Observer Pattern

Problem Statement:

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**;

**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();

}

}

**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