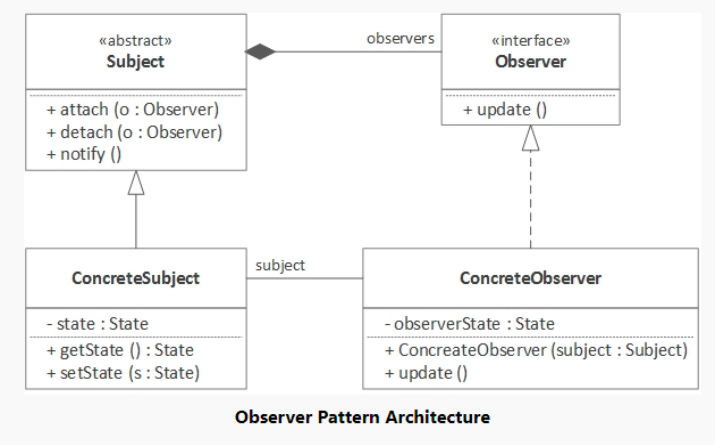
EL PATRÓN OBSERVER

- Imagínate que hay un producto y muchos clientes que están interesados en dicho producto de forma que si el producto baja de precio queremos avisar a los clientes interesados en el producto.
- Imagínate que los resultados de un torneo de tenis se quieren comunicar a una serie de suscriptores a medida que van concluyendo los partidos
- etc.

Las situaciones anteriores son distintas pero tienen en común que hay un objeto que es observado por muchos(un producto observado por muchos clientes y un torneo observado por muchos suscriptores) y que cuando cambia el estado del objeto observado, los observadores desean recibir una notificación.
El patrón observer nos da el esquema para solucionar esta situación. Un UML que describe el patrón observer es el siguiente



Observer Pattern Architecture

Encontrarás en la web descripciones del patrón con diseños UML con pequeñas variaciones respecto a este, pero la idea esencial es siempre la misma.

El sujeto(subject) representa al observado. En el UML anterior para hacer un diseño más extensible hay subject abstracto y a partir se generan sujetos concretos. A menudo en lugar de una clase abstracta como en el uml anterior encontraremos en su lugar un interface que se suele llamar "observable".
Vamos a resaltar unos cuantos puntos claves del patrón:

- a través del interface Observable(o clase abstracta subject) definimos el comportamiento que todo objeto observable debe de tener y normalmente como mínimo implica que todo objeto observable tiene una lista de observadores y:
  - tiene un método addObserver() para añadir o registrar un observador, es decir, añadirlo a la lista
  - removeObserver() para dar de baja un observador, es decir, borrarlo de la lista
  - notifyObservers() un método que recorre la lista de observadores para notificarles un cambio
- A través del interface Observer se define un método update() a menudo con parámetros. El parámetro puedo ser un Simple String o cualquier otro tipo de objeto, incluso puede ser el propio objeto observado con lo que el observador puede acceder al observado y ver su estado.

A través de un ejemplo se entienden mejor estos conceptos.

Ejercicio U7_B8B_E1:
En una tienda virtual hay una serie de productos. Un producto puede variar de precio en cualquier momento. Se permite que un objeto Producto "sea seguido" por objetos que

se suscriben a él y son notificados por dicho producto cuando sufre una bajada de precio. Concretamente hay dos tipos posibles de observadores : Clientes y Empleados de Marketing.
Cuando un producto cambia de precio este avisa a sus observadores enviándoles un objeto de la clase Aviso

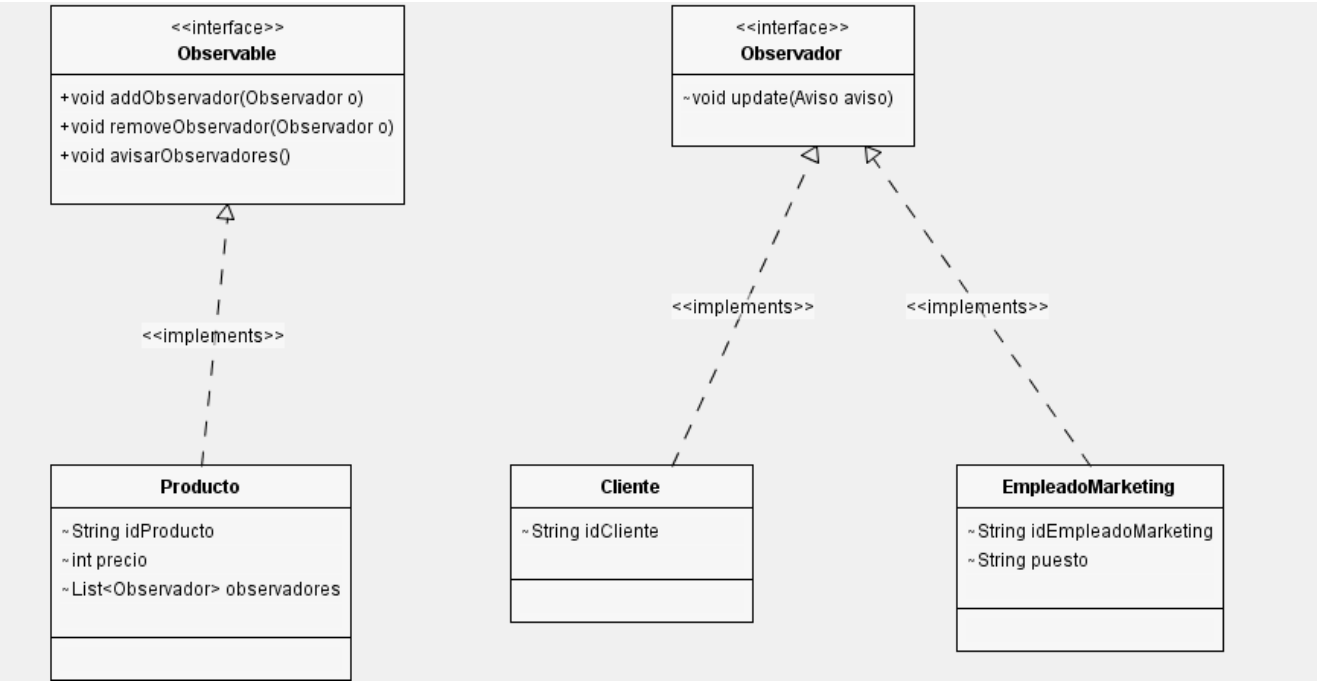
```
class Aviso{
    String idProducto;
    int precio;
   Date  fecha;

    Aviso(String idProducto, int precio, Date  fecha) {
        this.idProducto = idProducto;
        this.precio = precio;
        this.fecha = fecha;
    }

}
```

que contiene el precio actualizado y el instante en el que se produjo la actualización del precio.

Para resolver el problema, aplicamos el patrón Observer siguiendo el siguiente diagrama UML de clases simplificado que simplemente muestra los interfaces que hay que implementar y las clases que deben implementarlos . De los interfaces también muestra los métodos que hay que implementar.



En el dibujo + significa public y ~ modo acceso paquete(default) aunque no te influye en la solución ya que puedes considerar que todas las clases pertenecen al mismo paquete. No aparecen en el diagrama anterior las clases Aviso y la clase TiendaVirtual. La clase TiendaVirtual es la clase principal, que contiene el main y contiene el siguiente código:

```
class TiendaVirtual {

    public static void main(String[] args) {
        Producto p1= new Producto("p1",100);
        Cliente c1=new Cliente("Ana");
        EmpleadoMarketing e1=new EmpleadoMarketing("111","ofertas");
        p1.addObservador(c1);
```

```
    p1.addObservador(e1);
    p1.addObservador(new Cliente("Juan"));
    p1.setPrecio(90);
    System.out.println("");
    p1.removeObservador(e1);
    p1.setPrecio(80);
  }

}
```

y genera la siguiente salida:

```
Soy el cliente Ana y fui avisado  de bajada de precio en producto p1 a 90 euros
Soy el empleado 111 de  ofertas y fui avisado  de bajada de precio en producto p1 a 90 euros el Sun Apr 23 16:32:15 CEST 2023
Soy el cliente Juan y fui avisado  de bajada de precio en producto p1 a 90 euros

Soy el cliente Ana y fui avisado  de bajada de precio en producto p1 a 80 euros
Soy el cliente Juan y fui avisado  de bajada de precio en producto p1 a 80 euros
```

SE PIDE: Escribir los métodos estrictamente necesarios de las clases Producto, Cliente y EmpleadoMarketing, para que funcione el main anterior produciendo la salida anterior.

Los métodos que se usan en el ejemplo anterior son muy habituales fíjate:

- los objetos observables tiene que tener dos capacidades
  - tener la capacidad de llevar un registro de objetos que lo quieren observar para lo que es muy común añadirlos y borrarlos a una List
  - avisar a los observadores cuando se produce un cambio
- los objetos observadores tiene un método como update() que será el que use el observable para enviar un aviso al observador

Ejercicio U7_B8B_E2: Si usamos el código que generamos en el ejercicio anterior, el siguiente main no hace el borrado en la lista que se pretende. Soluciona el problema

```
class TiendaVirtual {

  public static void main(String[] args) {
    Producto p1= new Producto("p1",100);
    Cliente c1=new Cliente("Ana");
    EmpleadoMarketing e1=new EmpleadoMarketing("111","ofertas");
    p1.addObservador(c1);
    p1.addObservador(e1);
    p1.addObservador(new Cliente("Juan"));
    p1.setPrecio(90);
    System.out.println("");

    EmpleadoMarketing quieroBorrar=new EmpleadoMarketing("111","ofertas");
    p1.removeObservador(quieroBorrar);
    p1.setPrecio(80);
  }

}
```