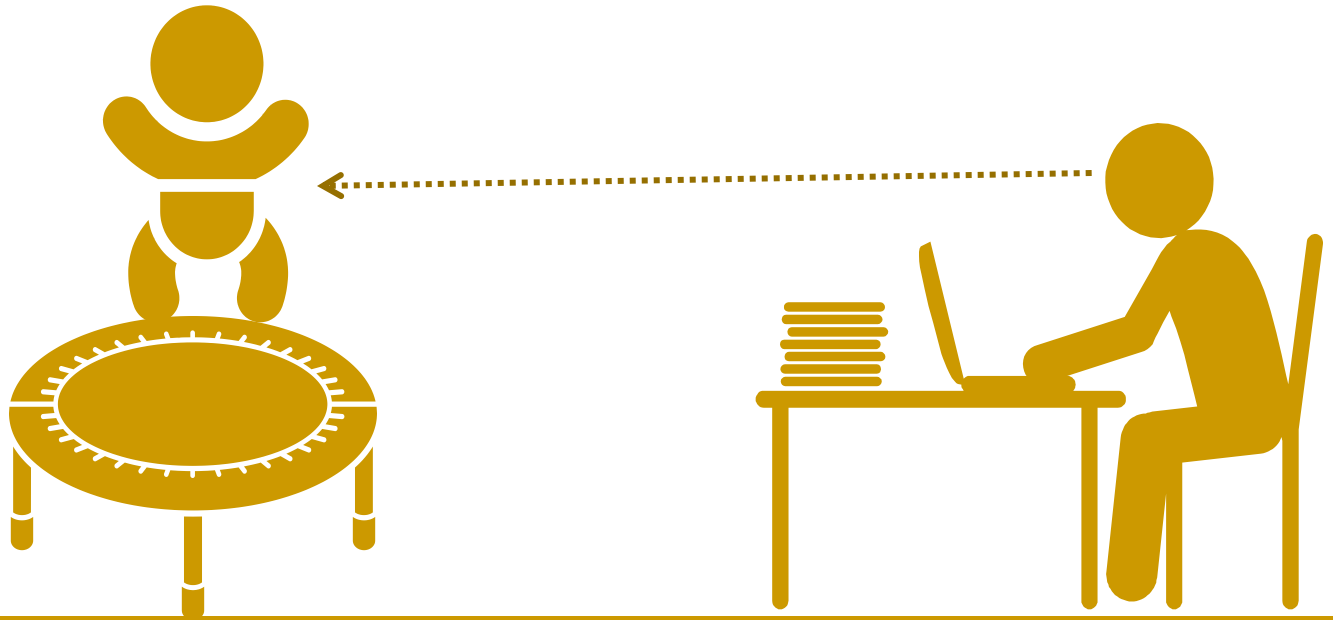
Observer
(观察者, Behavioral Pattern)

Kai SHI

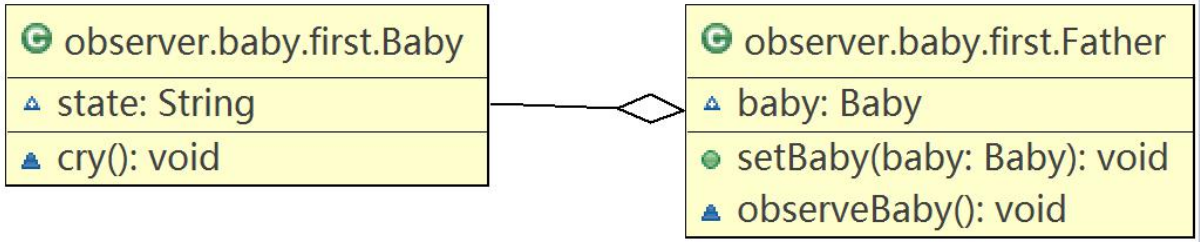
Problem: Looking After Children

- When the baby cries, the father must check the baby immediately.



First Try

code: observer.baby.first



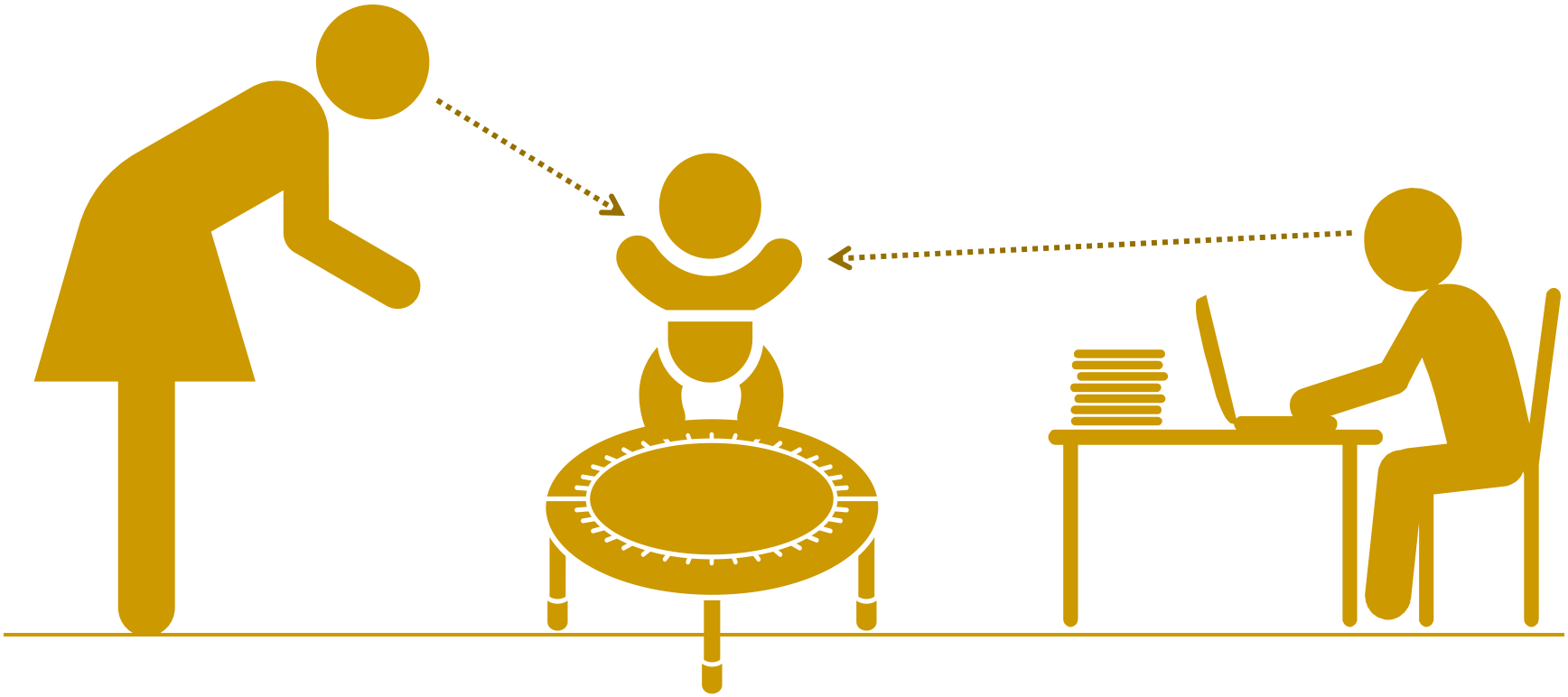

```
public class Baby {
    String state="happy";

    void cry(){
        state="crying";
        new Thread(){
            public void run(){
                while(state.equals("crying")){
                    System.out.println("I will cry until die!");
                }
            }
        }.start();

    }
}
```

```
public class Father {
    Baby baby;

    public void setBaby(Baby baby) {
        this.baby = baby;
    }

    void observeBaby(){
        new Thread(){
            public void run(){
                while(true){
                    if(baby.state.equals("crying")){
                        System.out.println("Do not cry!");
                        baby.state="happy";
                    }
                }
            }
        }.start();
    }
}
```

Loop: waste CPU time

Requirements Change: More Adults

- When the baby cries, both parents could check the baby immediately.



Second Try

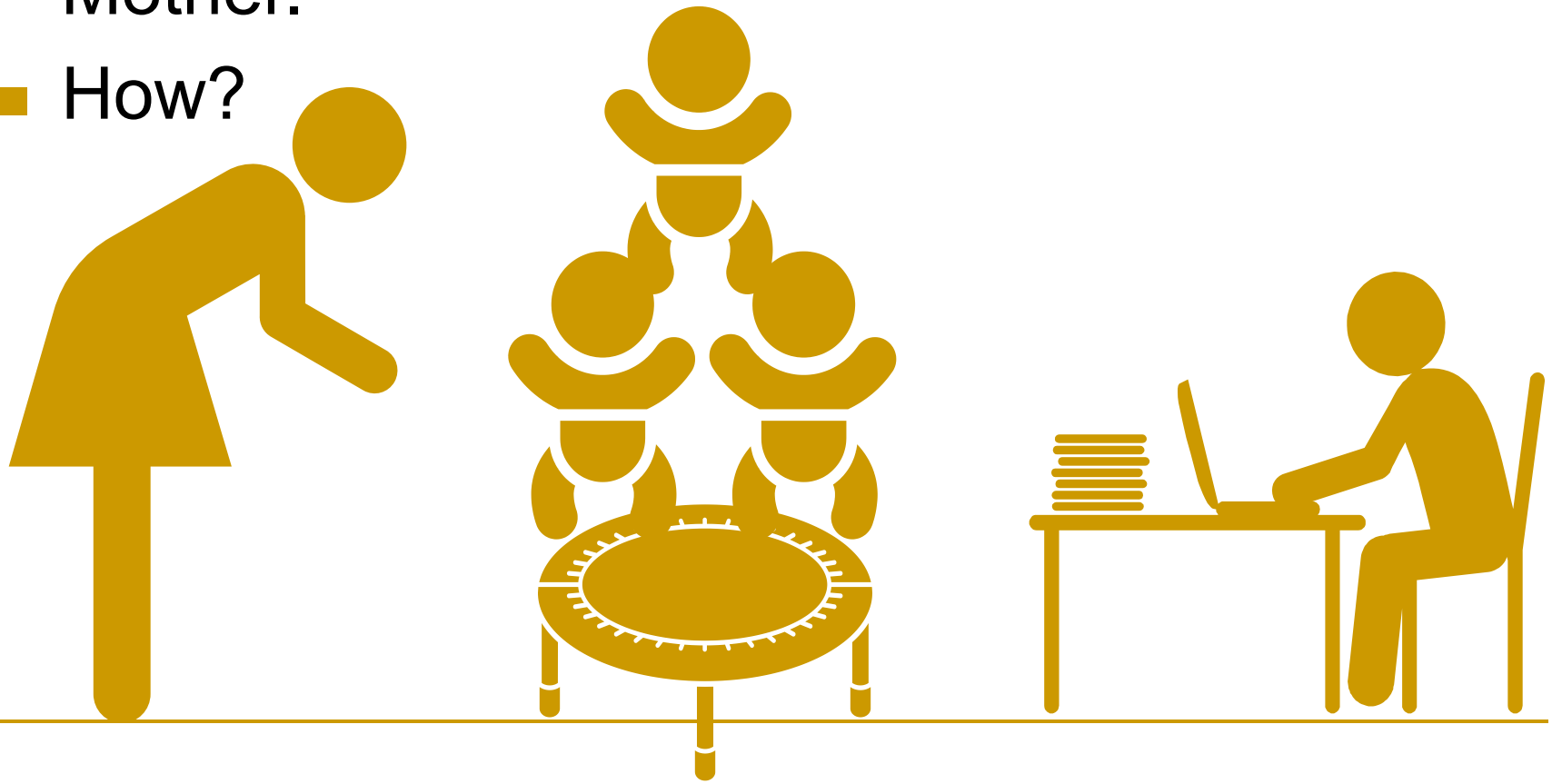
code: observer.baby.second

- It's easy, just add Mother class.

```
public class Mother {
    Baby baby;

    public void setBaby(Baby baby) {
        this.baby = baby;
    }

    void observeBaby(){
        new Thread(){
            public void run(){
                while(true){
                    if(baby.state.equals("crying")){
                        System.out.println("Mother: Do not cry!");
                        baby.state="happy";
                    }
                }
            }
        }.start();
    }
}
```

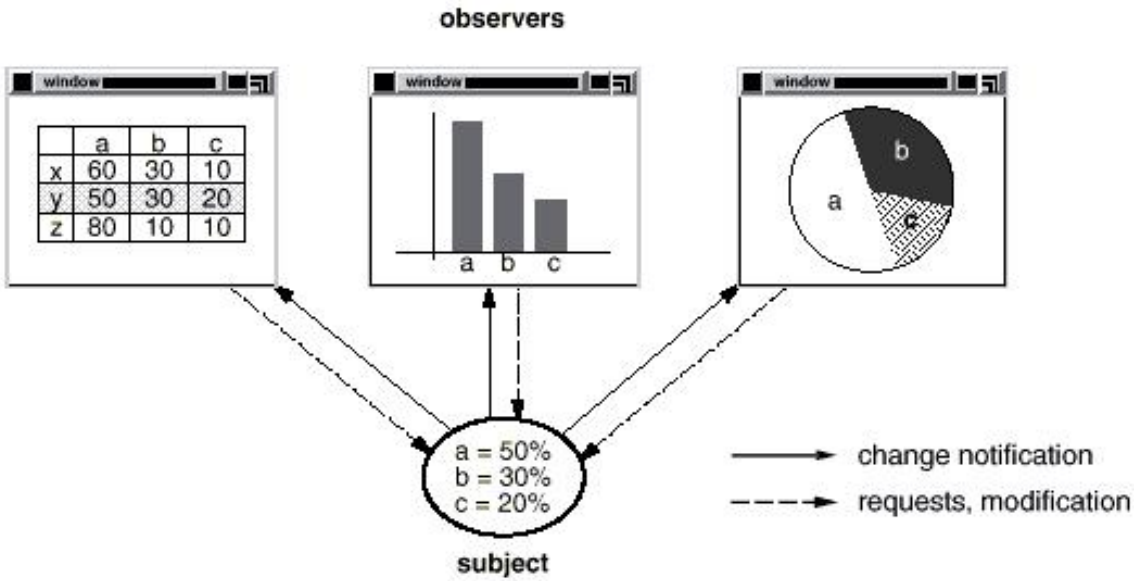
Requirements Change: More Babies

- It's bad. We have to change Father and Mother.
- How?



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.



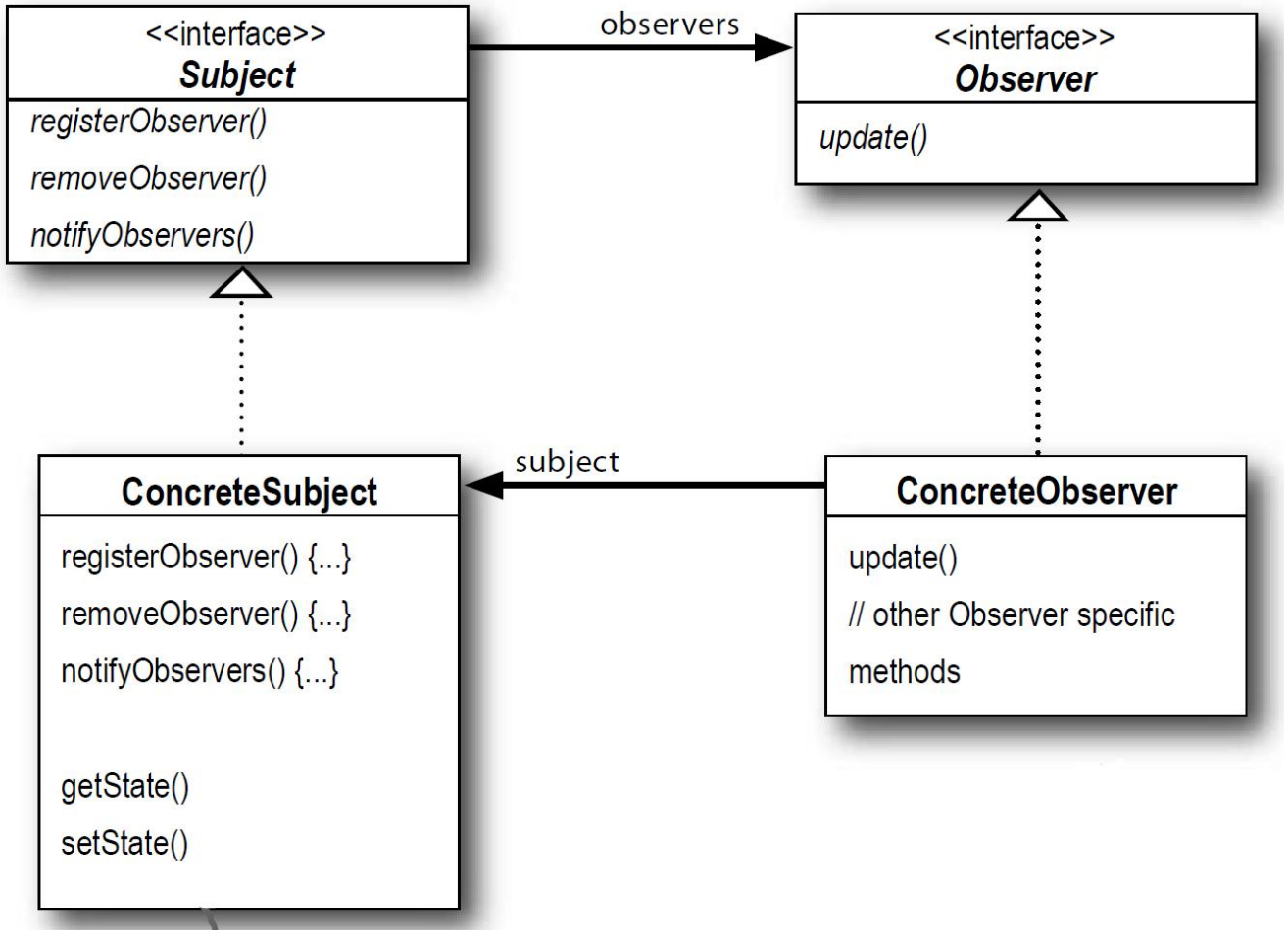
Observer Pattern

- Also Known As
  - Dependents, Publish-Subscribe, Model-View
- Motivation
  - The need to maintain consistency between related objects without making classes tightly coupled.

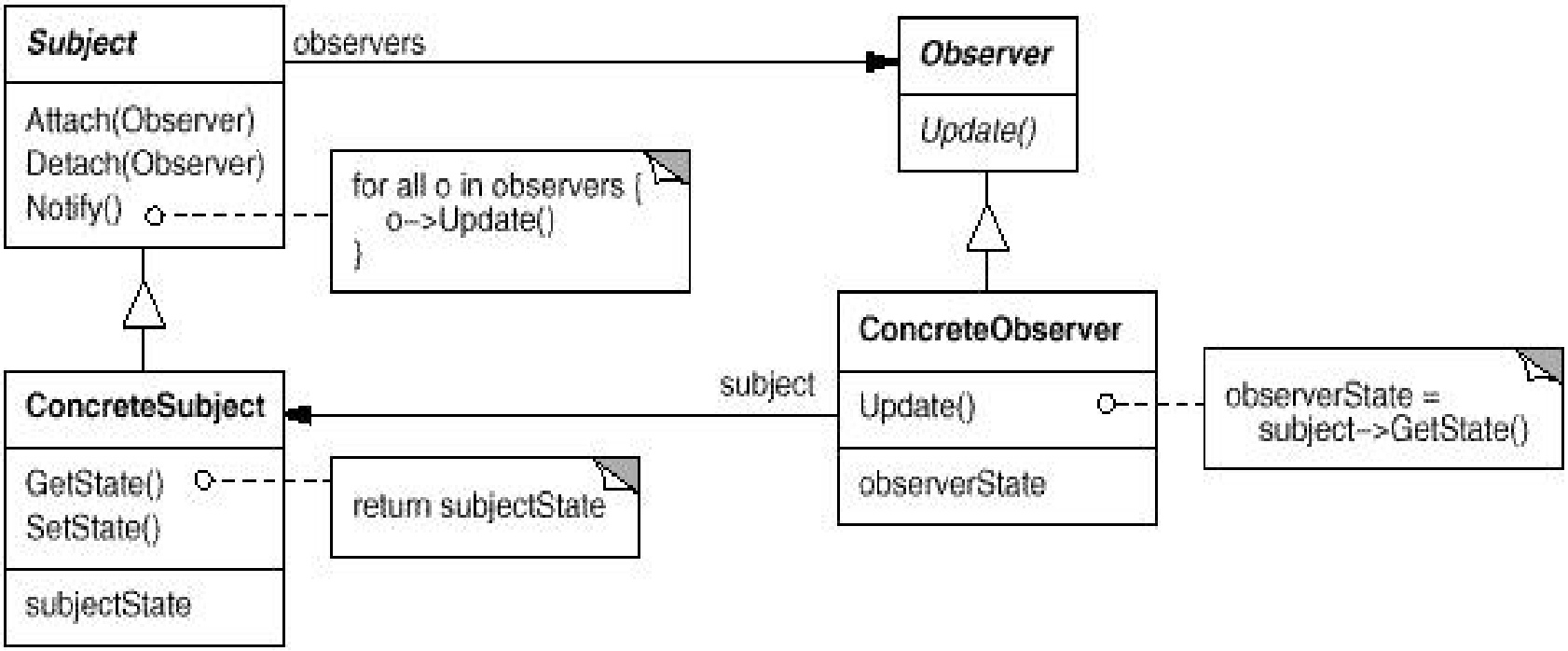
Applicability: Use observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Class Diagram



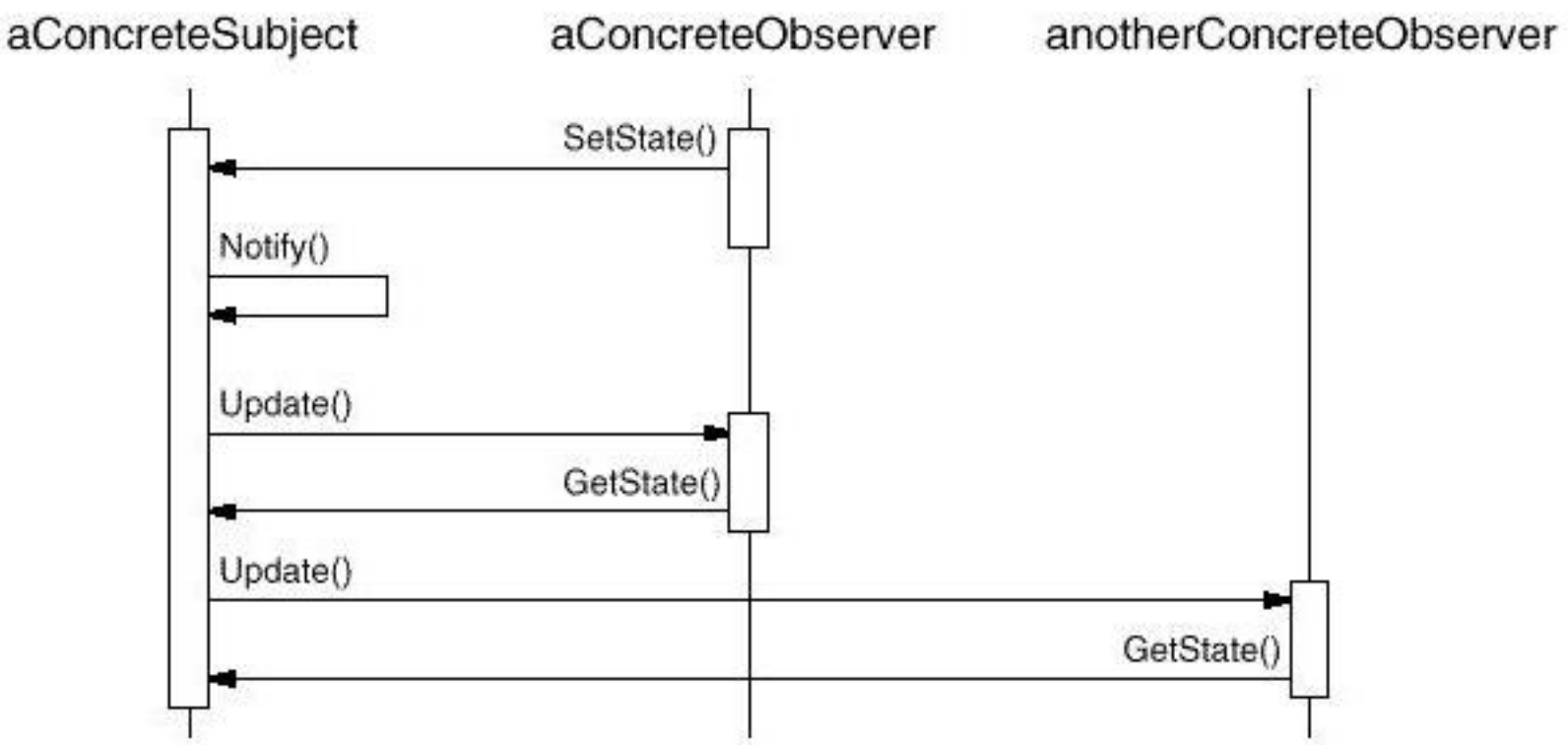
Class Diagram (GoF)



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
- **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

Collaborations



Consequences (1/2): Advantages

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

Consequences (2/2): Drawbacks

- **Unexpected updates**
  - Observers have no knowledge of each other's presence, they can be blind to the cost of changing the subject.
  - A seemingly innocuous (表面上无害的) operation on the subject may cause a cascade of updates to observers and their dependent objects.
  - Moreover, dependency criteria that aren't well-defined or maintained usually lead to spurious (伪造的) updates, which can be hard to track down.
- The simple update protocol provides no details on what changed in the subject. The observers have to discorver what changed themselves, thus we need to deduce the changes.

Implementation Issues (1/2)

- How does the subject keep track of its observers?
  - Array, linked list
- What if an observer wants to observe more than one subject?
  - let 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 (2/2)

- 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!
- 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

Known Uses

- Smalltalk Model/View/Controller user interface framework
  - Model = Subject
  - View = Observer
  - Controller is whatever object changes the state of the subject
- Java AWT/Swing Event Model (We'll see this later.)

Push vs Pull

- Pull model (Observer要自己找什么变了)
  - Emphasizes the subject's ignorance of its observers
  - May be inefficient, because Observer classes must ascertain what changed without help from the Subject.
- Push model (Subject负责安排妥妥的)
  - Assumes subjects know something about their observers' needs
  - Make observers less reusable. because Subject classes make assumptions about Observer classes that might not always be true.

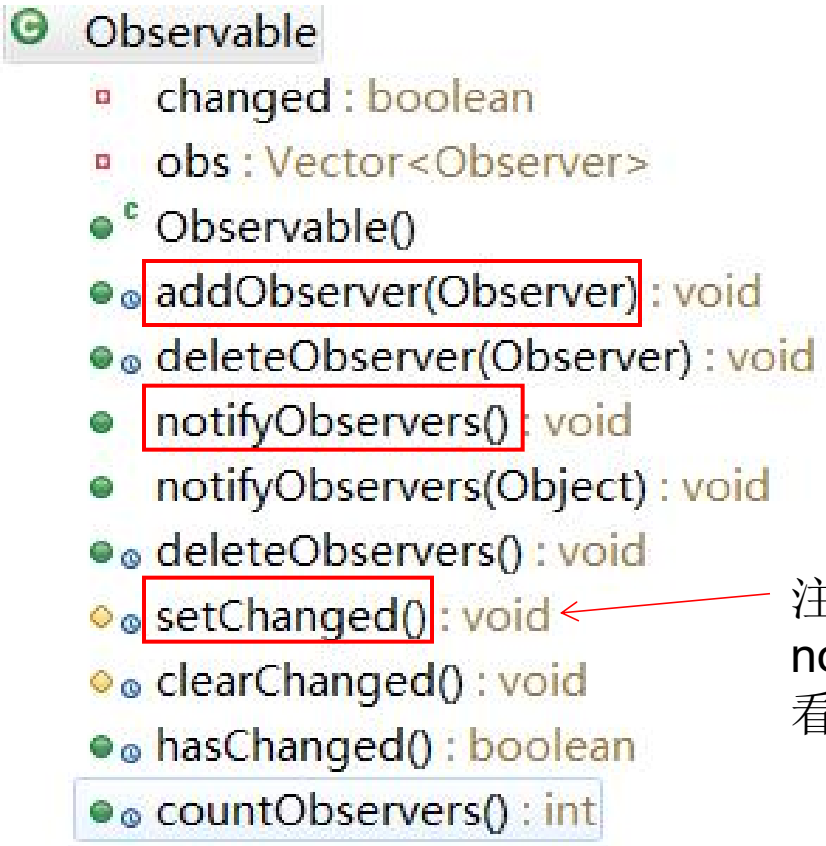
Java's Built-in Observer Pattern (1/3)

```
public interface Observer {
    /**
     * This method is called whenever the observed object is changed. An
     * application calls an <tt>Observable</tt> object's
     * <code>notifyObservers</code> method to have all the object's
     * observers notified of the change.
     *
     * @param   o     the observable object.
     * @param   arg   an argument passed to the <code>notifyObservers</code>
     *                 method.
     */
    void update(Observable o, Object arg);
}
```

i.e., Subject

```
public class Observable {
    private boolean changed = false;
    private Vector<Observer> obs;

    /** Construct an Observable with zero Observers. */

    public Observable() {
        obs = new Vector<>();
    }

    /**
     * Adds an observer to the set of observers for this ob
     * that it is not the same as some observer already in
```

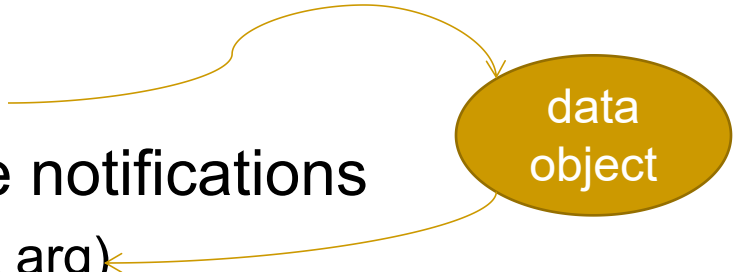
Java's Built-in Observer Pattern (2/3)



注意：只有在setChange()被调用后，notifyObservers()才会去调用update()：稍后看Observable源码

Java's Built-in Observer Pattern (3/3): How to use

- For an Object to become an observer
  - implement the Observer interface
  - addObserver() on Observable object
- For the Observable to send notifications
  - call the setChanged() method to signify that the state has changed
  - call one of two notifyObservers() methods:
    - notifyObservers()
    - notifyObservers(Object arg)
- For an Observer to receive notifications
  - update(Observable o, Object arg)
    - If you want to "push" data to the observers you can pass the data as a data object to the notifyObserver(arg) method. If not, then the Observer has to "pull" the data it wants from the Observable object passed to it.



Problems with java.util.Observable

- Observable is a class
  - Because Observable is a class, we have to subclass it.
- Observable protects crucial methods
  - The setChanged() method is protected, thus we must subclassed Observable if we want to call setChanged().
  - It violates “favor composition over inheritance”

Solve the Problem of Built-in Observer Pattern

- 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

Rewrite “Looking After Babies”

- code: observer.baby.observer

Java AWT/Swing Event Model

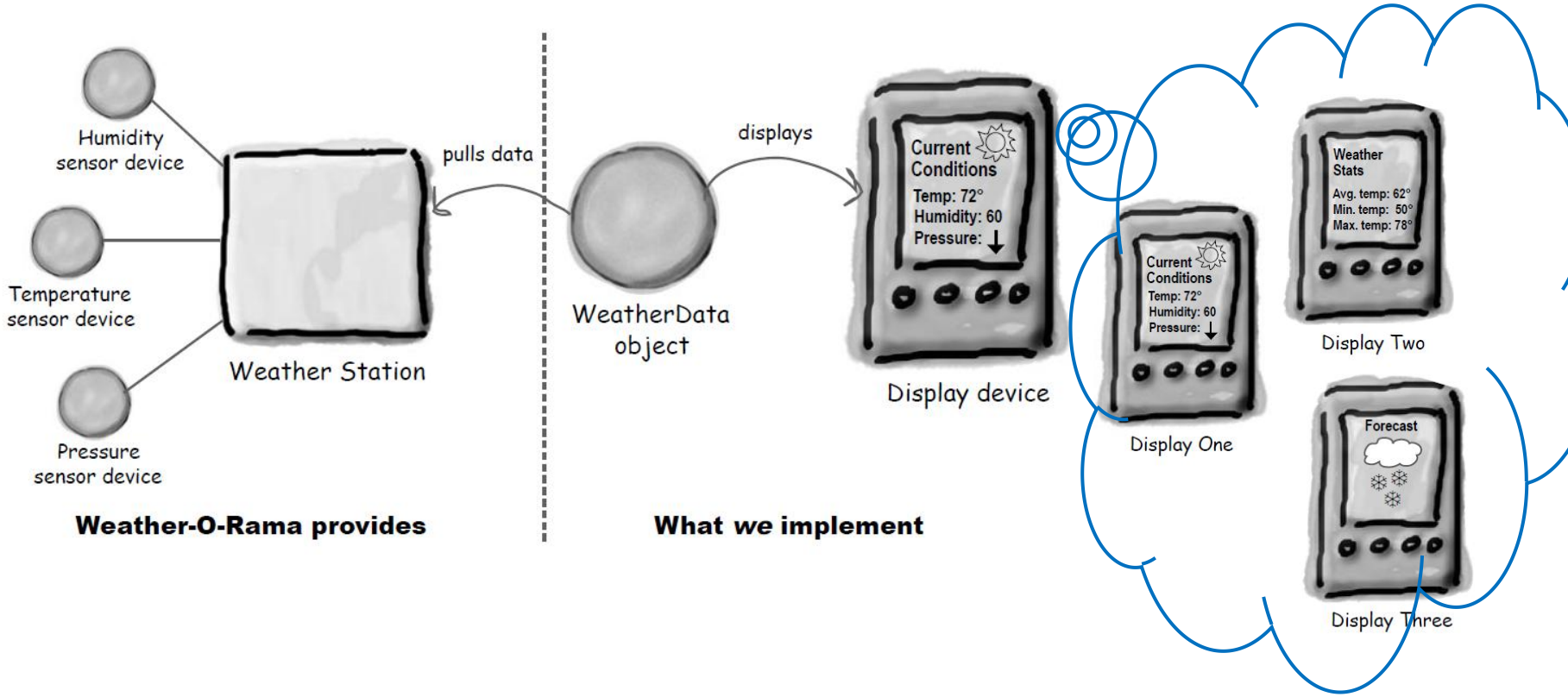
- Java GUI event model based on the Observer Pattern
- GUI components which can generate GUI events are called event sources
- Objects that want to be notified of GUI events are called event listeners
- Event generation is also called firing the event
- Comparison to the Observer Pattern:
  - ConcreteSubject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

Example: JButton

- Example code: observer.gui.Buttontest
- ActionListener
  - actionPerformed method
- JButton extends AbstractButton
  - AbstractButton's addActionListener method
  - AbstractButton's actionPerformed method
    - indeed fireActionPerformed

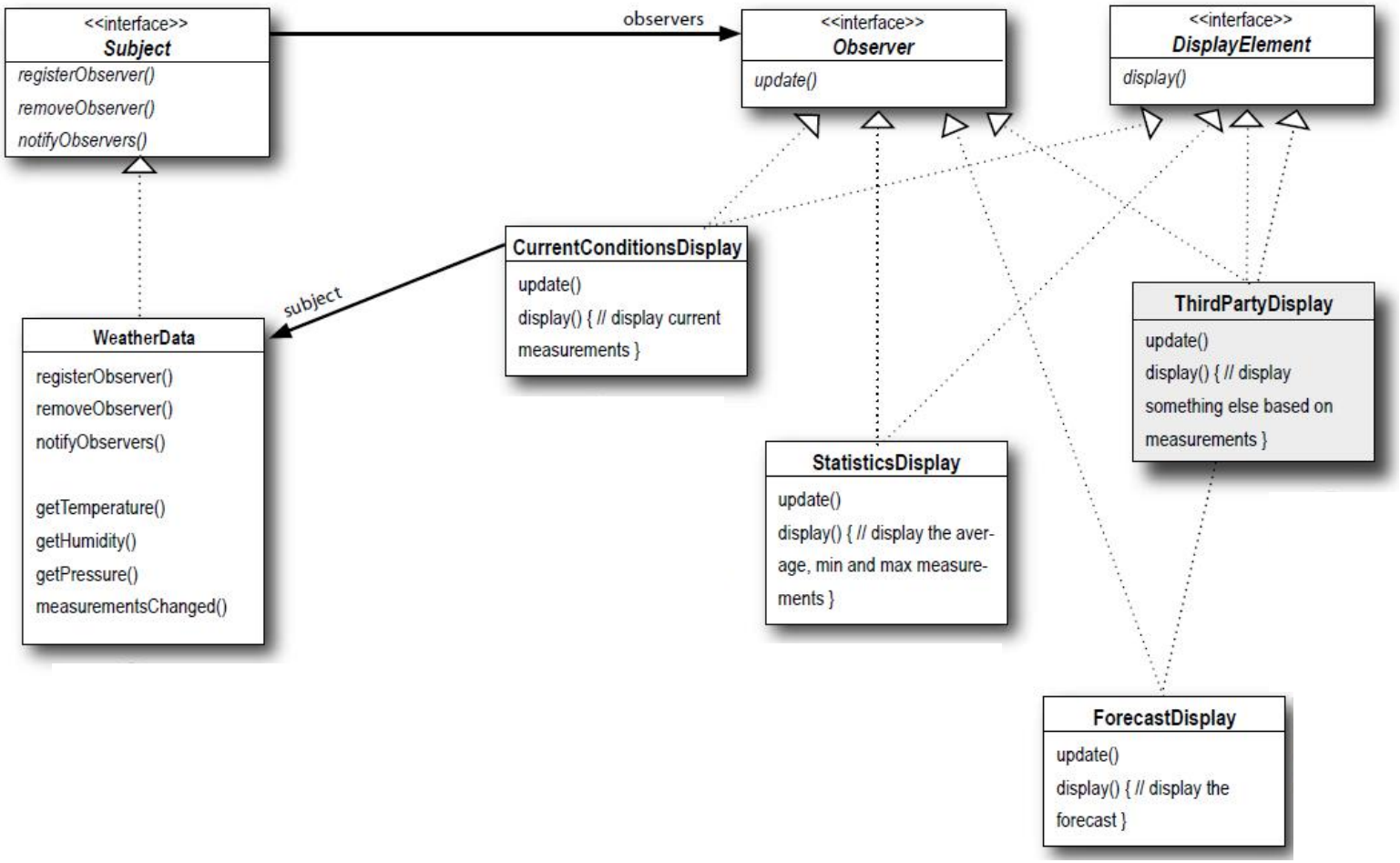
```
for (int i = listeners.length-2; i>=0; i-=2) {
    if (listeners[i]==ActionListener.class) {
        // Lazily create the event:
        if (e == null) {
              String actionCommand = event.getActionCommand();
              if(actionCommand == null) {
                 actionCommand = getActionCommand();
              }
              e = new ActionEvent(AbstractButton.this,
                                  ActionEvent.ACTION_PERFORMED,
                                  actionCommand,
                                  event.getWhen(),
                                  event.getModifiers());
        }
        ((ActionListener)listeners[i+1]).actionPerformed(e);
    }
}
```

Another Question:
The Weather Monitoring Application



- Draw class diagram (do not use Java built-in observer pattern)
- Write the code

Class diagram



Code

- code: net.dp.observer

Q: For each design principle, describe how the Observer Pattern makes use of the principle. (答案在下页)

- Identify the aspects of your application that vary and separate them from what stays the same.
- Program to an interface, not an implementation (DIP: Dependence Inversion Principle)
- Favor composition over inheritance. (CRP: Composite/Aggregate Reuse Principle)

Answers (1/3)

- Q1: Identify the aspects of your application that vary and separate them from what stays the same.
- Ans: The thing that varies in the Observer Pattern is the state of the Subject and the number and types of Observers. With this pattern, you can vary the objects that are dependent on the state of the Subject, without having to change that Subject. That's called planning ahead!

Answers (2/3)

- Q2: Program to an interface, not an implementation (DIP: Dependence Inversion Principle)
- Ans: Both the Subject and Observer use interfaces. The Subject keeps track of objects implementing the Observer interface, while the observers register with, and get notified by, the Subject interface. As we've seen, this keeps things nice and loosely coupled.

Answers (3/3)

- Q3: Favor composition over inheritance. (CRP: Composite/Aggregate Reuse Principle)
- Ans: The Observer Pattern uses composition to compose any number of Observers with their Subjects. These relationships aren't set up by some kind of inheritance hierarchy. No, they are set up at runtime by composition!