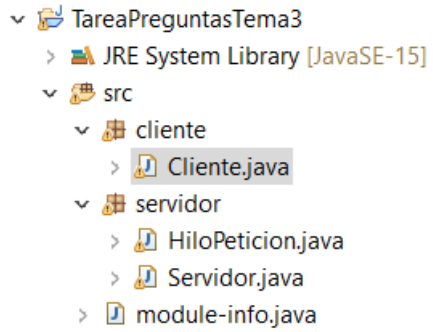
Tarea Tema 3

Contenido

1 Opción A

Vamos **a realizar un servidor y un cliente usando Sockets TCP**. El servidor **lanzará un hilo por cada petición que resolverá la petición del cliente**. La estructura del proyecto será la siguiente.



El cliente

El **cliente construirá unas preguntas (Strings) a las que el servidor contenta si o no**. El cliente se **conecta al servidor por un socket TCP por el puerto 8000**. El servidor **mandará en una lista de cadenas la pregunta con la respuesta incluida al cliente**. El servidor en HiloServidor. El resultado de la ejecución en cliente será:

```
MANDAMOS PREGUNTAS....
Escribimos las respuestas
 ¿Hace frio?   Si
 ¿Tienes hambre?   No
 ¿Es un día soleado?   No
 ¿Te gusta el pescado?   No
```

El servidor

Se **encargará primero de abrir el socket de tipo servidor TCP** por el **puerto 8000**. Segundo , de **aceptar cada petición y lanzar una instancia de HiloPeticion de tipo Thread** que **resolverá la petición SocketTCP**. De esta **manera el servidor puede seguir aceptando más peticiones**. Escribirá por pantalla:

Cuando se inicia:

```
PROGRAMA SERVIDOR INICIADO....
```

Cuando acepta una nueva petición:

```
ACEPTAMOS UNA NUEVA PETICION....
```

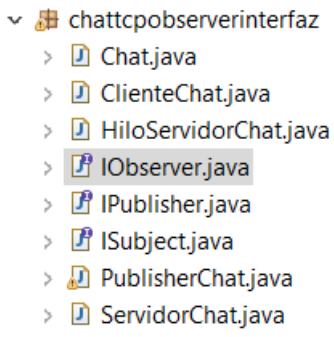
HiloPeticion es el hilo **encargado de resolver la petición del cliente**. Contestará **al cliente de** manera que para cada pregunta aleatoriamente se contesta **Si o No**. Devolverá al cliente una lista de tipo String a través del Socket TCP con las preguntas contestadas.

Entrega: se entregará el proyecto en el fichero comprimido apellidosNombreTarea3OpcionA.zip

2 Opción B (Se recomienda que hagáis está, aprenderéis mucho más).

En clase **hemos desarrollado un Chat TCP**. La **idea es modificarlo aplicando el patrón de diseño Observer**. Se **proporcionará en el Anexo I apuntes sobre el patrón observer**. El **código inicial lo encontrareis en el libro de PSP de Maria Jesus Ramos**

La estructura del proyecto será similar a:



Se proporcionan tres interfaces y el ejemplo del Anexo para que realicéis un chat siguiendo el patrón Observer.

```
package chattcpobserverinterfaz;
```

```
public interface IObserver {

	public void modificarChat(String mensaje) ;


}
```

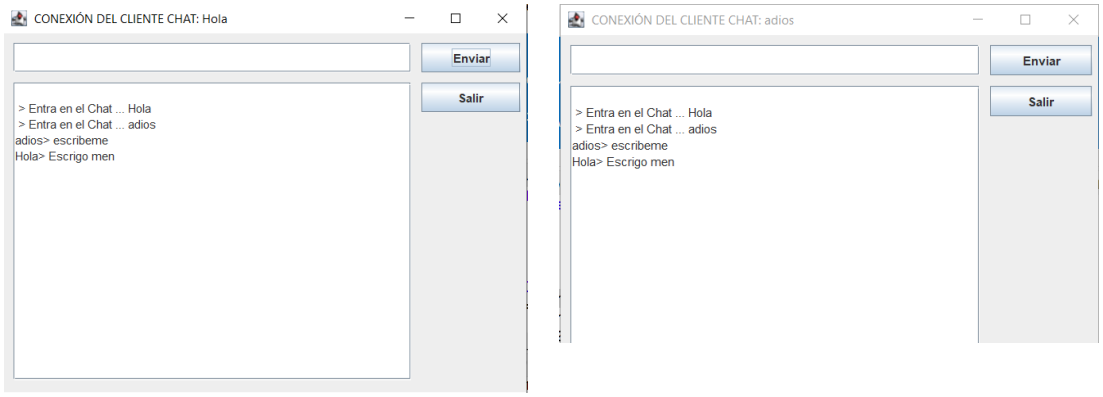
```
package chattcpobserverinterfaz;

public interface IPublisher {

	public void subscribirse(IObserver observer) throws Exception;

	public void eliminarSubscripcion(IObserver observer);

	public void notificarATodos();

}
```

```
package chattcpobserverinterfaz;

public interface ISubject {

	public  boolean cambioEstadoChat();

}
```

Entrega: se entregará el proyecto en el fichero comprimido apellidosNombreTarea3Opcionb.zip

La clase ClienteChat

Es una ventana que **nos permite recibir y enviar mensajes de/a otros clientes** conectados al Chat.



La Clase ServidorChat

Se **encargará de abrir un Socket TCP para aceptar las conexiones** al **Chat de los clientes. Para cada cliente se encargará de crear un HiloServidorChat** que **gestionará a ese cliente individualmente**. Los **puertos están definidos en el fichero de constantes** siguiente que usaran cliente y servidor.

```
public class ConstantesServidor {


	public static final int PUERTO = 44444;

}
```

En el **programa inicial controla las conexiones, debéis mantenerlo**, pero ahora todo el **control** de **conexiones lo deberéis hacer a través de SubscriberChat**.

En ejecución el servidor se ve así:

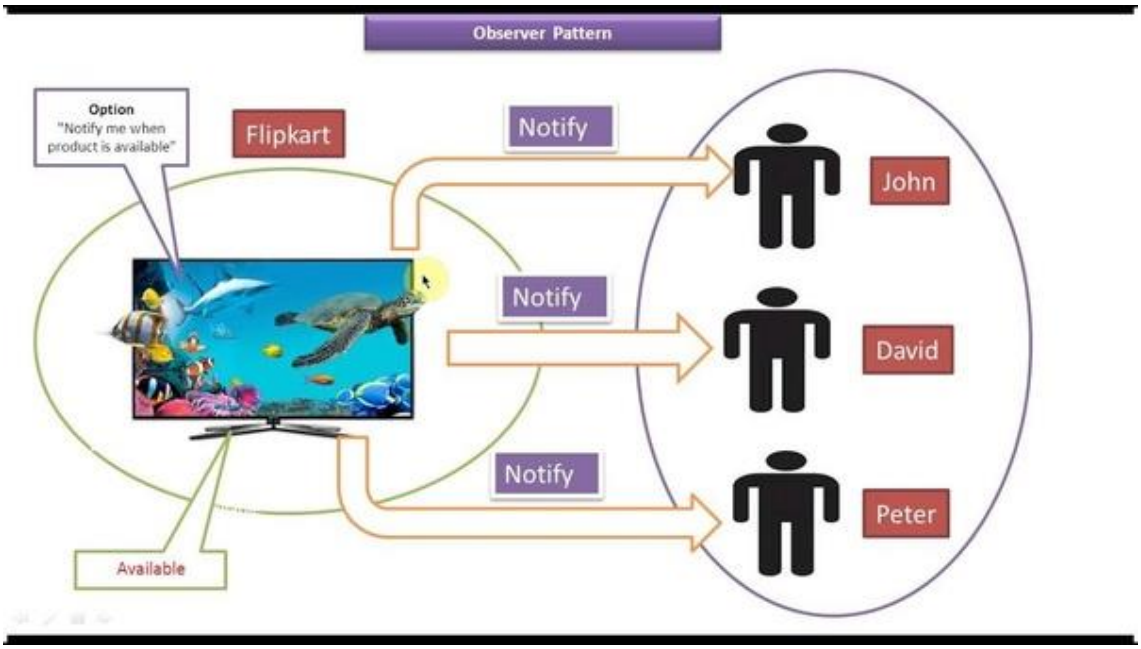
```
Servidor iniciado...
NUMERO DE CONEXIONES ACTUALES: 1
NUMERO DE CONEXIONES ACTUALES: 2
NUMERO DE CONEXIONES ACTUALES: 1
NUMERO DE CONEXIONES ACTUALES: 0
```

3 Anexo I. Patron Observer

Observer

Observer es un **patrón de diseño de comportamiento que te permite definir** un mecanismo **de suscripción para notificar a varios objetos** sobre cualquier **evento o cambio de estado.**

Usar cuando: un **cambio en un objeto requiere cambiar otros**.
Ejemplo: se usa en cualquier sistema **de notificaciones en tu móvil**, cuando **te suscribes a una página web**, cuando te **suscribes a un canal de Youtube**, para **grupos de chat**, **comercio electrónico** etc. En el siguiente ejemplo se notifica al usuario cuando un nuevo producto está disponible.



Se usa mucho en chats, **subscripciones a canales de redes sociales, de sitios web, para Streaming, web reactivas, robótica, sensores**, etc. En resumen, **es uno de los patrones con mucha aplicación** en la actualidad, conviene que **lo entendáis bien.**

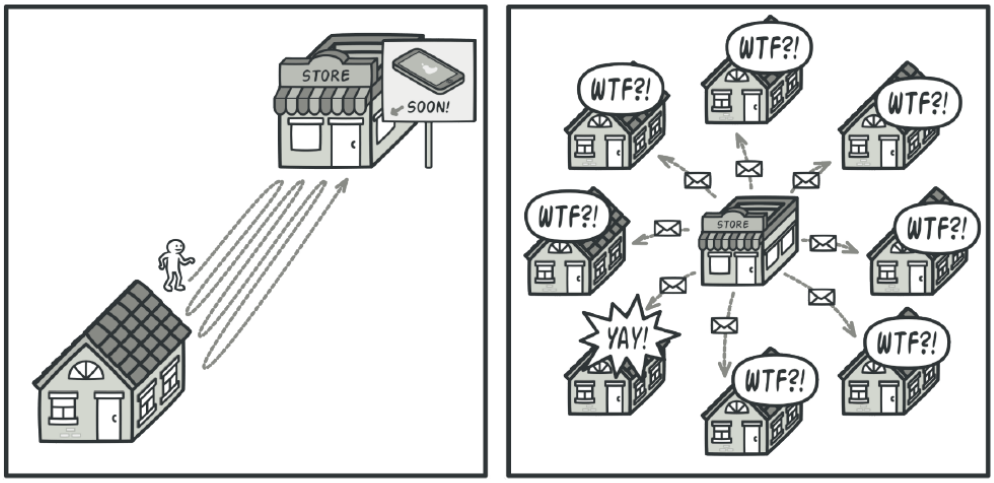
Intent

- Defina **una dependencia uno a muchos entre objetos para que cuando un objeto cambie de estado**, todos **sus dependientes sean notificados y actualizados** automáticamente.
- **Encapsular los componentes principales** (o comunes o motor) en una **abstracción sujeto y los componentes variables** (un interfaz de usuario, una aplicación cliente que consume datos) en una **jerarquía de** observers.
- La parte "**Visual" de Model-View-Controller,** un patrón de diseño de **aplicaciones que veréis el año que viene**, sería **el observer** en este caso.

Problema

Imaginad que teneis dos tipos de objetos: un tipo cliente y un tipo Tienda. El cliente está muy interesado en una marca en particular de producto (por ejemplo, es un nuevo modelo del iPhone) que debe estar disponible en la tienda pronto.
El cliente podía visitar la tienda todos los días y comprobar el producto. Disponibilidad.

Pero mientras el producto todavía está en camino, la mayoría de estos estos viajes no tienen sentido.



Visiting the store vs. sending spam

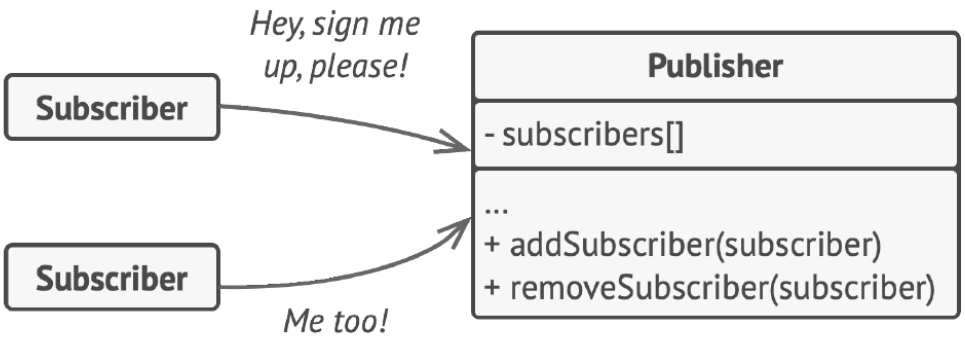
Por otro lado, **la tienda podría enviar muchísimos correos** electrónicos (que puede considerarse spam) a todos los clientes cada vez que un nuevo producto está disponible. Esto **ahorraría a algunos clientes desde interminables viajes** a la tienda. Al mismo tiempo, **habría molestado a otros clientes** que no están interesados en nuevos productos, **con correos SPAM.**

Solución

El objeto que tiene algún estado interesante a menudo se llama subject, *sujeto,* pero **ya que también va a notificar a otros objetos** sobre **los cambios en su estado**, lo **llamaremos *publisher***. Todos los d**emás objetos que desea realizar un seguimiento de los cambios en el estado** del editor se **denominan *subscriber son los observers***.

El patrón Observer propone que se agregue una suscripción mecanismo a la clase de publisher para que **los objetos individuales puedan suscribirse o darse de baja** de una **corriente de eventos que viene** de ese **publisher**. Resolverlo es bastante simple. Para implementar este mecanismo podemos usar:

1) **Un array o colección campo para almacenar una lista de referencias** a objetos observers o subscriptores.
2) Varios métodos públicos que permiten agregar suscriptores a y eliminarlos de esa lista.



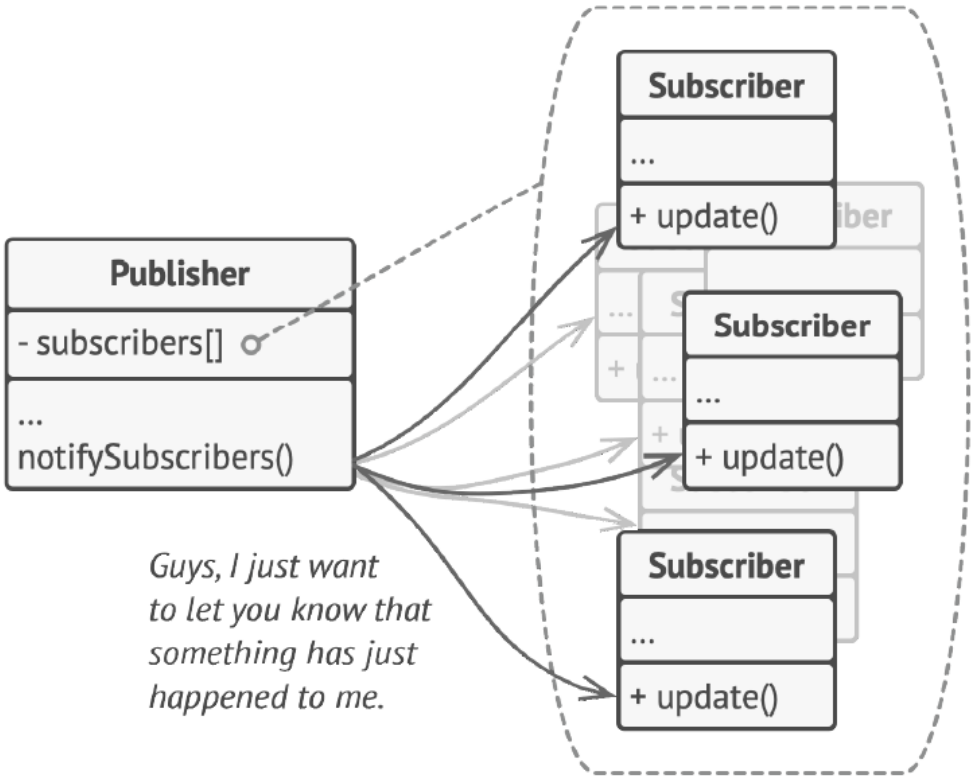
A subscription mechanism lets individual objects subscribe to event notifications.

Ahora, **cada vez que le sucede un evento importante al editor**, **examine el estado de sus suscriptores y llama a los metódos de notificación** específica **necesarios con respecto a ese cambio de estado**, por ejemplo **que el nuevo Iphone ha llegado** sería un cambio de estado .

Las **aplicaciones reales podrían tener decenas de clases de suscriptores** diferentes que están **interesados en rastrear eventos del mismo publisher**, por ejemplo, tienda, almacen, cliente anonimo, premium, etc.

No **queremos acoplar al editor todos esas Clases**, crearíamos **demasiadas subclases**. Además, es posible que ni **siquiera conozcamos algunas de esas clases** de antemano si tu **clase de publisher** se supone que **debe ser utilizado por otros desarrolladores.**

Es por eso **que es crucial que todos los suscriptores implementen lo mismo interfaz** y que el publisher se comunica con ellos sólo a través de esa interfaz. Esta interfaz debe declarar un método de notificación, con un conjunto de parámetros que el editor puede utilizar para pasar algunos datos contextuales junto con la notificación.

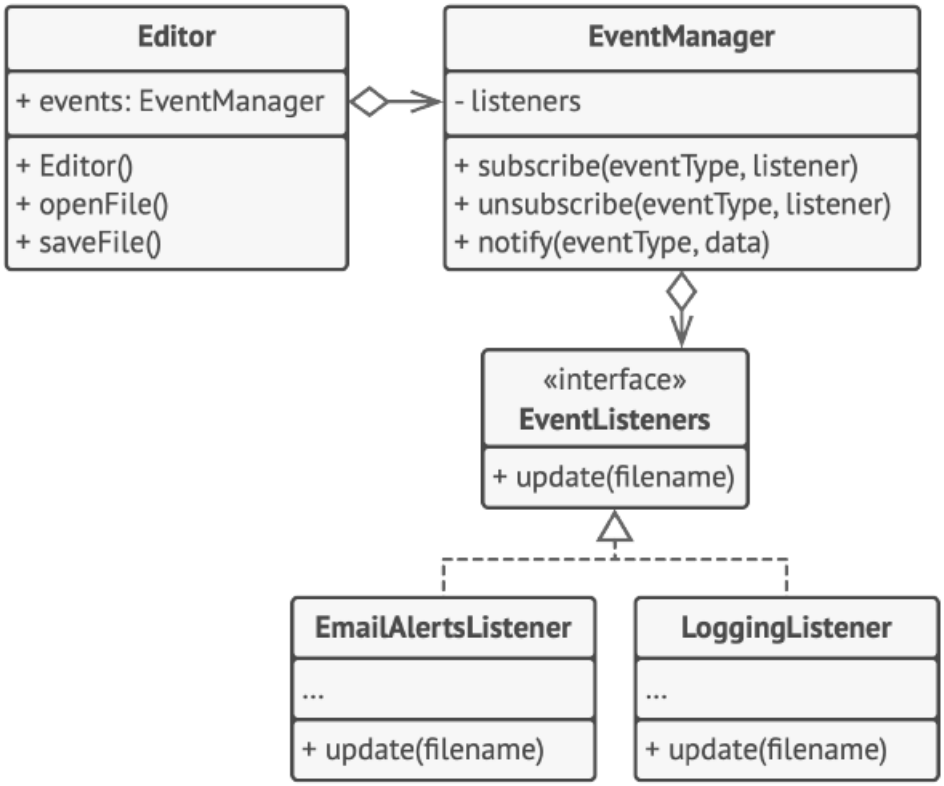


Si **la aplicación tiene varios tipos diferentes de publisher** y quieren **hacer que sus suscriptores sean compatibles con todos ellos**, puede ir aún más lejos y hacer que todos los publisher sigan la misma interfaz. Esta interfaz sólo tendría que describir algunos métodos de suscripción. La interfaz permitiría a los suscriptores para observar los estados de los editores sin acoplamiento a su clases concretas.

Ejemplo de problema resuelto con Observer

Tenemos un Editor de texto como OneDrive, en la que **reacciona a eventos, porque podemos compartir documentos con otros usuarios**, y que **estos otros usuarios lo modifiquen**. Cuando un **documento de OneDrive cambia de estado**, porque alguien lo ha modificado, **podemos notificar a los usuarios que comparten el documento**, vía email, o **en la propia aplicación si esta logeado**, el **usuario elige el método** de **suscripción.**

Fijaos **como EventListeners es el interfaz que implementan los Observers**, **Email, y Logging**. El **EventManager es el objeto Publisher**, el **que notificará a los Observer cada vez que haya un cambio de estado**. El Editor de Texto el Subject, es el que está sujeto al cambio de estado.

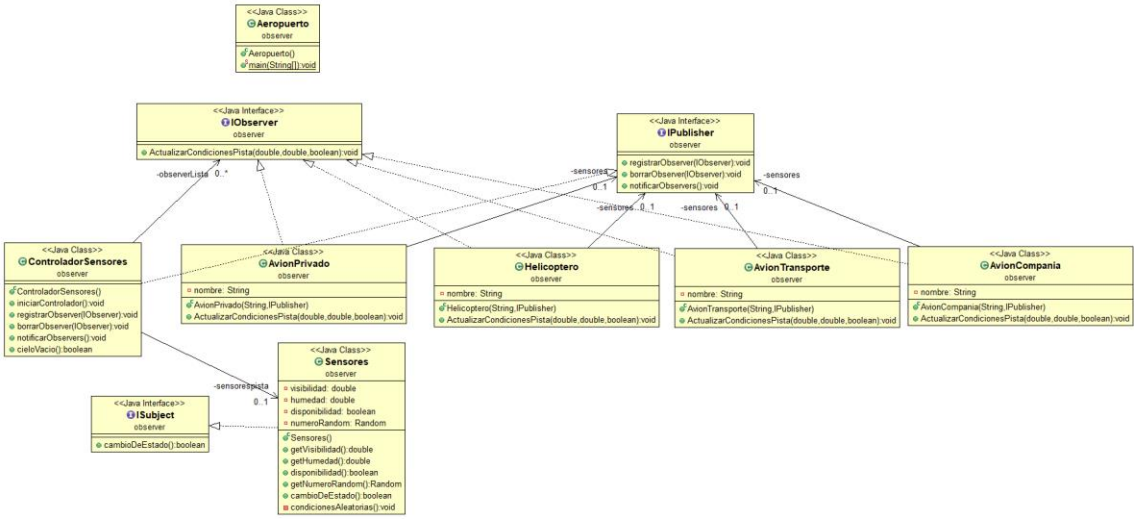


Ejemplo implementado en Java de Observer. Practica guiada Observer.

Tenemos **un aeropuerto que recibirá aparatos de distinto tipo para aterrizar**. Pero **los aparatos sólo pueden aterrizar si las condiciones climatológicas** de la pista son **correctas.**
Componentes :

- **Nuestro Subject**, sujeto **serán los sensores de la pista**, que cambia de estado cada tiempo (perdemos el tiempo con un bucle for). Sería **nuestro Publisher**, es el que **publica los cambios de estado**
- **Nuestro Subscriber un programa que controla los sensores**, **permite el registro de observers** y **notifica a nuestros observer**.
- **Nuestros Observers, serán los aviones y helicópteros** que **quieren aterrizar**, y por tanto se subscriben al servicio. Serán **nuestros Subscriber**. Cuando **los aviones hayan aterrizado quitaran las subscripción** o registro al Servicio.
- El **programa principal que crea 100 aparatos e inicializa el controlador de sensores**. Cuando **no haya más aviones en el cielo**, el **programa se termina.**

Nuestro **modelo de clases para nuestro programa. Montad el ejemplo antes de leer los apuntes.**



Sensores, el subject.

Vamos **exponiendo las diferentes clases y explicando el programa.** Empezamos con el sensor, la clase Sensores **que implementa el Interfaz ISubject** con un método para conocer el **cambio de estado del sensor, cambioDeEstado()**. Podemos **así obtener sus tres datos,** posteriormente, **desde nuestro controlador ControladorSensores.**
ISubject es el interfaz para nuestro **Subject sensores**, define :

```
public boolean cambioDeEstado();
```

Tened en cuenta que es una simulación de un sensor. Los **sensores tardan un tiempo en responder entre cambios**, lo **simulamos con un bucle for, que pierde tiempo**, en cambioDeEstado().
Después **generamos tres datos aleatorios para visibilidad, disponibilidad y humedad** en `condicionesAleatorias()`.

```
public boolean cambioDeEstado() {

        for (double i = 1; i < 1000000; i++) {

            // Perdemos tiempo
    }

    condicionesAleatorias();
```

ISubject.java

```
public interface ISubject {


	public boolean cambioDeEstado();

}
```

Sensores.java

```
import java.util.Random;

public class Sensores implements ISubject{

	private double visibilidad, humedad = 0;

	private boolean disponibilidad = false;

	private Random numeroRandom = new Random();

	public double getVisibilidad() {
		return visibilidad;
	}



	public double getHumedad() {
		return humedad;
	}



	public boolean disponibilidad() {
		return disponibilidad;
```

```
	}



	public Random getNumeroRandom() {
		return numeroRandom;
	}



	public boolean cambioDeEstado() {

		for (double i = 1; i < 1000000; i++) {

			// Perdemos tiempo
		}

		condicionesAleatorias();

		return true;

	}

	private void condicionesAleatorias() {

		visibilidad = numeroRandom.nextDouble() * 100;
		humedad = numeroRandom.nextDouble() * 100;

		if (numeroRandom.nextInt(2) == 1)
			disponibilidad = true;
		else
			disponibilidad = false;

	}

}
```

La clase ControladorSensores, el Publisher

Para el Publisher, el encargado de controlar los sensores también definimos un interfaz, IPublisher, con tres métodos definidos para registrar, borrar el registro y notificar a nuestros observers.

IPublisher.java

```
public interface IPublisher {

        public void registrarObserver(IObserver o);

        public void borrarObserver(IObserver o);

        public void notificarObservers();

}
```

Y **seguimos con la clase ControladorSensores**, que **funcionará como un publisher**, recogiendo **los datos del sensor cuando haya un cambio y comunicándolo a todos los Observers**, los **aparatos que tienen que aterrizar**. Los **observers son guardados y borrados de un HashSet**, propiedad **privada de la clase**. Crearemos **unos sensores en el constructor de la clase**, guardados en sensorespista.

```
private HashSet<IObserver> observerLista

private Sensores sensorespista;
```

El **método iniciaControlador**, nos **permite iniciar el controlador y dar respuesta a nuestros Observers registrados**. Mientras **en el cielo haya aviones el controlador no acaba**. Escucha cambios de estado del sensor, y cuando se produce notifica a los observers.

```
if (sensorespista.cambioDeEstado())
                            notificarObservers();
```

```
public void iniciarControlador() {
```

```
            while(!cieloVacio()) {

                  if (sensorespista.cambioDeEstado())
                        notificarObservers();
            }

}
```

Registrarobserver y borrarObserver, añade o quita el observer del HashSet. Son **sencillos.** Por último, **notificarObserver, recorre el HashSet con un Iterator, y actualiza las condiciones de pista** para cada **Observer** llamando **al método del Observer** actualiza condiciones de pista. **Clonamos la lista** para que no haya interferencias de observers que han aterrizado, y la recorremos con el Iterator.

```
HashSet<IObserver> listaObservers = (HashSet<IObserver> ) observerLista.clone();

Iterator<IObserver> it = listaObservers.iterator();
                  IObserver ob;
                  while (it.hasNext()) {
```

Avisamos a cada Observer de que las **condiciones de la pista de aterrizaje han cambiado**. Y con esto **hemos terminado con el Publisher**. Vamos con los **Observers.**

```
ob.ActualizarCondicionesPista(
                              sensorespista.getVisibilidad(),
      sensorespista.getHumedad(), sensorespista.disponibilidad());
```

ControladorSensores.java

```
import java.util.HashSet;
import java.util.Iterator;
public class ControladorSensores implements IPublisher{

      private HashSet<IObserver> observerLista = new HashSet<IObserver>();
```

```
private Sensores sensorespista;

public ControladorSensores() {

        sensorespista = new Sensores();



}

public void iniciarControlador() {

        while(!cieloVacio()) {

                sensorespista.cambioDeEstado();

                notificarObservers();
        }

}

@Override
public void registrarObserver(IObserver o) {
        // TODO Auto-generated method stub
        observerLista.add(o);

}
```

```
	@Override
	public void borrarObserver(IObserver o) {
		// TODO Auto-generated method stub
		observerLista.remove(o);

	}




	@Override
	public void notificarObservers() {
		// TODO Auto-generated method stub


			HashSet<IObserver> listaObservers = (HashSet<IObserver> ) observe
rLista.clone();

			Iterator<IObserver> it = listaObservers.iterator();
			IObserver ob;
			while (it.hasNext()) {

			ob=it.next();
			ob.ActualizarCondicionesPista(
				sensorespista.getVisibilidad(),

					sensorespista.getHumedad(), sensorespist
a.disponibilidad());

			}




	}
```

```
        public boolean cieloVacio() {


                return (observerLista.size()<=0);

        }

}
```

Los Observers, Aviones y Helicopteros.

Nota: Podíamos **haber llamado Subscribers a los Observers**, pero como estamos haciendo un ejemplo de Sensores, **he preferido dejarlo como Observers**, ambos son válidos.

Como **ya hemos dicho los aparatos están en el aire esperando a ver que las condiciones sean favorables** para **aterrizar**, humedad, visibilidad, y que la pista este disponible. Todos los aparatos **implementan un interfaz común IObserver**, que define el **método actualizaCondicionesPista**. Cuando el publisher llama a este Método, el Observer reacciona a los cambios y comprueba si puede aterrizar.

IObserver.java

```
public interface IObserver {


        public void ActualizarCondicionesPista(double visibilidad, double humedad, boolean
disponibilidad);


}
```

Tenemos **cuatro clases que implementan el Observer, AvionCompania, AvionPrivado, AvionTranporte y Helicóptero**. Todas se comportan igual en el ejemplo, pero podría haber diferencias en la manera de aterrizar o dependiendo de las condiciones de la pista.
Empezamos **a explicar AvionCompania**, el resto son iguales. Y lo primero **es ver su constructor**, que **recibe como parámetro un nombre y un parámetro de tipo IPublisher,** que en el programa principal será de tipo **ControladorSensores**, nuestra **implementación de Publisher**. Conforme **se crea un objeto de la clase se registra** en el **Publisher** . Eso quiere decir que **el avión quiere aterrizar**.

```
public AvionCompania(String nombre , IPublisher sensores)
this.sensores.registrarObserver(this);
```

Lo siguiente **es el método con el que el Subscriber** avisa **al AvionCompania** de que las **condiciones de la pista han cambiado** `public void ActualizarCondicionesPista`. El **avión** pierde algo de tiempo en aterrizar**, con el for**, y **comprueba que las condiciones para aterrizar,** hay disponibilidad y la visibilidad y humedad están dentro de los **umbrales razonables para el aterrizaje**. Si **aterriza al avión ya no le hace falta conocer datos** de los sensores, se borra de la subscripción con sensores.borrarObserver(this);.

```
if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {

                    System.out.println("Avion de compañia Aterriza de nombre:" + nombre);
                    sensores.borrarObserver(this);
```

```
@Override
        public void ActualizarCondicionesPista(double visibilidad, double humedad, boolean
disponibilidad) {

                for (double i=1; i<1000000 ;i++) {

                        //Perdemos tiempo
        }

                if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {

                        System.out.println("Avion de compañia Aterriza de nombre:" + nomb
re);
                        sensores.borrarObserver(this);


                }
```

```
        }
```

El **resto de Observers son iguales, os dejo el código** pero no hay nada más que explicar. Importante,

Importante. Programamos para las abstracciones no para las implementaciones.
AvionCompania, no recibe un ControladorSensores el Publisher, recibe su interfaz IPublisher.
`public AvionCompania(String nombre , IPublisher sensores)`
Podríamos cambiar de Publisher y sería transparente para los Observers.

AvionCompania.java

```
public AvionCompania(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                 this.sensores.registrarObserver(this);

        }
```

AvionCompania.java

```
public class AvionCompania implements IObserver {

        private String nombre;;
        private IPublisher sensores;



        public AvionCompania(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                 this.sensores.registrarObserver(this);
```

```
        }

        @Override
        public void ActualizarCondicionesPista(double visibilidad, double humedad, boolean
disponibilidad) {

                for (double i=1; i<1000000 ;i++) {

                        //Perdemos tiempo
        }

                if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {

                        System.out.println("Avion de compañia Aterriza de nombre:" + nomb
re);
                        sensores.borrarObserver(this);


                }




        }



}
```

AvionPrivado.java

```
public class AvionPrivado implements IObserver {

        private String nombre;;
        private IPublisher sensores;
```

```
        public AvionPrivado(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                this.sensores.registrarObserver(this);

        }

        @Override
        public void ActualizarCondicionesPista(double visibilidad, double humedad, boolean
disponibilidad) {

                for (double i=1; i<1000000 ;i++) {

                        //Perdemos tiempo
        }

                if (disponibilidad && visibilidad > 0.5 && humedad >95 ) {

                        System.out.println("Avion privado Aterriza de nombre:" + nombre);

                        sensores.borrarObserver(this);
                }

        }




}
```

AvionTransporte.java

```
public class AvionTransporte implements IObserver {
```

```
        private String nombre;;
        private IPublisher sensores;

        public AvionTransporte(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                this.sensores.registrarObserver(this);

        }

        @Override
        public void ActualizarCondicionesPista(double visibilidad, double humedad, boolean
disponibilidad) {

                for (double i=1; i<1000000 ;i++) {

                        //Perdemos tiempo
        }

                if (disponibilidad && visibilidad > 0.1 && humedad <99 ) {

                        System.out.println("Avion de transporte Aterriza de nombre:" + no
mbre);

                        sensores.borrarObserver(this);
                }

        }

}
```

Helicóptero.java

```
public class Helicoptero implements IObserver {
        private String nombre;;
        private IPublisher sensors;
```

```
        public Helicoptero(String nombre , IPublisher sensores) {

                this.nombre=nombre;
                this.sensores=sensores;
                 this.sensores.registrarObserver(this);

        }

        @Override
        public void ActualizarCondicionesPista(double visibilidad, double humedad, boolean
disponibilidad) {
                // TODO Auto-generated method stub

        }
}
```

El programa principal, Aeropuerto

Creamos el ControladorSensores contenido en una variable de tipo Interfaz IPublisher.

```
ControladorSensores controladorsensor = new ControladorSensores() ;
```

Creamos 400 aviones en el for

```
       for (int i=0; i<100; i++) {
             new AvionCompania("avioncomp" +i , controladorsensor);
             new AvionPrivado("avionpri" +i ,controladorsensor);
```

E inicializamos nuestro ControladorSensores

```
controladorsensor.iniciarControlador();
```

Aeropuerto.java

```
import java.util.Scanner;
public class Aeropuerto {


	public static void main(String[] args) {

	boolean finPrograma= false;

	ControladorSensores controladorsensor = new ControladorSensores() ;




	for (int i=0; i<100; i++) {

		new AvionCompania("avioncomp" +i , controladorsensor);
		new AvionPrivado("avionpri" +i ,controladorsensor);
		new AvionTransporte("aviontransporte" +i, controladorsensor);

		new AvionTransporte("helicoptero" +i, controladorsensor);


	}


	controladorsensor.iniciarControlador();
```

```
}


}
```

Un ejemplo de ejecución sería. Probadlo.

```
Avion de compañia Aterriza de nombre:avioncomp97
Avion privado Aterriza de nombre:avionpri51
Avion de compañia Aterriza de nombre:avioncomp66
Avion de compañia Aterriza de nombre:avioncomp80
Avion de compañia Aterriza de nombre:avioncomp0
Avion privado Aterriza de nombre:avionpri17
Avion privado Aterriza de nombre:avionpri83
```