Observer method – Python Design Patterns

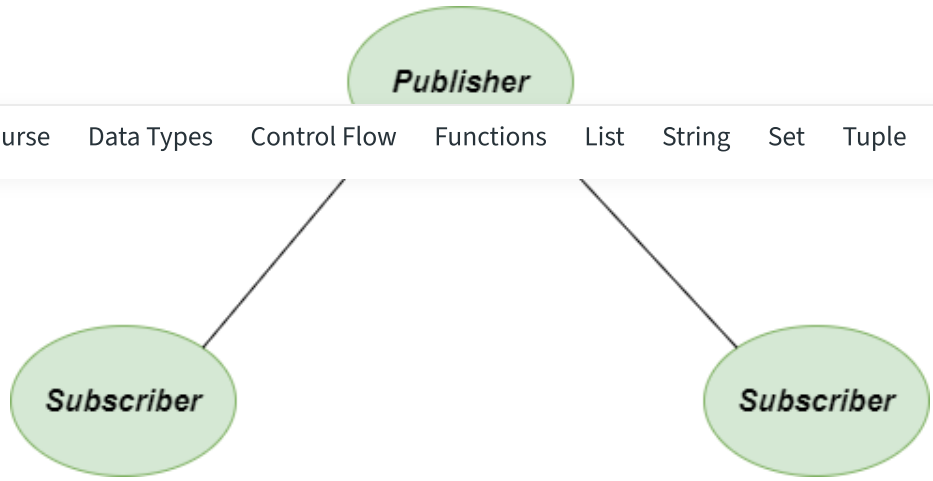
The observer method is a **Behavioral design Pattern** which allows you to define or create a subscription mechanism to send the notification to the multiple objects about any new event that happens to the object that they are observing. The subject is basically observed by multiple objects. The subject needs to be monitored and whenever there is a change in the subject, the observers are being notified about the change. This pattern defines one to Many dependencies between objects so that one object changes state, all of its dependents are notified and updated automatically.

Problem Without using Observer Method

Imagine you want to create a calculator application that has different features such as addition, subtraction, changing base of the numbers to hexadecimal, decimal, and many other features. But one of your friends is interested in changing the base of his favorite number to Octal base number and you are still developing the application. So, what could be the solution to it? Should your friend check the application daily just to get to know about the status? But don't you think it would result in a lot of unnecessary visits to the application which were definitely not required. Or you may think about that each time you add the new feature and send the notification to each user. Is it OK? Sometimes yes but not every time. Might be some users get offended by a lot of unnecessary notifications which they really don't want.

Solution using Observer Method

Let's discuss the solution to the above-described problem. Here comes the object **Subject** into the limelight. But it also notifies the other objects also that's why we generally call it **Publisher**. All the objects that want to track changes in the publisher's state are called subscribers.



Observer-method-solution-diagram

Python3

```
class Subject:

    """Represents what is being observed"""

    def __init__(self):

        """create an empty observer list"""

        self._observers = []

    def notify(self, modifier = None):

        """Alert the observers"""

        for observer in self._observers:
            if modifier != observer:
                observer.update(self)

    def attach(self, observer):

        """If the observer is not in the list,
        append it into the list"""

        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer):

        """Remove the observer from the observer list"""

        try:
            self._observers.remove(observer)
        except ValueError:
            pass
```

```
class Data(Subject):

    """monitor the object"""

    def __init__(self, name =''):
        Subject.__init__(self)
        self.name = name
        self._data = 0

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, value):
        self._data = value
        self.notify()


class HexViewer:

    """updates the Hexviewer"""

    def update(self, subject):
        print('HexViewer: Subject {} has data 0x{:x}'.format(subject.name, subj

class OctalViewer:

    """updates the Octal viewer"""

    def update(self, subject):
        print('OctalViewer: Subject' + str(subject.name) + 'has data '+str(oct(


class DecimalViewer:

    """updates the Decimal viewer"""

    def update(self, subject):
        print('DecimalViewer: Subject % s has data % d' % (subject.name, subjec

"""main function"""

if __name__ == "__main__":

    """provide the data"""

    obj1 = Data('Data 1')
    obj2 = Data('Data 2')

    view1 = DecimalViewer()
    view2 = HexViewer()
    view3 = OctalViewer()

    obj1.attach(view1)
    obj1.attach(view2)
```

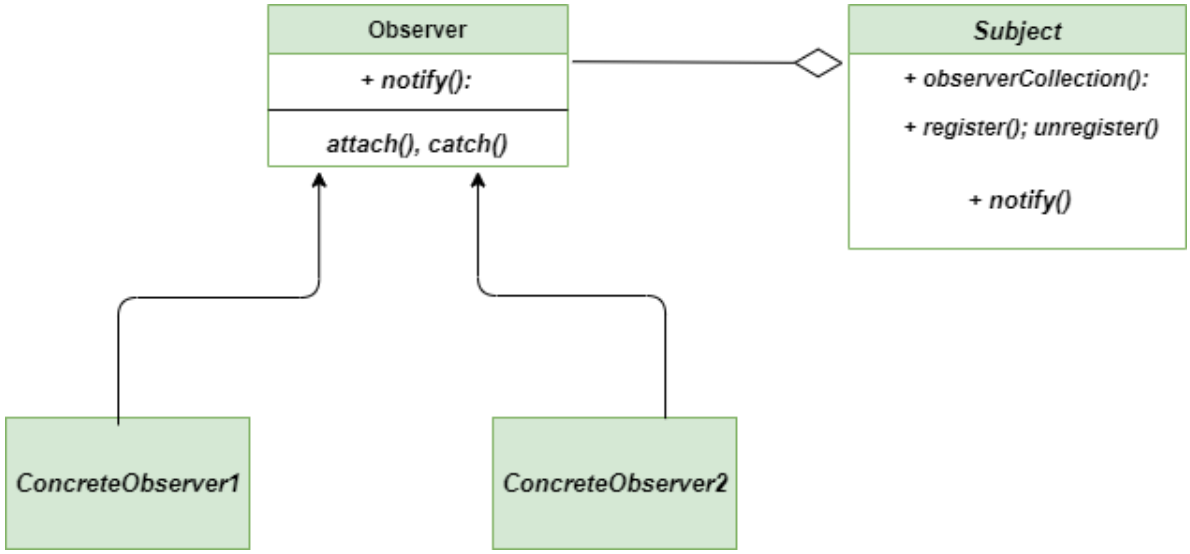
```
    obj1.attach(view3)

    obj2.attach(view1)
    obj2.attach(view2)
    obj2.attach(view3)

    obj1.data = 10
    obj2.data = 15
```

Class Diagram

Following is the class diagram for the Observer Method



Class-diagram-Observer-method

Output

```
DecimalViewer: Subject Data 1 has data 10
HexViewer: Subject Data 1 has data 0xa
OctalViewer: SubjectData 1has data 0o12
DecimalViewer: Subject Data 2 has data 15
HexViewer: Subject Data 2 has data 0xf
OctalViewer: SubjectData 2has data 0o17
```

Advantages

- **Open/Closed Principle:** Introducing subscriber classes is much easier in Observer method as compared to others without making changes in the client's code.
- **Establishes Relationships:** Its really easy to establishes the relationships at the runtime between the objects.
- **Description:** It carefully describes about the coupling present between the objects and the observer. Hence, there is no need to modify Subject to add or remove observers.

Disadvantages

- **Memory Leakage:** Memory leaks caused by **Lapsed Listener Problem** because of explicit register and unregistering of observers.
- **Random Notifications:** All the subscribers present gets notification in the random order.
- **Risky Implementations:** If the pattern is not implemented carefully, there are huge chances that you will end up with large complexity code.

Applicability

- **Multi-Dependency:** We should use this pattern when multiple objects are dependent on the state of one object as it provides a neat and well tested design for the same.
- **Getting Notifications:** It is used in social media, RSS feeds, email subscription in which you have the option to follow or subscribe and you receive latest notification.
- **Reflections of Object:** When we do not coupled the objects tightly, then the change of a state in one object must be reflected in another object.

Further Read – **Observer Method in Java**

Last Updated : 06 Oct, 2022

4

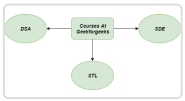
Similar Reads


Implementing Newsletter Subscription using Observer Design Pattern in Python

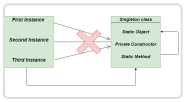

Factory Method - Python Design Patterns


Composite Method - Python Design Patterns

Decorator Method - Python Design Patterns


Abstract Factory Method - Python Design Patterns


Builder Method - Python Design Patterns


Singleton Method - Python Design Patterns


Adapter Method - Python Design Patterns


Prototype Method - Python Design Patterns


Bridge Method - Python Design Patterns

Related Tutorials

OpenAI Python API - Complete Guide


Pandas AI: The Generative AI Python Library

Python for Kids - Fun Tutorial to Learn Python Programming

Data Analysis Tutorial

Flask Tutorial

Previous Next

Article Contributed By :


chaudhary_19

chaudhary_19

Follow

Vote for difficulty

Current difficulty : Easy

Easy | Normal | Medium | Hard | Expert

Improved By : surinderdawra388, anikaseth98

Article Tags : python-design-pattern, Python

Practice Tags : python

Improve Article | Report Issue

GeeksforGeeks
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org