Observer pattern

Somebody's watching me...

Purpose of the Observer pattern

You have an object, “Subject” whose state many other objects are interested in.

In particular, the many other objects are interested in changes in the Subject.

You can say that the Subject is observable by many observers (or listeners) which are notified whenever the Subject changes.

Examples from the Java API

The obvious example of the observer pattern from the Java API comes from Swing and the way events are handled.

Every component in Swing (and AWT) has methods for registering various listeners. The methods are using the semantics of adding a listerner, which means that multiple objects can listen to events triggered by the component.

The component is the Subject and the concrete listeners are the observers.

Let’s look at a small simple demo.

Swing Demo - Multi-lingual Panic button

The idea of this silly demo is to show that a button can have several listeners interested in button clicks. When someone clicks the panic button, all listeners are notified and an alarm message goes out in Spanish, English, Swedish and German.

The idea is to encapsulate the message to an ActionListener, so that we can add listeners later for more languages.

This would be done dynamically, and not hard-coded as in this demo.

Adding the listeners

```
theButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
      System.out.println("Hilfe!");
    }});
// Add a Swedish panic button listener
theButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
      System.out.println("Hjälp!");
    }});
// Add an English panic button listener
theButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
      System.out.println("Help!");
    }});
// Add a Spanish panic button listener
theButton.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent ae){
      System.out.println("Socorro!");
    }});
```

When someone clicks - all listeners are notified

Socorro!

Help!

Hjälp!

Hilfe!

Note that the order in which the listeners are called is unknown to us.

General design of the observer pattern

Typically, you define an interface for the Subject e.g. “Observable” and an interface for the observers e.g. “Observer”.

The subject to be observed implements Observable and all the observers implement Observer.

Observable interface

```
public interface Observable {
  public void add(Observer o);// an observer wants to register
  public void delete(Observer o);// unregister observer
  public void notifyObservers(); // notify all observers
}
```

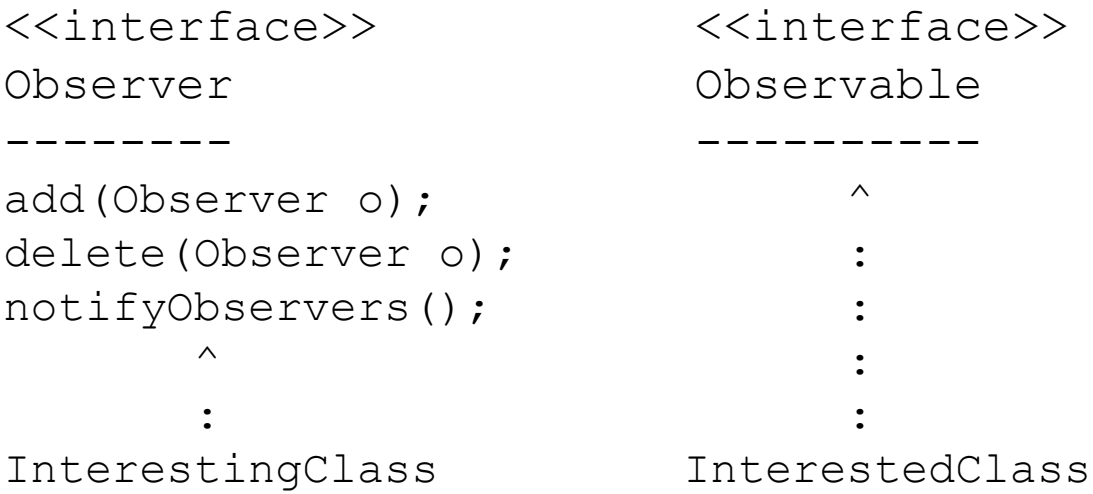
Observer interface

```
public interface Observer{
  public void update(Observable o);
}
```

Typically, an Observer registers in the constructor (which takes an Observable as an argument):

```
public class SubjectObserver implements Observer{
  public FinalScore(Observable o){
    o.add(this); // Add itself to the observable
  }
}
```

Typical Design



Typical code - The Subject (observable thing)

```
class InterestingClass implements Observable{
  // a list of observers, and other instance variables...
  public void add(Observer o){
    if(!observers.contains(o)){
      observers.add(o);
    }
  }
  public void delete(Observer o){
      observers.remove(o);
  }
  public void notifyObservers(){ // This should be called when changed
    for(Observer o : observers){
      o.update(this);
    }
  }
  public void changesHappen() { /*...*/  notifyObservers(); }
}
```

Typical code - The observer (observes the Subject)

```
class InterestedClass implements Observer{
  public InterestedClass(Observable o){ // constructor
    o.add(this);//tell the observable that we're interested
  }

  // update will be called by the observable
  // to tell us something has changed!
  public void update(Observable o){ // pull method
    //get data from o and do something with it
  }
}
```

Could we use observers for our game?

The game engine crew decides that the engine should keep track of all characters that are killed during the game. They create a FragList class for this.

The FragList object keeps track of many things:

- How many characters have each character killed?
- What characters has each character killed?
- Is it game over? (client code tells the FragList if it is)

Each time a character dies, the game engine tells this to the FragList

Urban dictionary:
frag: to kill an enemy in a single person shooter computer game.

Who would be interested in kills?

Now, the crew decides that it would be cool if a statistics message is shown after a kill:

```
====Current frag list====
Sir Playsalot: 1 frag(s)
========================
```

They decide to create a GameStats class responsible for printing this message each time a character dies.

The GameStats class should observe the FragList and everytime FragList changes, it should be notified and print this message.

Characters class - keeps track of live characters

They also decide that they should have a Characters class, which keeps track of every player in the game (bad guys, and good guys).

Quickly, they realize that the Characters object must be interested to what's going on in FragList! As soon as a character dies, FragList knows, so the characters object should remove this character.

The characters object can be used to print a list of all characters at the start of the game, and at the end.

Characters object - listing the live characters

At game start:

```
============ players in this game ===============
Bill with health: 100 carrying Ugly wooden club
Dolly with health: 100 carrying Ugly wooden club
Sir Humpty with health: 100 carrying Excalibur
Sir Playsalot with health: 100
Steve with health: 100 carrying Ugly wooden club
Trolly with health: 100 carrying Ugly wooden club
================================================
```

At game over:

```
Final score:
Sir Playsalot with health: 355 carrying Shotgun
```

At game over, some final frag stats are shown

An additional class is created to give a list of each live player’s victories:

```
====Kills in the game===
{Sir Playsalot=[Sir Humpty, Trolly, Dolly, Steve, Bill]}
========================
```

Design of the Subject and Observers


```
// Client code:
// Observable list:
FragList fragList = new FragList();
//Observers:
GameStats gameStats = new GameStats(fragList);
FinalScore finalScore = new FinalScore(fragList);
Characters characters = new Characters(fragList);
//...
characters.addCharacters(opponents);
characters.addCharacter(player);
```

Client code

```
FragList fragList = new FragList();
GameStats gameStats = new GameStats(fragList);
FinalScore finalScore = new FinalScore(fragList);
Characters characters = new Characters(fragList);
//...
characters.addCharacters(opponents);
characters.addCharacter(player);
    System.out.println("============ players in this game ==============");
    System.out.println(characters);
    System.out.println("================================================");
// ... after a fight:
if(c.health()<1){
  fragList.incFrags(player.name()); // this should be a function of...
  fragList.addKill(player, c);      // this! - You fix it :P
}
// ...
fragList.gameOver();
System.out.println("Final score: ");
System.out.println(characters);
```

Improvements needed

When a player dies, it should notify FragList itself (it shouldn’t be the responsibility of the game engine / client code) FragList can be an observer as well as an observable! It can observe characters ;-)

The FragList class needs only one public method for the clients:

```
fragList.addKill(Character killer, Character frag);
```

It doesn’t need also a method to update kill-count of a character - it can be inferred from that one.

The constructor of each concrete character could add itself to a global Characters object. (All of these decisions are up to the designer)

Can you fix this?

Maybe you can download the code and fix these flaws.

Note - as with all examples in these design patterns examples, things are simplified and not perfect.

And as with most design patterns, some treats fix some problems but the pattern might at the same time introduce new problems.

All patterns have strengths and weaknesses.

As usual, the example is rather simplified and not perfect, but we think that it will make you think about how observer can be used.