


Observer

28.1 The Problem

Often systems have objects that are dependent on some underlying core information: if this information changes then the objects must react accordingly. One such classic example is a spreadsheet where the data in the sheet may be simultaneously displayed as bar charts and pie charts in their own windows. Of course I want these windows to be synchronized i.e. if I change a value in a cell in the spreadsheet then the pie chart and bar chart should immediately redraw to reflect the change. Many other examples exist as for instance

- A central computer to monitor a parking lot may receive information about bought parking time from the lot's set of parking pay stations in order to calculate the number of free parking spaces and display it at the entrance.
- A calendar application shows current time both as text in a status field and as a highlight of entries in the day view. As time passes both need to be updated. It must also pop up reminders for important meetings at the correct time.
- In a UML diagram editor, the object representing an association line between two classes must monitor any movement of either of the connected classes in order to reposition and redraw.

Common to all these examples is that one or several objects can only behave properly if they are constantly notified of any state changes in the monitored object.

 Review the set of computer applications that you normally use. Find examples in which some objects need to be synchronized with others for the application to work correctly.

28.2 A Solution

I can use the compositional design principles and the concepts of *roles and responsibilities*, to find a solution to this challenge. I first note two distinct roles in the problem: the monitored object (e.g. the spreadsheet cell) and the dependent objects (e.g. the pie chart window, the bar chart window, and potentially lots of other windows showing the cell's value in some way). These roles are traditionally termed the **subject** role and the **observer** role. Observers are the dependent objects while the subject contains the state information that they monitor.

Looking for variability, the ③ *consider what should be variable in your design* principle, I conclude that I cannot in advance know the type of processing to make when the subject's state changes; the only thing I know is that the observers will have to make *some kind* of processing. For example, the spreadsheet cell should not know about drawing pie charts or bar charts as this would lead to tight coupling. But it should allow the observers, like the pie chart window and the bar chart window, to discover that it has indeed changed its value. So, the variable behavior is the actual *processing* which must take place whenever the cell changes value—I do not know in advance the kind of diagrams that must be redrawn or the type of recalculations that must be made based on the change. The common behavior is the *notification*—I do know that dependent objects have to be told that the cell's value has changed.

I can now use principle ① *program to an interface* to define an interface that encapsulates the processing responsibility. Traditionally, this interface is called **Observer** and contains a single method responsible for the processing named `update`.

Listing: chapter/observer/Observer.java

```
/** Observer role in the Observer pattern
 */

public interface Observer {
    /** Perform processing appropriate for the changed state.
     * Subject invokes this method every time its state changes.
     */
    public void update();
}
```

Thus, e.g. the pie chart window must implement **Observer** and put its redrawing behavior in the `update` method.

Finally, I ② *favor object composition* by letting the subject maintain a set of observers, and every time the subject changes state, it is responsible for invoking the `update` method of all observers, as shown in Figure 28.1. Traditionally, this is handled by a method called `notify`. The subject must of course also have methods to allow *registering* observers, that is, adding observers to and removing observers from the set (not shown in the figure).

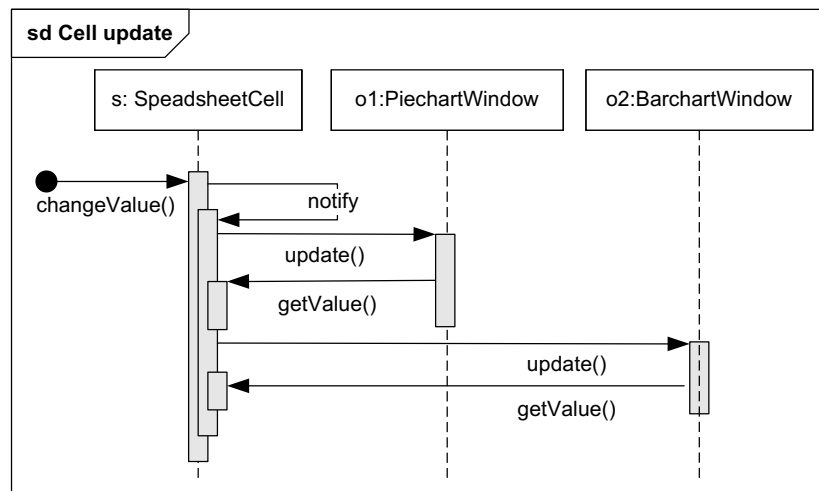


Figure 28.1: Spreadsheet updating based on cell change.

28.3 Example

As a bare bone example, consider the following code fragment.

Fragment: chapter/observer/DemoObserver.java

```

public class DemoObserver {
    public static void main(String[] args) {
        SpreadsheetCell a1 = new SpreadsheetCell();
        Observer
            observer1 = new PieChartWindow(a1),
            observer2 = new BarChartWindow(a1);
        a1.addObserver(observer1);
        a1.addObserver(observer2);

        a1.changeValue(32);
        a1.changeValue(42);

        a1.removeObserver(observer1);
        a1.changeValue(12);
    }
}

class PieChartWindow implements Observer {
    SpreadsheetCell myCell;
    public PieChartWindow(SpreadsheetCell c){
        myCell = c;
    }
    public void update() {
        System.out.println( "Pie chart notified: value: "+
            myCell.getValue() );
    }
}


class BarChartWindow implements Observer {
    SpreadsheetCell myCell;

```

```
public BarChartWindow(SpreadsheetCell c) {
    myCell = c;
}
public void update() {
    System.out.println( "Value "+ myCell.getValue()+
                        " in Bar chart" );
}
}
```

The `SpreadsheetCell` is the subject and in the main method I create two observers (of different types) and register them at the subject using `addObserver`. Next, the value of the cell is changed a couple of times, leading to notifications to its observers. When you run the application, the output is

```
Pie chart notified: value: 32
Value 32 in Bar chart
Pie chart notified: value: 42
Value 42 in Bar chart
Value 12 in Bar chart
```

 Review the other interfaces and classes for the example in folder `chapter/observer`.

28.4 The Observer Pattern

This is the OBSERVER pattern (design pattern box 28.1, page 67). Its intent is

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

The OBSERVER pattern is a pattern in which the protocol aspect is very important. Remember the definition of protocol in Chapter 15 as *a convention detailing the sequence of interactions expected by a set of roles*. In order for an observer to get notified, it must first register itself in the subject so the subject knows who to notify. This is the *registration protocol*, the first frame in the sequence diagram shown in Figure 28.2.

The loop frame represents the second protocol, the *notification protocol*, that is, the typical interaction that takes place while the application is running. Here the subject's state is changed in some way (shown by the `setState()` call) and the subject then notifies all registered observers by invoking their `update()` method in turn. To rephrase the loop frame in "protocol talk", the observer protocol dictates that any state change in the subject role must result in an `update` invocation in all registered observers.

There are some comments on this protocol. First, the sequence diagram shows that it is an observer that force the subject's state change. This is a typical situation when the observer pattern is used to handle multiple graphical views (as in the MODEL-VIEW-CONTROLLER pattern, discussed in the next chapter). But generally, the state change can come from any source, not just observers. Still, any state change starts the notification protocol. Second, any observer is free to do anything it wishes. In Figure 28.2 they both retrieve the subject's state but an observer may decide to ignore a state change.

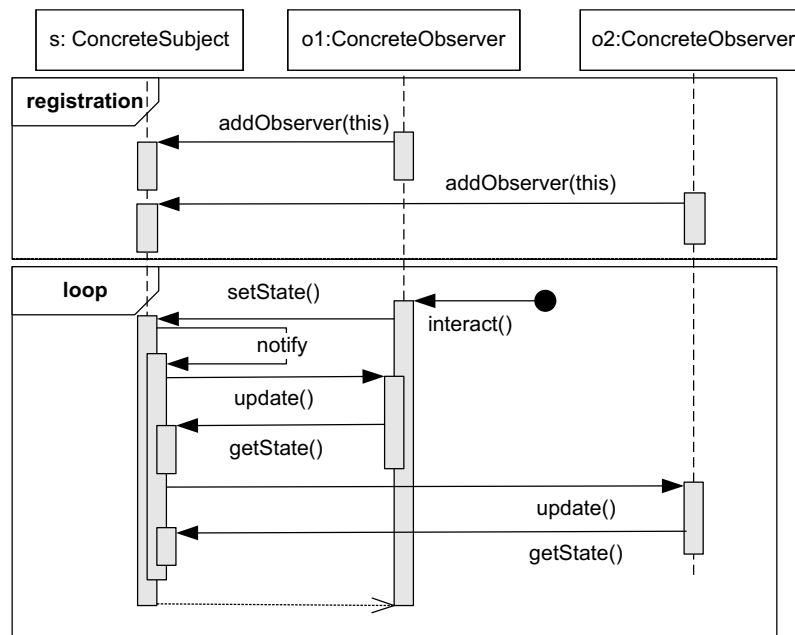


Figure 28.2: Observer pattern protocol.

An interface for the subject role typically looks like:

Listing: chapter/observer/Subject.java

```

/** Subject role in the Observer pattern
 */

public interface Subject {
    /** add an observer to the set of observers receiving notifications
     * from this subject.
     * @param newObserver the observer to add to this subject's set */
    public void addObserver(Observer newObserver);
    /** Remove the observer from the set of observers receiving
     * notifications from this subject.
     * @param observer the observer to remove from the set.*/
    public void removeObserver(Observer observer);
    /** notify all observers in case this subject has changed state. */
    public void notifyObservers();
}
  
```

However, it is not in general possible to make a single interface that describes all types of subjects. The reason is that the subject must have methods to manipulate its state and these are specific for the particular domain, for instance, the methods for changing state in a spread sheet cell are different from those that change state in a pay station. However, we can list the responsibilities that the two roles must have.

Subject

- Must handle storage, access, and manipulation of state
- Must maintain a set of observers and allow adding and removing observers to this set
- Must notify every observer in the set of any state change by invoking each observer's `update` method

Observer

- Must register itself in the subject
- Must react and process subject state changes every time a notification arrives from the subject, that is, the `update` method is invoked

The discussion above is the classic description of OBSERVER as it appeared in the book by Gamma et al. (1995), but the pattern comes in variants. The description above is the **pull variant**. In the pull variant the `update` method call is simply a notification of a state change, but no information is provided on the nature of the state change: the observer must itself retrieve the state from the subject by calling `get`-methods.

An alternative is the **push variant** protocol. Here there may be several update methods, and the update methods include parameters. The parameters and set of update methods provide detailed information about the actual state change in the subject. This variant (partly) eliminates the need for `getState()` calls in the sequence diagram above. If the subject keeps a lot of distinct state information this variant is better as the observers do not have to waste time figuring out what actually changed.

A good example of the push variant observer protocol is the AWT and Swing graphical user interface library. For instance, the `MouseListener` interface is the **Observer** role while a graphical canvas/panel is the **Subject**. The listener interface declares several update methods, for instance `mousePressed(MouseEvent e)` and `mouseReleased(MouseEvent e)`. Therefore an observer immediately knows if the mouse was pressed or released, and furthermore the `MouseEvent` object holds the X and Y coordinates of the mouse position.

Basically, observer defines an **event handling** system: the subject *emits or fires events when its state changes* to all its registered observers. This is often used in the terms used when discussing and implementing observers: when an update method is called, we talk about “emitting events”; the methods that loop over all observers and call the update methods are often called `fireXevent()`, `notifyXevent()` or the like; the observer update methods are often called `onXEvent()` or similar.

In learning iteration 7, you will see several applications of the OBSERVER pattern in the MiniDraw framework.

28.5 Analysis

The OBSERVER pattern's intent states that it defines a one-to-many relation: one subject may have many observers. Actually, it is a many-to-many relation, as a single observer can of course register itself at many different subjects at the same time.

Exercise 28.1: In Section 28.1 I gave a few examples of systems that may be implemented using the OBSERVER pattern. Analyze them and explain those where a single observer may register itself at several subjects.

The observer pattern's main benefit is the *loose coupling between subject and observer role*. All the subject needs to know is the observer interface: it is then able to notify any type of observer now and in the future. Furthermore, the pattern defines a *broadcast communication protocol* as a subject can handle any number of observers.

A liability is the *inability to distinguish types of state changes*. This is most easily demonstrated by an example: consider a subject that holds a coordinate (x,y) as two integer instance variables each with a set method: `setX` and `setY`. As both methods change the subject's state, both will trigger a full notification protocol. This means that a change to both coordinates at the same time, which conceptually is a single state change, will nevertheless generate *two* update invocations in all observers. In a graphical application, for instance, this may introduce "screen flicker" where the graphics are redrawn multiple times and therefore flickering. One possible solution is to add methods to the subject role that are similar to a database commit, like for instance `beginUpdate` and `endUpdate`. The first method simply disables the notification protocol, while the latter enforces a notification. This way several state changing calls can be made to the subject before the notification is made. While it solves the problem at the technical level, it suffers that programmers have to remember to encapsulate state changes in the transaction methods. If they are not paired correctly it leads to defects that can be difficult to identify.

A final thing to consider is cyclic dependencies: A observes on B that observes on C that observes A (note that A serves both the **Observer** and **Subject** roles here). Any state change will then lead to an infinite loop of update calls. Standard circular dependence detection code must then be added to break the loop. For instance the notify method in A may include an internal flag:

```
private boolean inNotify = false;
public void notifyObservers() {
    if (inNotify) { return; }
    inNotify = true;
    [notify all observers]
    inNotify = false;
}
```

The use of OBSERVER is pervasive in modern software and there are other terms for the roles. Subject is often called *observable* while observer is called *listener* in the Java Swing libraries. The Java library `java.util` contains an `Observer` interface (push variant) and a default `Observable` class that may come in handy.

If you look at the **Subject** role it actually has two rather distinct responsibilities: handling state changes and handling observer notifications. The former is domain specific while the latter is similar for all subjects. It therefore sometimes makes sense to encapsulate the latter code in a separate object and then let the subject delegate this handling. I will show how this can be done in the discussion of MiniDraw.

28.6 Selected Solutions

Discussion of Exercise 28.1:

As an example of an observer that registers itself at multiple subjects, consider a UML diagram editor. The association line object (observer) will register itself in both class box objects (subjects) and receive events whenever either of them are moved so it can redraw itself correctly to keep the class boxes connected.

Also, the parking lot monitor that displays the number of free parking spaces needs to monitor several pay stations that each have the subject role.

28.7 Review Questions

What is the intent of the OBSERVER pattern? Describe some typical problems in systems that it addresses.

Describe the OBSERVER pattern solution: What roles are involved, what are their responsibilities, what is the protocol?

Describe how the OBSERVER pattern solution can be viewed as an example of compositional design.

28.8 Further Exercises

Exercise 28.2:

Design a pay station monitoring system. It should be responsible for monitoring a set of pay stations on a parking lot and keep track of each buy transaction in each pay station. Based on the amount of parking time bought in each transaction and knowledge of the total number of parking spaces it should be able to calculate the number of free parking spaces. Take your starting point in the design for the pay station, and focus your work on the event notification mechanism.

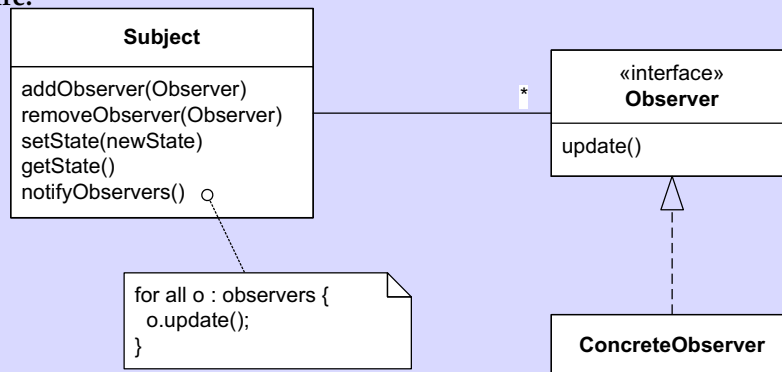
Exercise 28.3:

Develop the production and test code based on exercise 28.2. You may disregard the algorithm to calculate free parking space and replace it with *Fake It* code.

[28.1] Design Pattern: Observer

- Intent** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Problem** A set of objects needs to be notified in case a common object changes state to ensure system wide consensus and consistency. You want to ensure this consistency in a loosely coupled way.
- Solution** All objects that must be notified (**Observers**) implements an interface containing an `update` method. The common object (**Subject**) maintains a list of all observers and when it changes state, it invokes the `update` method on each object in the list. Thereby all observing objects are notified of state changes.

Structure:



- Roles** **Observer** specifies the responsibility and interface for being able to be notified. **Subject** is responsible for holding state information, for maintaining a list of all observers, and for invoking the `update` method on all observers in its list. **ConcreteObserver** defines concrete behavior for how to react when the subject experiences a state change.
- Cost - Benefit** The benefits are: *Loose coupling between Subject and Observer* thus it is easy to add new observers to the system. *Support broadcast communication* as it is basically publishing information in a one-to-many relation. The liabilities are: *Unexpected/multiple updates*: as the coupling is low it is difficult for observers to infer the nature of subject state changes which may lead to the observers updating too often or spuriously.