**Observer Pattern**Group 07 – Dung  
Reference:  
Head First Design Patterns – Eric Freeman & Elisabeth Freeman with Kathy Sierra & Bert Bates  
GeeksforGeeks

1. Example

In social media like Facebook, YouTube, Instagram,… if a user **follows/subscribe** another user, a page, or a channel, whenever the objects he follows upload a new post, he will get notification about the post immediately or the post must appear on the feed of that user. Users who don’t follow these objects get nothing about the appearance of their new posts. However, may be on one day, he don’t want to view posts of one of those who he follows anymore, he can **unfollow** that object and stop getting notification about it.

We have to make sure that when a person or an organization uploads anything new, their followers must be notified immediately and the followers can unfollow also.

1. Definition

“The Observer Pattern defines a one-to-many dependency between objects so that when one

object changes state, all its dependents are notified and updated automatically.”

-Head First Design Patterns

* The objects which is followed is called **Subject**s, and its dependents are called **Observer**s.
* We will implement two **abstract** classes: **Subject** which the subjects’ concrete classes will inherit and **Observer** which the observers’ concrete classes will inherit.
* Their works:

|  |  |
| --- | --- |
| <*abstract*> **Subject** | <*abstract*> **Observer** |
| addObserver(Observer\* o);  removeObserver(Observer\* o);  notifyObservers(); | update(); //or getNotification(); |

|  |  |
| --- | --- |
| **ConcreteSubject**: public **Subject** | **ConcreteObserver**: public **Observer** |
| addObserver(Observer\* o);  removeObserver(Observer\* o);  notifyObservers();  doSomething() {  notifyObservers();  }  *//some specific members of the*  *//ConcreteSubject //…* | update();  *//some specific members of the //ConcreteObserver*  *//…* |

When a ConcreteObserver object (CO) wants to follow a ConcreteSubject object (CS):  
*CS.addObserver(CO);*  
For example CS class has a *vector<Observer\*> Obs* to manage its followers, so we have:  
***ConcreteSubject****::addObserver(****Observer****\* o) {  
 Obs.push\_back(o);  
}*Then we call  
CS.doSomething();  
The ConcreteSubject itself will notify their observers:  
***ConcreteSubject***::notifyObserver() {  
 for (int i = 0; i < Obs.size(); i++) {  
 Obs[i]->update();  
 }  
}  
At the end, the CO.update() will be called whenever CS.doSomething() is called.  
(*We can manage variety subjects by using vector<Subject\*> also*)

* Thanks to the polymorphism, the subjects can have different observers, different methods to add/remove observers, different things to notify observers and the observers are free to update in their own way.

1. Some other examples

* AirVisual mobile app get information about the air from several air stations, whenever the stations send new information, the program needs to update and display on the screen the new information (subject: AirStation, observer: DisplayScreen).
* In student management system, whenever the lecturer updates the scoreboard of a class, the invidual of the class’s scoreboard and studying result have to update and recalculate too (subject: ClassScoreboard, observer: StudentResult).
* Grab lets the customers see the position of the driver on the map, it means whenever the GPS of the driver’s device updates, the customer’s screen will receive that location to display on the map (subject: DriverGPS, observer: CusDisplay).

1. Advantages and disadvantages

Advantages: The subjects and their observers can interact to each other but they don’t have much knowledge about each other.

* We can add new observers whenever we want even at the runtime.
* The subject needn’t change anything to add a new type of observers.
* We can easily reuse subjects and observers independently.
* The changes to each of them doesn’t affect the other.
* The observers only get what the subjects give out, not come around and look at everything of the subject.
* The observers don’t need to “getState” everytime they deal with the state to make sure that it is the latest.
* The observers don’t need to have multiple methods to get all states that they need.

Disadvantages:

* The observers may need different states, some parts of the change aren’t needed to them but the subject can’t manage this. The subject pushes everything that everyone may needs and each of the observers have to receive them all (for example the AirStation pushes everything: temperature, humidity, weather,… but NotificationBarDisplay, an observer of it, only need the temperature).
* When the subject expands, adds more states, it has to go around and change all the update calls on every observer.
* Memory leaks caused by Lapsed listener problem because of explicit register and unregistering of observers.