

Observer pattern

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, in "event driven" software. Most modern languages such as C# have built-in "event" constructs which implement the observer pattern components.

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.

Contents

Overview

- What problems can the Observer design pattern solve?
- What solution does the Observer design pattern describe?

Strong vs. Weak reference

Coupling and typical pub-sub implementations

Structure

- UML class and sequence diagram
- UML class diagram

Example

- Java
- Python
- C#

See also

References

External links

Overview

The Observer ^[2] design pattern is one of the twenty-three well-known *"Gang of Four" design patterns* that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

What problems can the Observer design pattern solve?

The Observer pattern addresses the following problems:^[3]

- A one-to-many dependency between objects should be defined without making the objects tightly coupled.
- It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.
- It should be possible that one object can notify an open-ended number of other objects.

Defining a one-to-many dependency between objects by defining one object (subject) that updates the state of dependent objects directly is inflexible because it couples the subject to particular dependent objects. Tightly coupled objects are hard to implement, change, test, and reuse because they refer to and know about (how to update) many different objects with different interfaces.

What solution does the Observer design pattern describe?

- Define Subject and Observer objects.
- so that when a subject changes state, all registered observers are notified and updated automatically.

The sole responsibility of a subject is to maintain a list of observers and to notify them of state changes by calling their `update()` operation.

The responsibility of observers is to register (and unregister) themselves on a subject (to get notified of state changes) and to update their state (synchronize their state with subject's state) when they are notified.

This makes subject and observers loosely coupled. Subject and observers have no explicit knowledge of each other. Observers can be added and removed independently at run-time.

This notification-registration interaction is also known as publish-subscribe.

See also the UML class and sequence diagram below.

Strong vs. Weak reference

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.

Coupling and typical pub-sub implementations

Typically the observer pattern is implemented with the "subject" (which is being "observed") being part of the object, whose state change is being observed, to be communicated to the observers upon occurrence. This type of implementation is considered "tightly coupled", forcing both the observers and the subject to be aware of each other and have access to their internal parts, creating possible issues of scalability, speed, message recovery and maintenance (also called event or notification loss), the lack of flexibility in conditional dispersion and possible hindrance to desired security measures. In some (non-polling) implementations of the publish-subscribe pattern (also called the pub-sub pattern), this is solved by creating a dedicated "message queue" server and at times an extra "message handler" object, as added stages between the observer and the observed object whose state is being checked, thus "decoupling" the software components. In these cases, the message queue server is accessed by the observers with the observer pattern, "subscribing to certain messages" knowing only about the expected message (or not, in some cases), but knowing nothing about the message sender itself, and the sender may know nothing about the receivers. Other implementations of the publish-subscribe pattern, which achieve a similar effect of notification and communication to interested parties, do not use the observer pattern altogether.^{[4][5]}

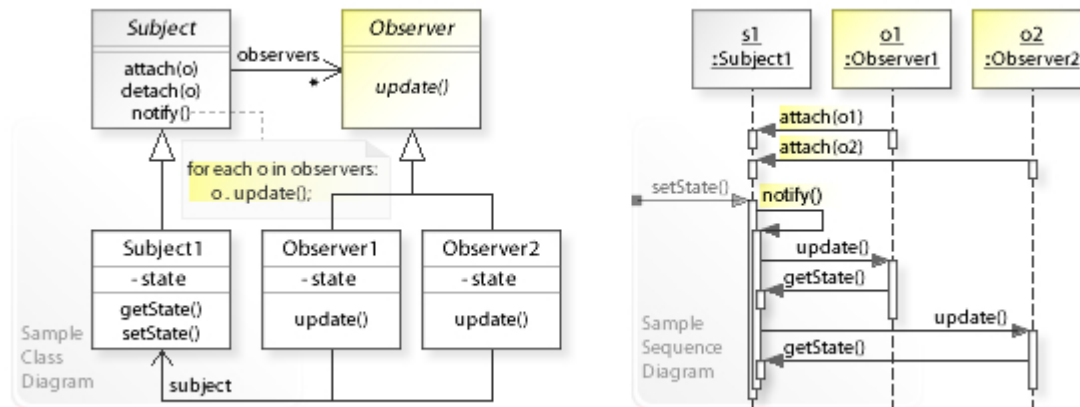
Still, in early implementations of multi-window operating systems like OS/2 and Windows, the terms "publish-subscribe pattern" and "event driven software development" were used as a synonym for the observer pattern.^[6]

The observer pattern, as described in the GoF book, is a very basic concept and does not deal with observance removal or with any conditional or complex logic handling to be done by the observed "subject" before or after notifying the observers. The pattern also does not deal with recording the "events", the asynchronous passing of the notifications or guaranteeing they are being received. These concerns are typically dealt with in message queueing systems of which the observer pattern is only a small part.

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

Structure

UML class and sequence diagram

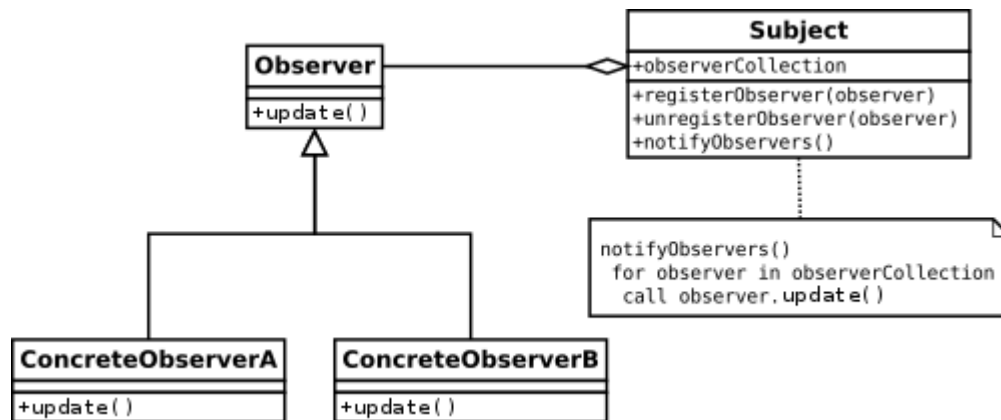


A sample UML class and sequence diagram for the Observer design pattern. [7]

In the above UML class diagram, the **Subject** class doesn't update the state of dependent objects directly. Instead, **Subject** refers to the **Observer** interface (`update()`) for updating state, which makes the **Subject** independent of how the state of dependent objects is updated. The **Observer1** and **Observer2** classes implement the **Observer** interface by synchronizing their state with subject's state.

The UML sequence diagram shows the run-time interactions: The **Observer1** and **Observer2** objects call `attach(this)` on **Subject1** to register themselves. Assuming that the state of **Subject1** changes, **Subject1** calls `notify()` on itself. `notify()` calls `update()` on the registered **Observer1** and **Observer2** objects, which request the changed data (`getState()`) from **Subject1** to update (synchronize) their state.

UML class diagram



UML class diagram of Observer pattern

Example

While the library classes `java.util.Observer` (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observer.html>) and `java.util.Observable` (<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Observable.html>) exist, they have been deprecated in Java 9 because the model implemented was quite limited.

Below is an example written in [Java](#) that takes keyboard input and treats each input line as an event. 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.

Java

```
import java.util.ArrayList;
import java.util.Scanner;

class EventSource {
    public interface Observer {
        void update(String event);
    }

    private final ArrayList<Observer> observers = new ArrayList<>();

    private void notifyObservers(String event) {
        observers.forEach(observer -> observer.update(event));
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void scanSystemIn() {
        var scanner = new Scanner(System.in);
        while (scanner.hasNextLine()) {
            var line = scanner.nextLine();
            notifyObservers(line);
        }
    }
}
```

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

        eventSource.addObserver(event -> {
            System.out.println("Received response: " + event);
        });

        eventSource.scanSystemIn();
    }
}
```

```
}  
}
```

Python

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')
```

C#

```
public class PayLoad  
{  
    public string Message { get; set; }  
}
```

```
public class Subject : IObservable<PayLoad>  
{  
    public IList<IObserver<PayLoad>> Observers { get; set; }  
  
    public Subject()  
    {  
        Observers = new List<IObserver<PayLoad>>();  
    }  
  
    public IDisposable Subscribe(IObserver<PayLoad> observer)
```

```
{
    if (!Observers.Contains(observer))
    {
        Observers.Add(observer);
    }
    return new Unsubscriber(Observers, observer);
}

public void SendMessage(string message)
{
    foreach (var observer in Observers)
    {
        observer.OnNext(new Payload { Message = message });
    }
}
}
```

```
public class Unsubscriber : IDisposable
{
    private IObservable<Payload> observer;
    private IList<IObservable<Payload>> observers;
    public Unsubscriber(IList<IObservable<Payload>> observers, IObservable<Payload> observer)
    {
        this.observers = observers;
        this.observer = observer;
    }

    public void Dispose()
    {
        if (observer != null && observers.Contains(observer))
        {
            observers.Remove(observer);
        }
    }
}
```

```
public class Observer : IObservable<Payload>
{
    public string Message { get; set; }

    public void OnCompleted()
    {
    }

    public void OnError(Exception error)
    {
    }

    public void OnNext(Payload value)
    {
    }
}
```

```
        Message = value.Message;
    }

    public IDisposable Register(Subject subject)
    {
        return subject.Subscribe(this);
    }
}
```


See also

- Implicit invocation
- Client–server model
- The observer pattern is often used in the entity–component–system pattern

References

- "Model-View-Controller" (<https://msdn.microsoft.com/en-us/library/ff649643.aspx>). MSDN. Retrieved 2015-04-21.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. pp. 293ff. ISBN 0-201-63361-2.
- "The Observer design pattern - Problem, Solution, and Applicability" (<http://w3sdesign.com/?gr=b07&ugr=proble>). *w3sDesign.com*. Retrieved 2017-08-12.
- Comparison between different observer pattern implementations (<https://github.com/millermedeiros/js-signals/wiki/Comparison-between-different-Observer-Pattern-implementations>) Moshe Bindler, 2015 (Github)
- Differences between pub/sub and observer pattern (<https://www.safaribooksonline.com/library/view/learning-javascript-design/9781449334840/ch09s05.html>) The Observer Pattern by Adi Osmani (Safari books online)
- The Windows Programming Experience (<https://books.google.co.il/books?id=18wFKrKdDm0C&pg=PA230&lpg=PA230>) Charles Petzold, Nov 10, 1992, PC Magazine (Google Books)
- "The Observer design pattern - Structure and Collaboration" (<http://w3sdesign.com/?gr=b07&ugr=struct>). *w3sDesign.com*. Retrieved 2017-08-12.

External links

-  Observer implementations in various languages at Wikibooks

Retrieved from "https://en.wikipedia.org/w/index.php?title=Observer_pattern&oldid=892872560"

This page was last edited on 17 April 2019, at 12:25 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

