PATRONES DE DISEÑO

OBSERVER

Jesús Ángel Rodríguez Martínez

16/09/2020

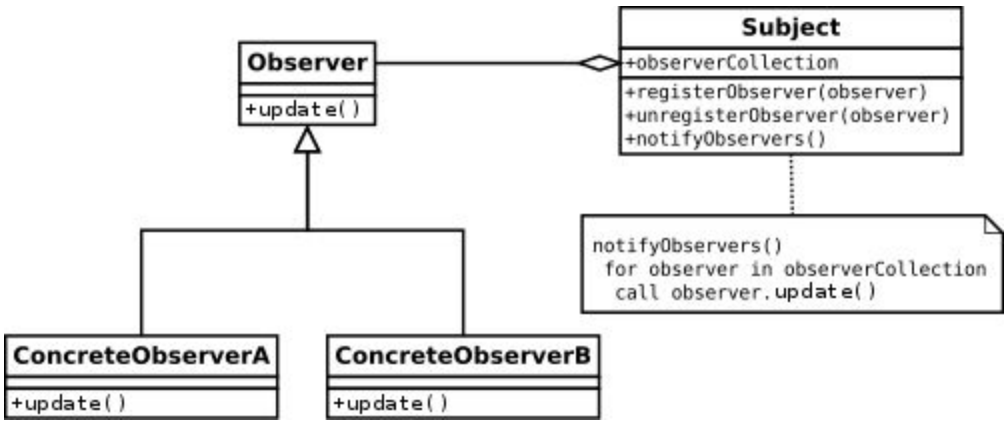
Diseño y arquitectura de software

INTRODUCCIÓN

Observer es un patrón de diseño de comportamiento que permite a un objeto notificar a otros objetos sobre cambios en su estado.

Este patrón proporciona una forma de suscribirse y cancelar la subscripción a estos eventos para cualquier objeto que implementa una interfaz suscriptora.

ESTRUCTURA



APLICACIÓN O USOS CONOCIDOS

Un juego cuenta con varios módulos como el personaje, mapa, misiones, inventario, etc, y muchas veces estos módulos aunque son en gran medida inconexos necesitan saber de los demás, entonces utilizando un gestor general se podría encargar de recopilar la información y distribuirla al resto de módulos.

Otro ejemplo sería un juego en línea donde suele haber un servidor al que se conectan todos los jugadores y cuando pasa algo el servidor envía información al resto de jugadores, así todos los jugadores saben en todo momento dónde están los otros jugadores, cuántos puntos de daño recibieron, eventos, etc.

¿QUÉ PROBLEMAS PUEDE RESOLVER ESTE PATRÓN?

1. Se debe definir una dependencia de uno a muchos entre objetos sin hacer que los objetos estén estrechamente acoplados.
2. Debe asegurarse que cuando un objeto cambia de estado, se actualiza automáticamente un número ilimitado de objetos dependientes.
3. Debería ser posible que un objeto pueda notificar un número ilimitado de otros objetos.

¿QUÉ PROBLEMAS TIENE ESTE PATRÓN?

El patrón de observador puede causar pérdidas de memoria , conocido como el **problema del oyente caducado**. La fuga ocurre cuando un observador no puede darse de baja del sujeto cuando ya no necesita escuchar. En consecuencia, el sujeto todavía mantiene una referencia al observador que evita que se recoge la basura - incluyendo todos los demás objetos que se refiere a - durante el tiempo que el sujeto está vivo, que podría ser hasta el final de la aplicación.

IMPLEMENTACIÓN EN PYTHON

```python
class Subscriber:
    def __init__(self, name):
        self.name = name
    def update(self, message):
        print('{} got message "{}"'.format(self.name, message))

class Publisher:
    def __init__(self):
        self.subscribers = set()
    def register(self, who):
        self.subscribers.add(who)
    def unregister(self, who):
        self.subscribers.discard(who)
    def dispatch(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)

pub = Publisher()
bob = Subscriber('Bob')
alice = Subscriber('Alice')
john = Subscriber('John')
pub.register(bob)
pub.register(alice)
pub.register(john)
pub.dispatch('First update from publisher')
pub.unregister(alice)
pub.dispatch('Second update from publisher')
```