



[Home](#) / [Design Patterns](#) / [Behavioral patterns](#) / [Observer](#)

Observer in C++



[Back to **Observer** description](#)

Observer design pattern

1. Model the "independent" functionality with a "subject" abstraction
2. Model the "dependent" functionality with "observer" hierarchy
3. The Subject is coupled only to the Observer base class
4. Observers register themselves with the Subject
5. The Subject broadcasts events to all registered Observers
6. Observers "pull" the information they need from the Subject
7. Client configures the number and type of Observers

```
#include <iostream>
#include <vector>
using namespace std;

class Subject {
    // 1. "independent" functionality
    vector < class Observer * > views; // 3. Coupled only to "interface"
    int value;
public:
    void attach(Observer *obs) {
        views.push_back(obs);
    }
    void setVal(int val) {
        value = val;
        notify();
    }
    int getVal() {
        return value;
    }
    void notify();
};

class Observer {
    // 2. "dependent" functionality
    Subject *model;
    int denom;
public:
    Observer(Subject *mod, int div) {
        model = mod;
        denom = div;
        // 4. Observers register themselves with the Subject
        model->attach(this);
    }
    virtual void update() = 0;
protected:
    Subject *getSubject() {
        return model;
    }
    int getDivisor() {
        return denom;
    }
};

void Subject::notify() {
    // 5. Publisher broadcasts
    for (int i = 0; i < views.size(); i++)
        views[i]->update();
}
```

```
class DivObserver: public Observer {
public:
    DivObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        // 6. "Pull" information of interest
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " div " << d << " is " << v / d << '\n';
    }
};

class ModObserver: public Observer {
public:
    ModObserver(Subject *mod, int div): Observer(mod, div){}
    void update() {
        int v = getSubject()->getVal(), d = getDivisor();
        cout << v << " mod " << d << " is " << v % d << '\n';
    }
};

int main() {
    Subject subj;
    DivObserver divObs1(&subj, 4); // 7. Client configures the number and
    DivObserver divObs2(&subj, 3); // type of Observers
    ModObserver modObs3(&subj, 3);
    subj.setVal(14);
}
```

Output

```
14 div 4 is 3
14 div 3 is 4
14 mod 3 is 2
```

Read next

This article is taken from our book **Design Patterns Explained Simply**.

All of the design patterns are compiled there. The book is written in clear, simple language that makes it easy to read and understand (just like this article).

We distribute it in PDF & EPUB formats so you can get it onto your iPad, Kindle, or other portable device immediately after your purchase.

♥ Learn more



Code examples

Java	Observer in Java	Observer in Java	
C++	Observer in C++: Before and after	Observer in C++: Class inheritance vs type inheritance	Observer in C++
PHP	Observer in PHP		
Delphi	Observer in Delphi		

Design Patterns
AntiPatterns
Refactoring
UML

My account
Forum
Contact us
About us

© 2007-2018 SourceMaking.com
All rights reserved.

[Terms / Privacy policy](#)