Observer pattern

From Wikipedia, the free encyclopedia

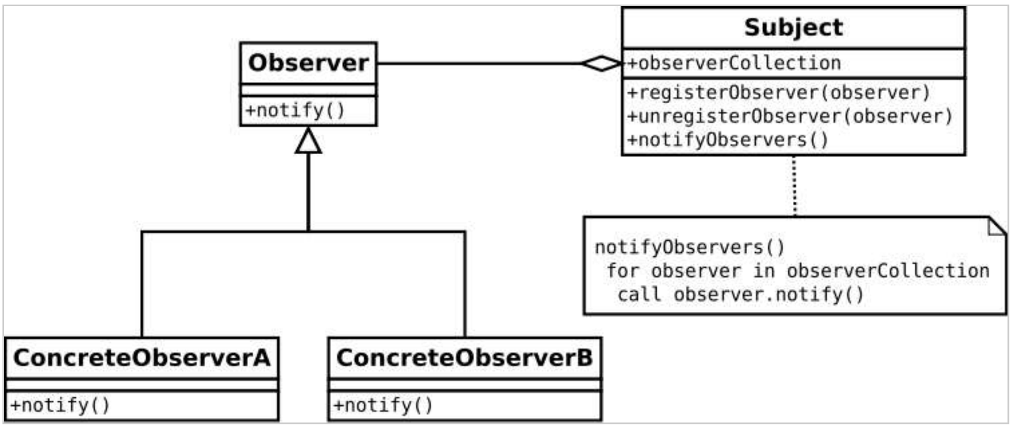
The **observer pattern** is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern.[1] The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.

The observer pattern can cause memory leaks, known as the lapsed listener problem, because in basic implementation it requires both explicit registration and explicit deregistration, as in the dispose pattern, because the subject holds strong references to the observers, keeping them alive. This can be prevented by the subject holding weak references to the observers.

Related patterns: Publish–subscribe pattern, mediator, singleton.

Contents

Structure



UML class diagram of Observer pattern

Example

Below is an example written in Java that takes keyboard input and treats each input line as an event. The example is built upon the library classes `java.util.Observer` (http://docs.oracle.com/javase/8/docs/api/java/util/Observer.html) and `java.util.Observable` (http://docs.oracle.com/javase/8/docs/api/java/util/Observable.html). When a string is supplied from System.in, the method `notifyObservers` is then called, in order to notify all observers of the event's occurrence, in the form of an invocation of their 'update' methods.

```
import java.util.Observable;
import java.util.Scanner;

class EventSource extends Observable implements Runnable {
    public void run() {
        while (true) {
            String response = new Scanner(System.in).next();
            setChanged();
            notifyObservers(response);
        }
    }
}
```

```
import java.util.Observable;
import java.util.Observer;

public class MyApp {
    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        EventSource eventSource = new EventSource();

        eventSource.addObserver(new Observer() {
            public void update(Observable obj, Object arg) {
                System.out.println("Received response: " + arg);
            }
        });

        new Thread(eventSource).start();
    }
}
```

A similar example in Python:

```
class Observable:
    def __init__(self):
        self.__observers = []

    def register_observer(self, observer):
        self.__observers.append(observer)

    def notify_observers(self, *args, **kwargs):
        for observer in self.__observers:
            observer.notify(self, *args, **kwargs)


class Observer:
    def __init__(self, observable):
        observable.register_observer(self)

    def notify(self, observable, *args, **kwargs):
        print('Got', args, kwargs, 'From', observable)


subject = Observable()
observer = Observer(subject)
subject.notify_observers('test')
```

See also

- Implicit invocation
- Client–server model

References

1. "Model-View-Controller". MSDN. Retrieved 2015-04-21.

External links

- Observer implementations in various languages at Wikibooks

Categories: Software design patterns