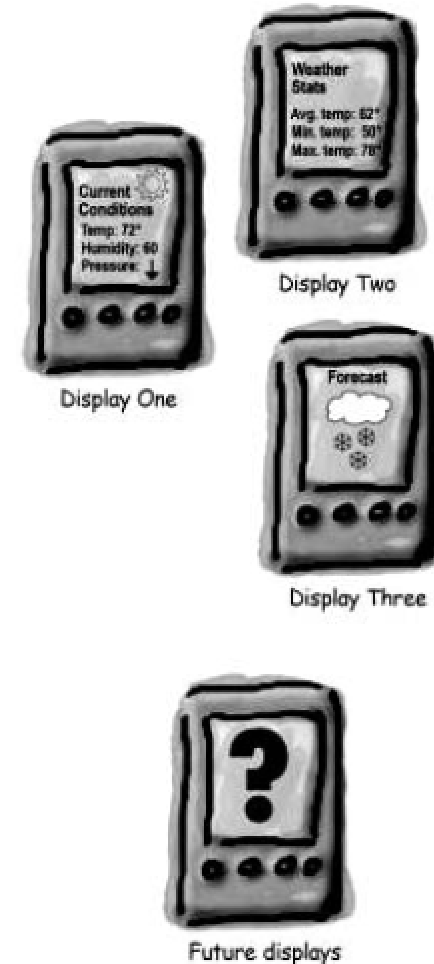
```
public class WeatherData{
...
    public void measurementsChanged() {
        float temperature = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

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

What's wrong?

`measurementsChanged()`

Maintenance!

- System must be expandable.
- Other developers can create new custom display elements.
- Users can add or remove as many display elements as they want to the application.



What's wrong?

```
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);
}
```

Area of change, we need to encapsulate this.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method takes the temp, humidity, and pressure values.

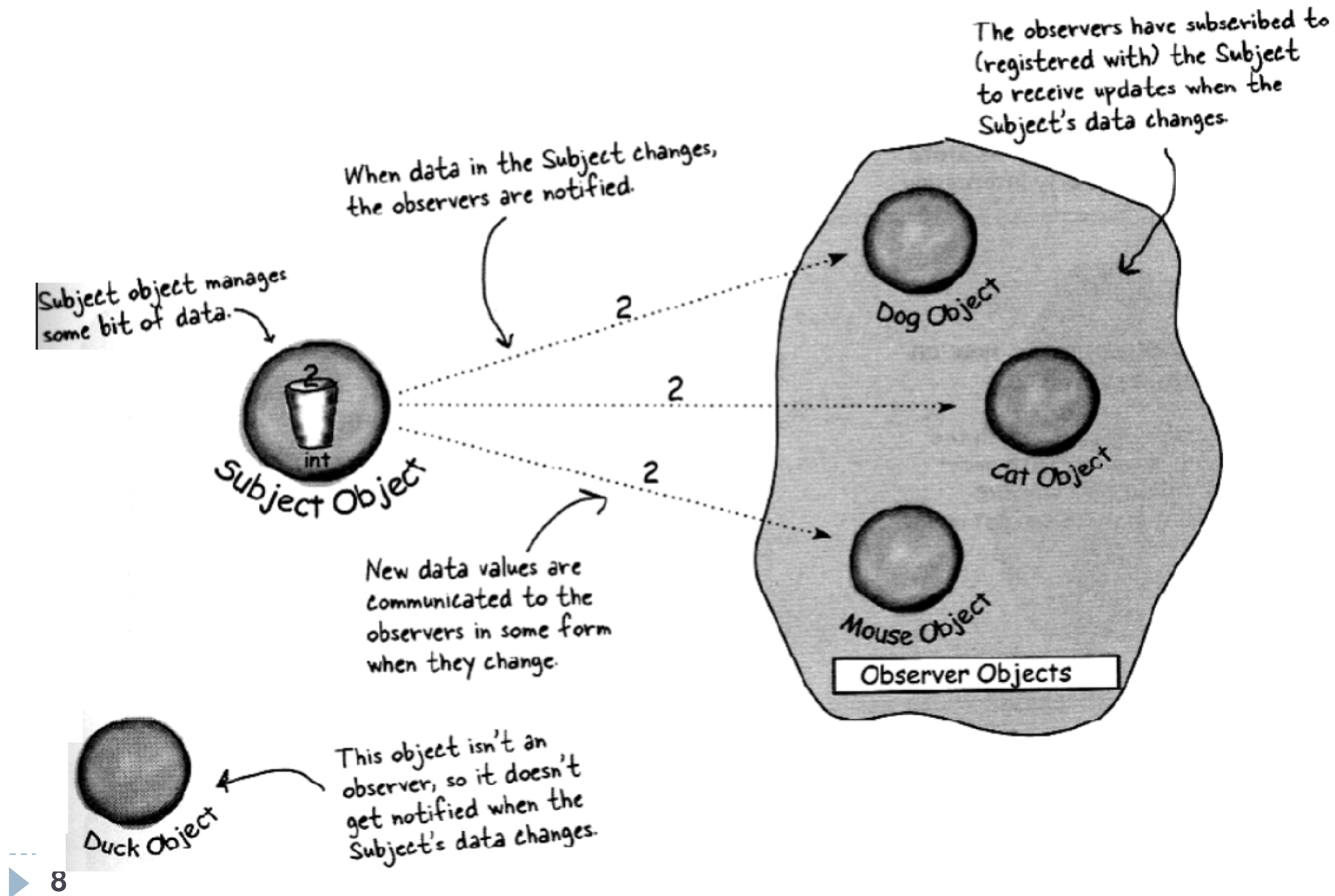
By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

This happens by chance, it can be false in the general case.

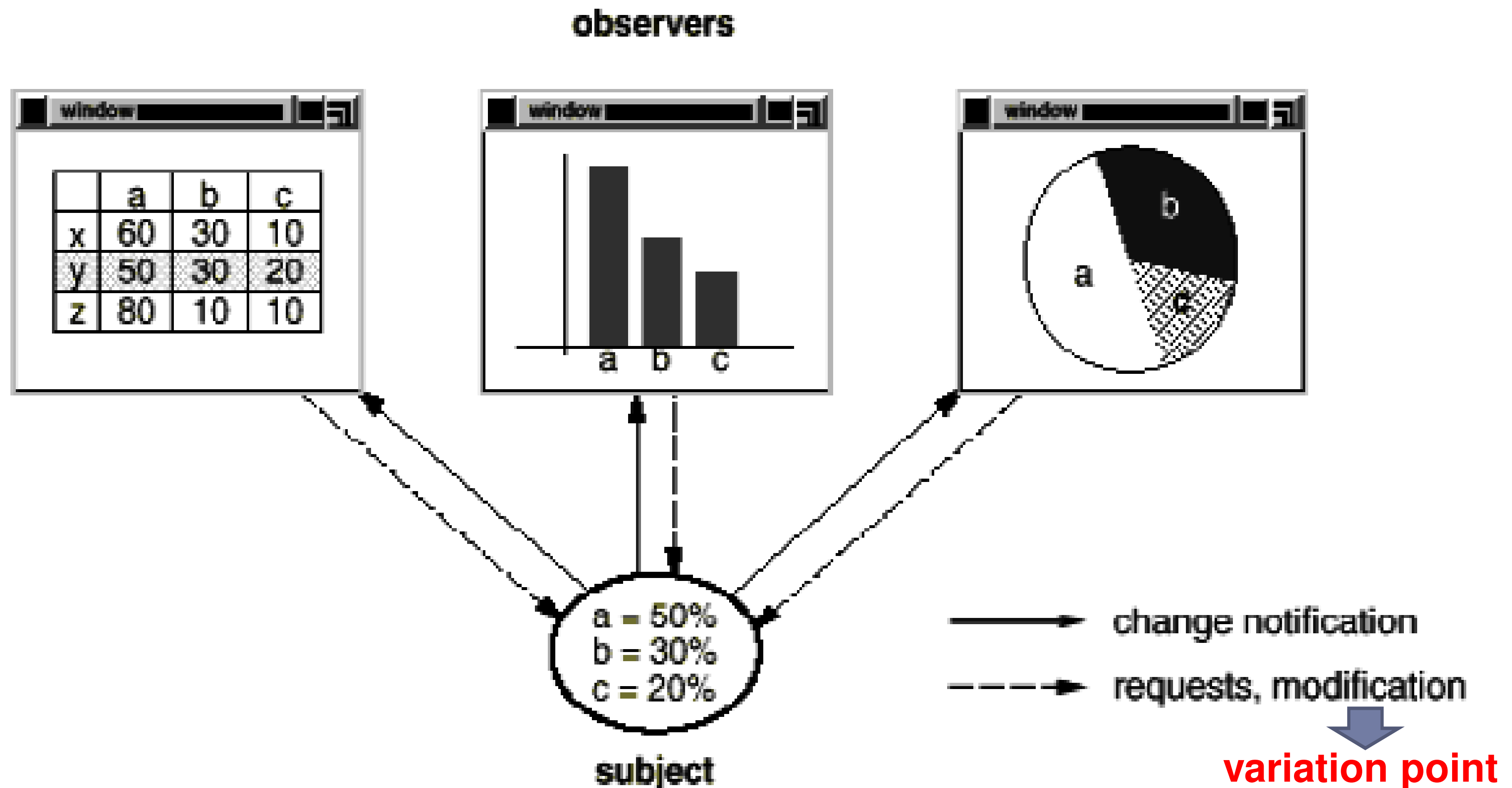
The Observer Pattern

- 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

Publish - Subscribe



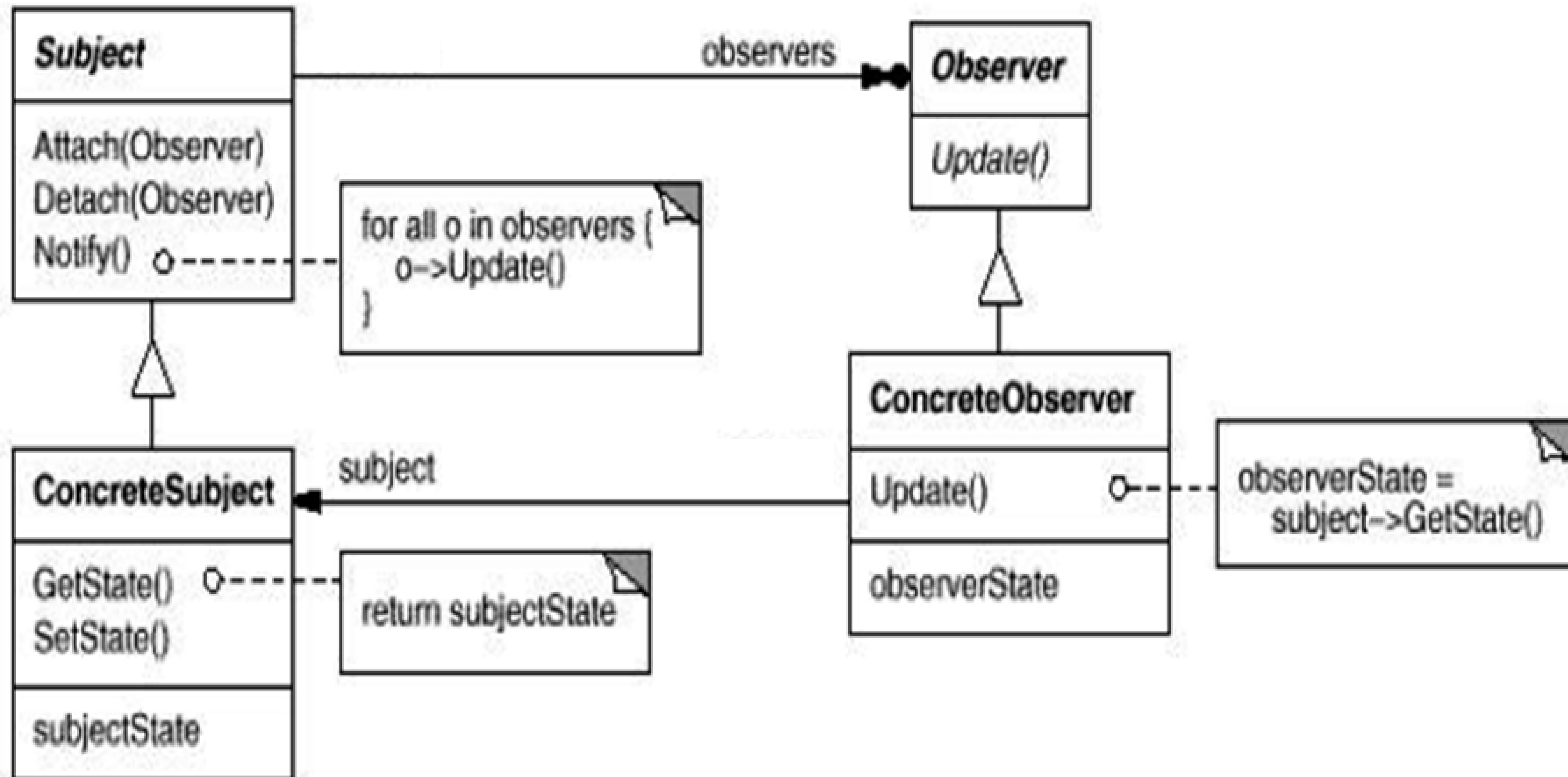
Example



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



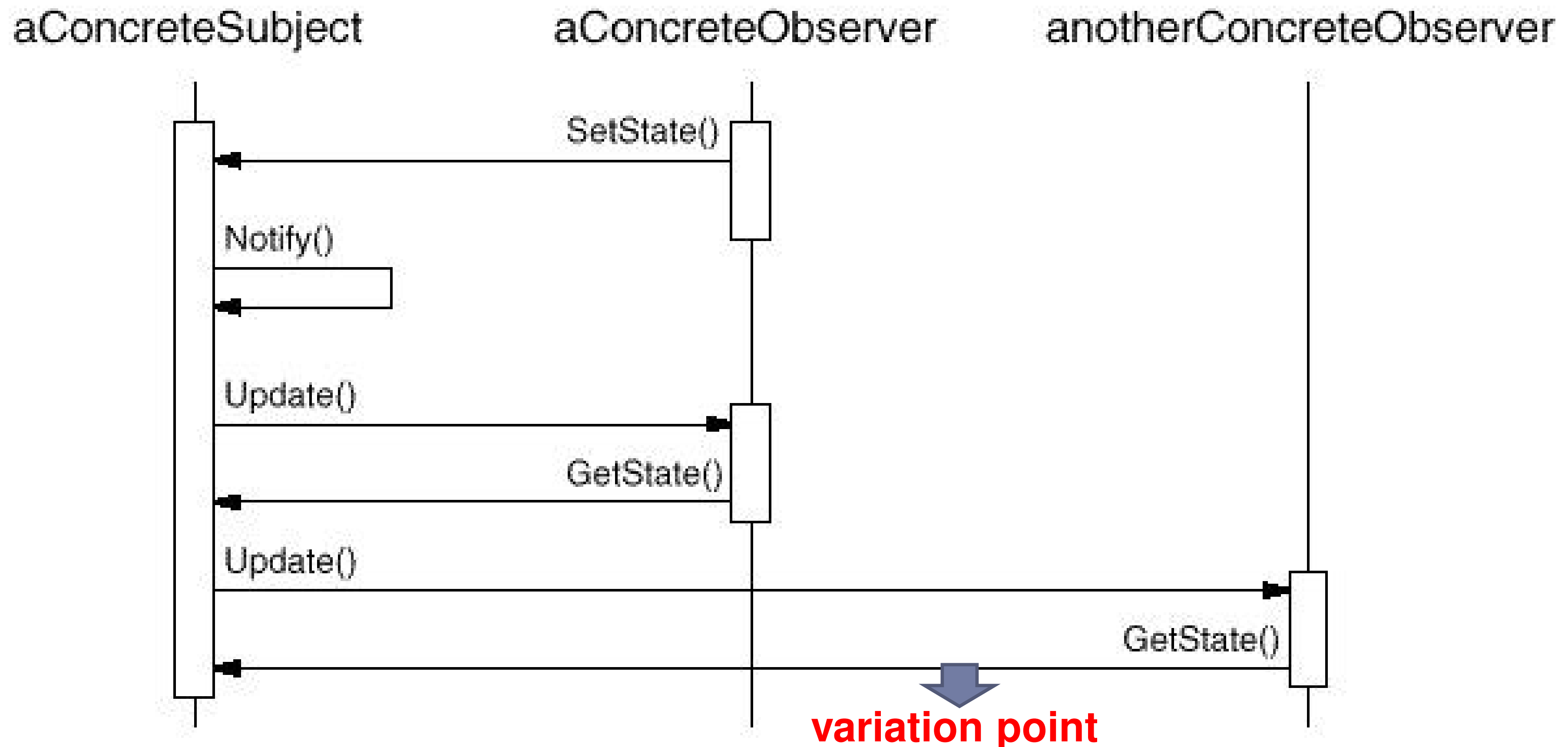
Participants

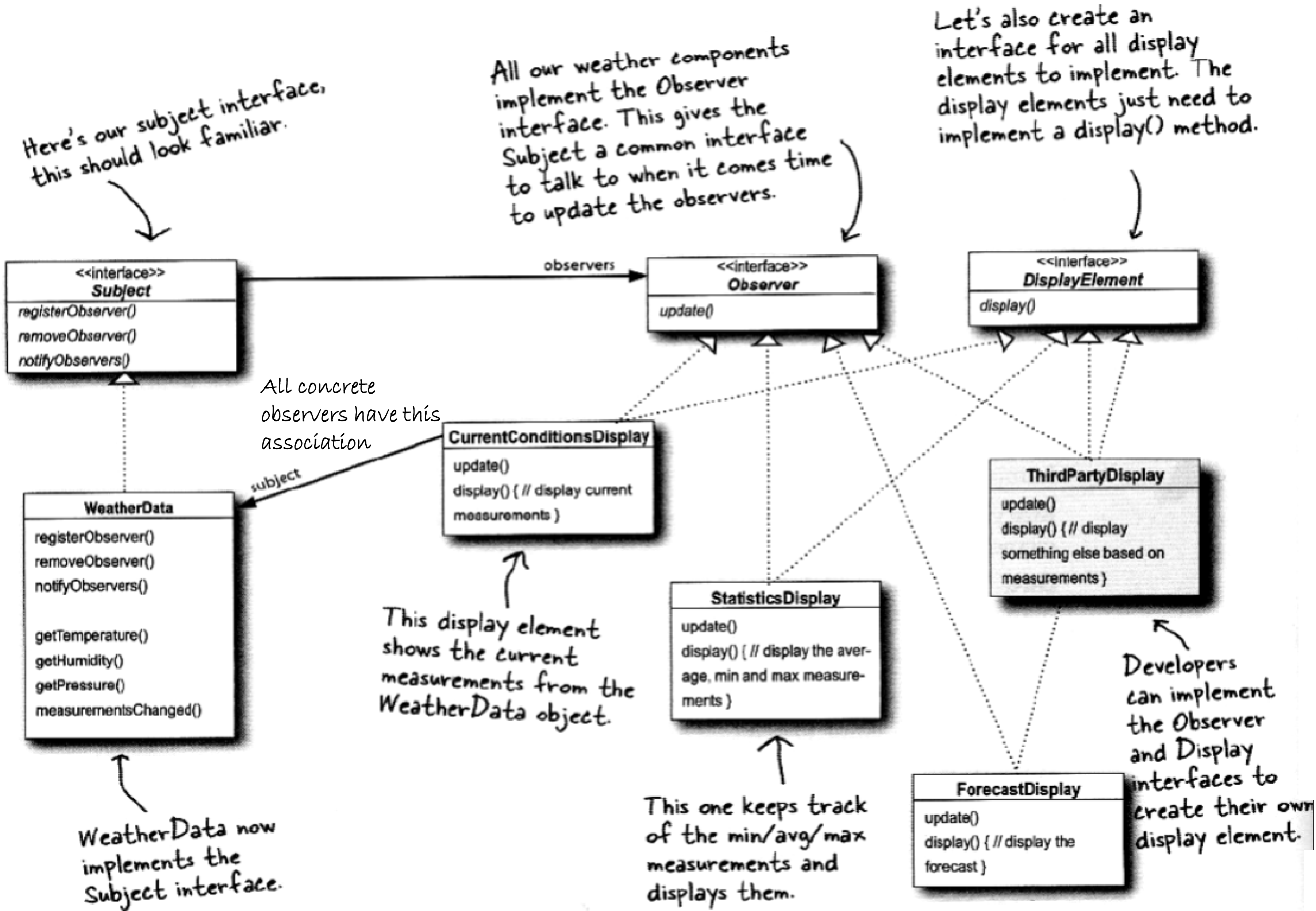
- Subject
  - Keeps track of its observers
  - Provides an interface for attaching and detaching Observer objects
- Observer
  - Defines an interface for update notification
- ConcreteSubject
  - The object being observed
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification 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 –
(not so) special case where observers also cause subject's state change





Here's our subject interface, this should look familiar.
All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.
Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
<<interface>>
DisplayElement
display()
All concrete observers have this association
subject
CurrentConditionsDisplay
update()
display() { // display current measurements }
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
This display element shows the current measurements from the WeatherData object.
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
Developers can implement the Observer and Display interfaces to create their own display element.
ForecastDisplay
update()
display() { // display the forecast }
WeatherData now implements the Subject interface.
This one keeps track of the min/avg/max measurements and displays them.

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 a mediator. (see next lecture, this style is more complex)
- Java 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
  - Pull Model - Very Little
- 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**
  - It can be subclassed to represent an object that the application wants to have observed (the subject).
  - An observable object can have one or more observers.
  - 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.
  - 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.

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.

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: 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

Example: a subject (observable)

```
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);
}
```

(nothing interesting so far....)

Example: a subject (observable)(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));
  }
}
```

Example: an Observer (of name changes)

```
public class NameObserver implements Observer {
  private String name;
  public NameObserver() {
    name = null;
    System.out.println(“NameObserver created: Name is " + name);
}
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!");
}}}
```

Example: 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!");
}}}
```

Example: The TestClass

```
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?
- 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

```
public class SpecialSubject extends ParentClass {
  private String name;
  private float price;
  private DelegatedObservable obs;
  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);    COMMENT
        s.getObservable().addObserver(priceObs);   VS DEMETER
        // 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");
}}
```

DISCUSSION: setChanged()

- The rational for *setChanged()* is clear in «pulling» situations
  - where the subject does not sent notifications, but simply chages its state and shows a flag.
  - Then, periodically, the observer checks the subject's state (through method hasChanged(), which is public – while setChanged() is not) and in case decides an action.

Java 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 Event Model (cont'd)

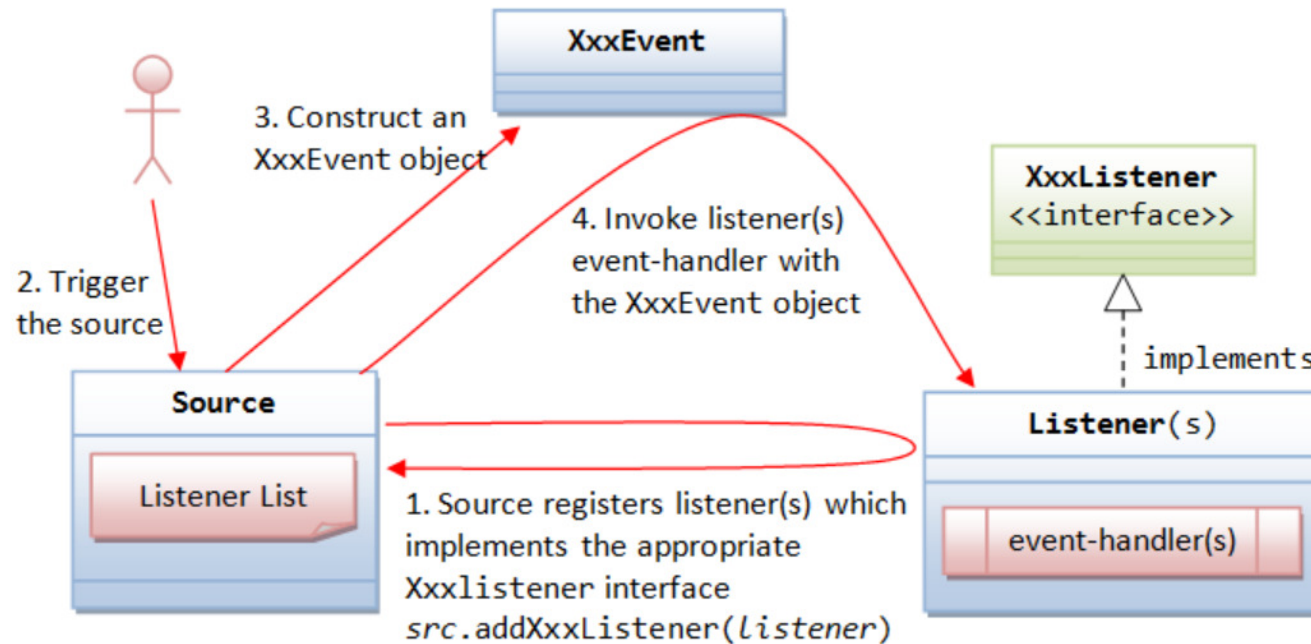
- An event listener must implement an interface which provides the method to be called by the event source when the event occurs
- Unlike the Observer Pattern which defines just the one simple Observer interface, the Java AWT event model has different listener interfaces, each tailored to a different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

Java Event Model (cont'd)

- Listeners for Low-Level Events
  - KeyListener
  - MouseListener
  - MouseMotionListener
  - …
- 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 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.



- The source object registers its listener(s) for a certain type of event.
- Source object fires event *event* upon triggered. For example, clicking a **Button** fires an ActionEvent, **mouse-click** fires MouseEvent, **key-type** fires KeyEvent, etc.
- How the source and listener understand each other? The answer is via an agreed-upon interface. For example, if a source is capable of firing an event called XxxEvent (e.g., MouseEvent) involving various operational modes (e.g., mouse-clicked, mouse-entered, mouse-exited, mouse-pressed, and mouse-released). Firstly, we need to declare an interface called XxxListener (e.g., MouseListener) containing the names of the handler methods

Homework

- You will build a point of sale GUI (may be a simple console) for a restaurant staff.
- Description:
  - Point of sale interfaces are designed to simplify the process of making transactions, often in a retail environment. We often see them used in restaurants where managers can input the menu and waiters can quickly enter customer orders, which are transmitted to the kitchen and cashier. Modern systems usually include a touch screen interface, which we will simulate with a mouse-based GUI (here, may be a console based GUI).

Homework (Cont'd)

- Consider a menu:
  Hamburger 3.99
  Cheeseburger 4.99
  Milk Shake 5.00
  Soda 2.00
  Fries 2.50
  ….
- The program should load the menu and create a panel full of large buttons (ideal for a touchscreen) (here may be a console based command list instead of a button) for each menu item.
- A waiter should be able to click on each menu item (here may write indexes of the commands) to add it to the current order (the subject).

Homework (Cont'd)

- This should add the item to a receipt panel which displays the full order and the total cost. The total cost should be accurate at all times, updated as each item is added (not only when the order is finalized).
  The waiter should be able to click a **Place Order** (notification needed now) button (here in console based application, select place order command from menu list) that simulates transmitting the order to the kitchen and cashier.
- Simply, a screen in the kitchen shows the list of the order and the number of the order. The screen of the cashier shows the order number, names and costs of the ordered items, and the total cost (two different observers).

Homework (Cont'd): the assignment

- Design the classes and their interfaces according to observer pattern. Draw the UML class diagram. (with paper and pencil if you want).

- With push and pull policies
- With or without the JAVA API

- 4 possible solutions, choose the one you better like (and please coordinate yourself to distribute the options)

- Send email with subject DPhomework2