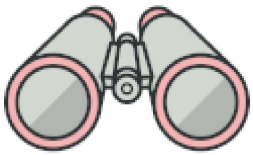

Observer in C++

Observer is a behavioral design pattern that allows some objects to notify other objects about changes in their state.

The Observer pattern provides a way to subscribe and unsubscribe to and from these events for any object that implements a subscriber interface.

Learn more about Observer

Complexity:

Popularity:

Usage examples: The Observer pattern is pretty common in C++ code, especially in the GUI components. It provides a way to react to events happening in other objects without coupling to their classes.

Identification: The pattern can be recognized by subscription methods, that store objects in a list and by calls to the update method issued to objects in that list.

Conceptual Example

This example illustrates the structure of the **Observer** design pattern. It focuses on answering these questions:

- What classes does it consist of?
- What roles do these classes play?
- In what way the elements of the pattern are related?

main.cc: Conceptual example

```
/**
 * Observer Design Pattern
 *
 * Intent: Lets you define a subscription mechanism to notify multiple objects
 * about any events that happen to the object they're observing.
```

```
 *
 * Note that there's a lot of different terms with similar meaning associated
 * with this pattern. Just remember that the Subject is also called the
 * Publisher and the Observer is often called the Subscriber and vice versa.
 * Also the verbs "observe", "listen" or "track" usually mean the same thing.
 */

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

class IObserver {
 public:
  virtual ~IObserver(){};
  virtual void Update(const std::string &message_from_subject) = 0;
};

class ISubject {
 public:
  virtual ~ISubject(){};
  virtual void Attach(IObserver *observer) = 0;
  virtual void Detach(IObserver *observer) = 0;
  virtual void Notify() = 0;
};

/**
 * The Subject owns some important state and notifies observers when the state
 * changes.
 */

class Subject : public ISubject {
 public:
  virtual ~Subject() {
    std::cout << "Goodbye, I was the Subject.\n";
  }

  /**
   * The subscription management methods.
   */
  void Attach(IObserver *observer) override {
    list_observer_.push_back(observer);
  }
  void Detach(IObserver *observer) override {
    list_observer_.remove(observer);
  }
  void Notify() override {
    std::list<IObserver *>::iterator iterator = list_observer_.begin();
    HowManyObserver();
    while (iterator != list_observer_.end()) {
      (*iterator)->Update(message_);
      ++iterator;
    }
```

```
  }

  void CreateMessage(std::string message = "Empty") {
    this->message_ = message;
    Notify();
  }
  void HowManyObserver() {
    std::cout << "There are " << list_observer_.size() << " observers in the list.\n";
  }

  /**
   * Usually, the subscription logic is only a fraction of what a Subject can
   * really do. Subjects commonly hold some important business logic, that
   * triggers a notification method whenever something important is about to
   * happen (or after it).
   */
  void SomeBusinessLogic() {
    this->message_ = "change message message";
    Notify();
    std::cout << "I'm about to do some thing important\n";
  }

 private:
  std::list<IObserver *> list_observer_;
  std::string message_;
};

class Observer : public IObserver {
 public:
  Observer(Subject &subject) : subject_(subject) {
    this->subject_.Attach(this);
    std::cout << "Hi, I'm the Observer \"" << ++Observer::static_number_ << "\".\n";
    this->number_ = Observer::static_number_;
  }
  virtual ~Observer() {
    std::cout << "Goodbye, I was the Observer \"" << this->number_ << "\".\n";
  }

  void Update(const std::string &message_from_subject) override {
    message_from_subject_ = message_from_subject;
    PrintInfo();
  }
  void RemoveMeFromTheList() {
    subject_.Detach(this);
    std::cout << "Observer \"" << number_ << "\" removed from the list.\n";
  }
  void PrintInfo() {
    std::cout << "Observer \"" << this->number_ << "\": a new message is available --> " << t
  }

 private:
  std::string message_from_subject_;
```

```
  Subject &subject_;
  static int static_number_;
  int number_;
};

int Observer::static_number_ = 0;

void ClientCode() {
  Subject *subject = new Subject;
  Observer *observer1 = new Observer(*subject);
  Observer *observer2 = new Observer(*subject);
  Observer *observer3 = new Observer(*subject);
  Observer *observer4;
  Observer *observer5;

  subject->CreateMessage("Hello World! :D");
  observer3->RemoveMeFromTheList();

  subject->CreateMessage("The weather is hot today! :p");
  observer4 = new Observer(*subject);

  observer2->RemoveMeFromTheList();
  observer5 = new Observer(*subject);

  subject->CreateMessage("My new car is great! ;)");
  observer5->RemoveMeFromTheList();

  observer4->RemoveMeFromTheList();
  observer1->RemoveMeFromTheList();

  delete observer5;
  delete observer4;
  delete observer3;
  delete observer2;
  delete observer1;
  delete subject;
}

int main() {
  ClientCode();
  return 0;
}
```

Output.txt: Execution result

```
Hi, I'm the Observer "1".
Hi, I'm the Observer "2".
Hi, I'm the Observer "3".
```

```
There are 3 observers in the list.
Observer "1": a new message is available --> Hello World! :D
Observer "2": a new message is available --> Hello World! :D
Observer "3": a new message is available --> Hello World! :D
Observer "3" removed from the list.
There are 2 observers in the list.
Observer "1": a new message is available --> The weather is hot today! :p
Observer "2": a new message is available --> The weather is hot today! :p
Hi, I'm the Observer "4".
Observer "2" removed from the list.
Hi, I'm the Observer "5".
There are 3 observers in the list.
Observer "1": a new message is available --> My new car is great! ;)
Observer "4": a new message is available --> My new car is great! ;)
Observer "5": a new message is available --> My new car is great! ;)
Observer "5" removed from the list.
Observer "4" removed from the list.
Observer "1" removed from the list.
Goodbye, I was the Observer "5".
Goodbye, I was the Observer "4".
Goodbye, I was the Observer "3".
Goodbye, I was the Observer "2".
Goodbye, I was the Observer "1".
Goodbye, I was the Subject.
```