

# The Observer Pattern

## Purpose

- Defines 1-to-n dependencies between objects so that changes to one object cause all dependent objects to be notified.

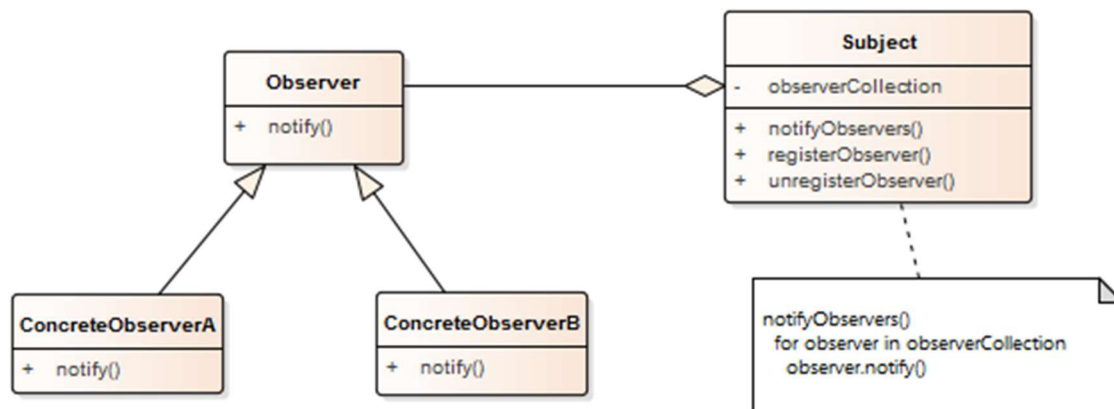
## Also known as

- Publisher-Subscriber (short Pub/Sub)

## Use Case

- One abstraction depends on the state of another abstraction
- A change to one object implies a change to another object
- Objects should be notified of state changes of another object without being tightly coupled

## Structure



### Subject

- Manages its collection of observers
- Allows the observers to register and unregister themselves

### Observer

- Defines an interface to notify the observers

### ConcreteObserver

- Implements the interface
- Is notified by the `Subject`

## Example

The following program `observer.cpp` directly implements the previous [class diagram](#).

```

// observer.cpp

#include <iostream>
#include <list>
#include <string>

class Observer {
public:
    virtual ~Observer(){};
    virtual void notify() const = 0;
};

class Subject {
public:
    void registerObserver(Observer* observer) {
        observers.push_back(observer);
    }
    void unregisterObserver(Observer* observer) {
        observers.remove(observer);
    }
    void notifyObservers() const { // (2)
        for (auto observer: observers) observer->notify();
    }

private:
    std::list<Observer*> observers;
};

class ConcreteObserverA : public Observer {
public:
    ConcreteObserverA(Subject& subject) : subject_(subject) {
        subject_.registerObserver(this);
    }
    void notify() const override {
        std::cout << "ConcreteObserverA::notify\n";
    }
private:
    Subject& subject_; // (3)
};

class ConcreteObserverB : public Observer {
public:
    ConcreteObserverB(Subject& subject) : subject_(subject) {
        subject_.registerObserver(this);
    }
    void notify() const override {
        std::cout << "ConcreteObserverB::notify\n";
    }
};

```

```

    }
private:
    Subject& subject_; // (4)
};

int main() {

    std::cout << '\n';

    Subject subject;
    ConcreteObserverA observerA(subject);
    ConcreteObserverB observerB(subject);

    subject.notifyObservers();
    std::cout << "    subject.unregisterObserver(observerA)\n";
    subject.unregisterObserver(&observerA); // (1)
    subject.notifyObservers();

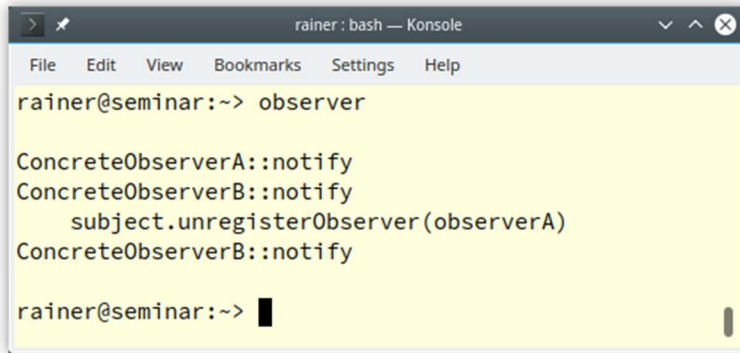
    std::cout << '\n';

}

```

The `Observer` supports the member function `notify`, and the `Subject` supports the member functions `registerObserver`, `unregisterObserver`, and `notifyObservers`. The concrete observers receive the subject in their constructor and use them to register themselves for the notification. They have a reference to the subject (lines 3 and 4). Only `observerA` is unregistered in line (1). The member function `notifyObservers` goes through all registered observers and notifies them (line 2).

The following screenshot shows the output of the program:



```
rainer: bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> observer

ConcreteObserverA::notify
ConcreteObserverB::notify
    subject.unregisterObserver(observerA)
ConcreteObserverB::notify

rainer@seminar:~> █
```

By the way, you may have noticed that I used no memory allocation in the previous program `observer.cpp`. This is how virtuality is typically used if you aren't allowed to allocate memory, such as in deeply embedded systems. Here is the corresponding main function using memory allocation:

```
int main() {

    std::cout << '\n';

    Subject* subject = new Subject;
    Observer* observerA = new ConcreteObserverA(*subject);
    Observer* observerB = new ConcreteObserverB(*subject);

    subject->notifyObservers();
    std::cout << "    subject->unregisterObserver(observerA) " << "\n";
    subject->unregisterObserver(observerA);
    subject->notifyObservers();

    delete observerA;
    delete observerB;

    delete subject;

    std::cout << '\n';

}
```

## Know Uses

The Observer Pattern is often used in architectural patterns such as [Model-View-Controller](#) (MVC) for graphical user interfaces or [Reactor](#) for event handling.

- **Model-View-Controller:** The model represents the data and its logic. The model notifies its dependent component, such as the views. The views are responsible for representing the data, and the controller is for the user input.
- **Reactor:** The Reactor registers the event handlers. The synchronous event demultiplexer (`select`) notifies the handles if an event occurs.

I will dedicate an entire future post to both architectural patterns.

## Variations

The `Subject` in the program `observer.cpp` simply sends a notification. However, more advanced workflows are often implemented:

The `Subject` sends a

- value.
- notification that a value is available. Afterward, the Observer has to pick it up.
- notification, including an indication of which value is available. The Observer picks it up if necessary.

## Related Patterns

- The [Mediator Pattern](#) establishes communication between the sender and the receiver. Each communication between the two endpoints goes, therefore, through the mediator. The Mediator and the Observer are pretty similar. The goal of the mediator is to decouple the sender and the receiver. On the contrary, the Observer established a one-way communication between the publisher and the subscriber.

## Pros and Cons

### *Pros*

- New observers (subscribers) can easily be added to the publisher
- Observers can register and unregister themselves at run time

### *Cons*

- Neither does the publisher provide a guarantee in which order the subscribers are notified, nor does it give the assertion of how long the notification takes when you have many subscribers.
- The publisher may send a notification, but a subscriber is not alive anymore. To avoid this drawback, you can implement the destructor of the concrete observers in such a way that the concrete observers unregister themselves in its destructor:

```
class ConcreteObserverA : public Observer {
```

```

public:
    ConcreteObserverA(Subject& subject) : subject_(subject) {
        subject_.registerObserver(this);
    }
    ~ConcreteObserverA() noexcept {
        subject_.unregisterObserver(this);
    }
    void notify() const override {
        std::cout << "ConcreteObserverA::notify\n";
    }
private:
    Subject& subject_;
};

```

The concrete observer `ConcreteObserverA` models the RAII Idiom: It registers itself in its constructor and unregisters itself in its destructor.