

Home › Design Patterns › Behavioral patterns › Observer

# Observer in C++



[Read full article](#)



[See the code](#)



Search



contents

## Design Patterns

- § [Creational patterns](#)
- § [Structural patterns](#)
- § [Behavioral patterns](#)
  - [Behavioral patterns](#)
  - [Chain of Responsibility](#)
  - [Command Design Pattern](#)
  - [Interpreter Design Pattern](#)
  - [Iterator Design Pattern](#)
  - [Mediator Design Pattern](#)
  - [Memento Design Pattern](#)

- [Null Object Design Pattern](#)
- [Observer Design Pattern](#)
- [State Design Pattern](#)
- [Strategy Design Pattern](#)
- [Template Method Design Pattern](#)
- [Visitor Design Pattern](#)

## Observer design pattern

01

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

02

```
#include <iostream>
#include <vector>
```

C++

```

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);

```

```
}
```

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

*output*

## List of Observer examples

### C# examples

- [Observer in C#](#)

### C++ examples

- [Observer in C++ <=\[You are here\]](#)
- [Observer in C++: Class inheritance vs type inheritance](#)
- [Observer in C++: Before and after](#)

### Delphi examples

- [Observer in Delphi](#)

### Java examples

- [Observer in Java](#)
- [Observer in Java](#)

### PHP examples

- [Observer in PHP](#)

◀ Null Object Design  
Pattern

↑ Observer

State Design Pattern ▶



This work is licensed under a [Creative Commons Attribution-NonCommercial-No  
Derivative Works 3.0 Unported License](#)