Observer Pattern

Behavioral pattern

Problem Statement:

Suppose that a large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Pattern Idea:

The **observer pattern** is a [software design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) in which an [**object**](https://en.wikipedia.org/wiki/Object_(computer_science)#Objects_in_object-oriented_programming), 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](https://en.wikipedia.org/wiki/Method_(computer_science)).

It is mainly used to implement distributed [event handling](https://en.wikipedia.org/wiki/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 Used the following problems:

* 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.

**Observer Pattern’s Advantages:**

* It supports the principle of loose coupling between objects that interact with each other
* It allows sending data to other objects effectively without any change in the Subject or Observer classes
* Observers can be added/removed at any point in time

**Code Implementation:**

**import** **java.util.ArrayList**;

**import** **java.util.Scanner**;

**abstract** **class** **Observer** {

**protected** Subject subject;

**public** **abstract** **void** update();

}

**class** **Subject** {

**private** List<Observer> observers = **new** ArrayList<>();

**private** **int** state;

**public** **void** add(Observer o) {

observers.add(o);

}

**public** **int** getState() {

**return** state;

}

**public** **void** setState(**int** value) {

**this**.state = value;

execute();

}

**private** **void** execute() {

**for** (Observer observer : observers) {

observer.update();

}

}

}

**class** **HexObserver** **extends** Observer {

**public** HexObserver(Subject subject) {

**this**.subject = subject;

**this**.subject.add(**this**);

}

**public** **void** update() {

System.out.print(" " + **Integer**.toHexString(subject.getState()));

}

}

**class** **OctObserver** **extends** Observer {

**public** OctObserver(Subject subject) {

**this**.subject = subject;

**this**.subject.add( **this** );

}

**public** **void** update() {

System.out.print(" " + **Integer**.toOctalString(subject.getState()));

}

}

**class** **BinObserver** **extends** Observer {

**public** BinObserver(Subject subject) {

**this**.subject = subject;

**this**.subject.add(**this**);

}

**public** **void** update() {

System.out.print(" " + **Integer**.toBinaryString(subject.getState()));

}

}

**public** **class** **ObserverDemo** {

**public** **static** **void** main( **String**[] args ) {

Subject sub = **new** Subject();

*// Client configures the number and type of Observers*

**new** HexObserver(sub);

**new** OctObserver(sub);

**new** BinObserver(sub);

Scanner scan = **new** Scanner(System.in);

**for** (**int** i = 0; i < 5; i++) {

System.out.print("\nEnter a number: ");

sub.setState(scan.nextInt());

}

}

}

**Drawbacks of Observer pattern:**

* The Observer interface has to be implemented by ConcreteObserver, which involves inheritance. There is no option for composition, as the Observer interface can be instantiated.
* If not correctly implemented, the Observer can add complexity and lead to inadvertent performance issues.
* Because observers don’t know about each other, a simple update to an observer might cause a long chain of other updates.
* Each observer decides whether it needs to update something when it receives the notifications from a subject that something has changed. Therefore, complicated observers have to do a lot of work to figure out what changed when they receive a notification