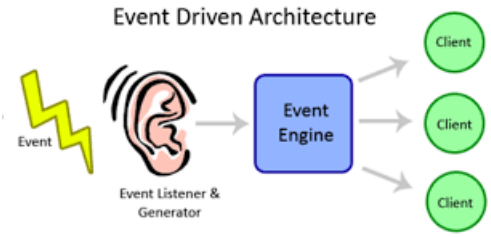
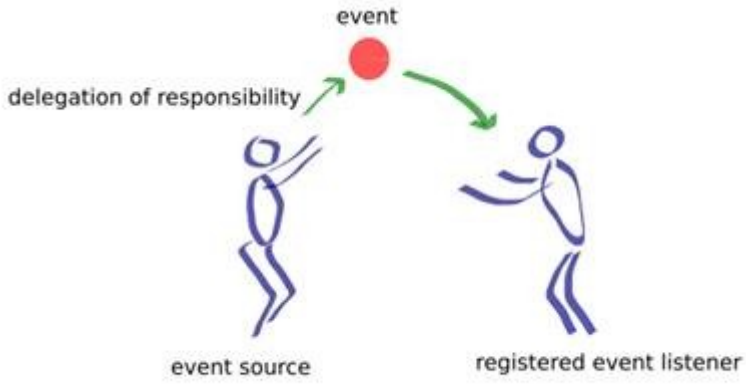
1. Eventos.

Es necesario que los diferentes elementos software se comuniquen entre si. Ya se ha visto la comunicación entre objetos utilizando los métodos de estos, pero estos no responden a situaciones asíncronas, como puede ser la llegada de un paquete por la tarjeta de red, o ante las acciones del usuario al pulsar un botón.

Estas situaciones también en la vida real, la comunicación con las personas puede ser la ordinaria (quedar por la tarde para hacer las prácticas de programación) aunque pueden suceder situaciones no esperadas (se ha roto el ordenador de casa y no es posible hacerlas) siendo necesario avisar a las personas interesadas para que actúen en consecuencia.



Se necesita establecer algún mecanismo que permita “avisar” a un conjunto objetos o elementos de situaciones asíncronas para que puedan actuar en consecuencia.



Un ejemplo puede ser preparar el desayuno, en una programación secuencial los pasos son.

1. Abrir frigorífico.
2. Coger la leche.
3. Cerrrar frigorífico.
4. Encender el fuego.
5. Poner a calentar la leche.
6. Esperar a que la leche esté caliente.
7. Poner la leche en el vaso.

8. Cortar el pan.
9. Conectar la tostadora.
10. Cologar el pan.
11. Esperar a que el pan este tostado.
12. Apagar la tostadora.
13. Poner el pan en el plato.
14. Desayunar.

Lo ideal sería tener dos métodos, uno para preparar la leche y otro las tostadas, y que al finalizar dichos métodos me avisara de que ha finalizado.

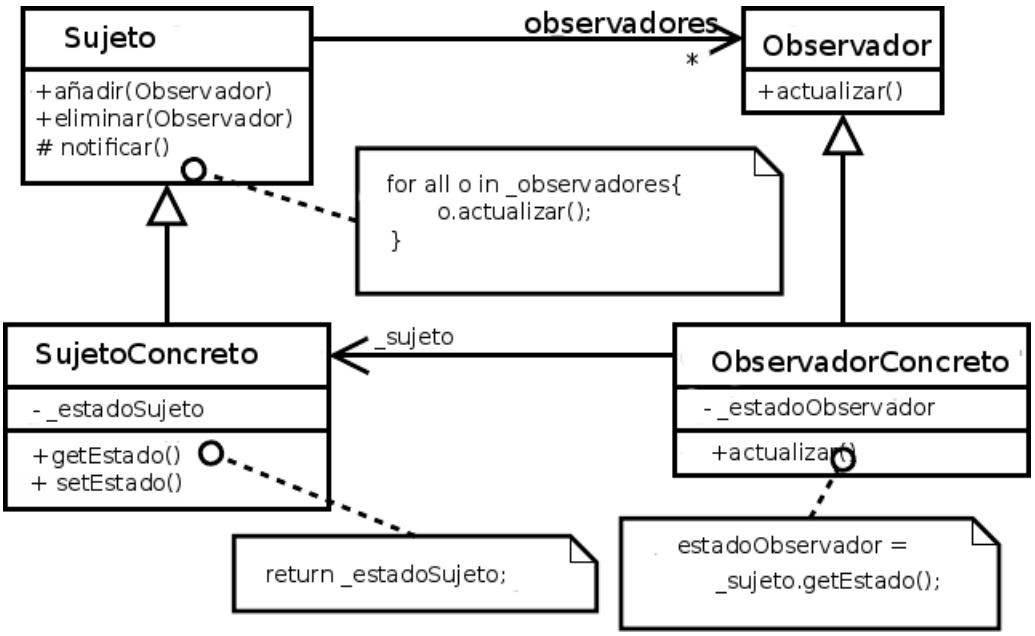
- Método prepararLeche(Vaso, Leche, Cazo, Vitro, al_terminar_avisame);
- Méotodo prepararTostada(Pan,Cuchillo,Plato,Tostadora, al_terminar_avisame);

El parámetro al_terminar_avisame, es un método que se ejecutara al finalizar

2. Patrón Observer.

Se hace necesario desarrollar un conjunto de clases que permitan gestionar los eventos que suceden en los programas. Existe un patrón de diseño que permite añadir a los diferentes componentes software la capacidad para la gestión de eventos, tanto emitirlos como recibirlos, el patrón es conocido como Observer.

Este patrón utiliza diferentes Interfaces que al ser implementada en las clases permiten suscribirse a eventos concretos de un objeto, y a su vez este objeto es capaz de “avisar” a los objetos subscritos a los mismos para tratarlos.



El patrón Observer proporciona funcionalidades de los sistemas que se comunican mediante mensajes:

- No hay que estar monitorizando un objeto en búsqueda de cambios, se hace la notificación cuando un objeto cambia.
- Permite agregar nuevos observadores para proporcionar otro tipo de funcionalidad sin cambiar el objeto observador.
- Bajo acoplamiento entre observable y observador, un cambio en el observable u observador no afecta al otro.

 Programación reactiva. https://www.youtube.com/watch?v=WvGIHxUYIgA

Un ejemplo sencillo de implentación del patrón Observer, para avisar a los objetos ante un fallo en una máquina de fabricación de calzado.

```
/**
 *
 * @author Pedro
 */
enum Estado {
 ENCENDIDO, APAGADO, FALLO, FALLO_ELECTRICO, SIN_MATERIAL
}

interface IObservadorMaquina {

 public void avisarCambioEstado(Object nuevoestado);

 public void avisarFalloElectrico();
}
```

```
abstract class GestorEventos {

 private LinkedList<IObservadorMaquina> observadores;
 private LinkedList<IObservadorMaquina> observadoresfalloElectrico;

 public GestorEventos() {
  this.observadores = new LinkedList<IObservadorMaquina>();
  this.observadoresfalloElectrico = new LinkedList<IObservadorMaquina>();
 }

 public void addListener(IObservadorMaquina observador) {
```

```
  this.observadores.add(observador);
 }

 public void removeListener(IObservadorMaquina observador) {
  this.observadores.remove(observador);
 }

 public void addListenerElectrico(IObservadorMaquina observador) {
  this.observadoresfalloElectrico.add(observador);
 }

 public void removeListenerElectrico(IObservadorMaquina observador) {
  this.observadoresfalloElectrico.remove(observador);
 }
 public void avisaErrorElectrico() {
  for (IObservadorMaquina elemento : this.observadoresfalloElectrico) {
   elemento.avisarFalloElectrico();
  }
 }
 public void avisaCambio(Object sender) {
  for (IObservadorMaquina elemento : this.observadores) {
   elemento.avisarCambioEstado(sender);
  }
 }
}
```

```
class Maquina extends GestorEventos {

 private Estado estado;

 public Maquina() {
  super();
  this.estado = Estado.APAGADO;
 }
 public void changeEstado(Estado nuevoestado) {
  if (!this.estado.equals(nuevoestado)) {
   this.estado = nuevoestado;
   this.avisaCambio(this.estado);
   if (this.estado.equals(Estado.FALLO_ELECTRICO)) {
    this.avisaErrorElectrico();
   }
  }
 }
```

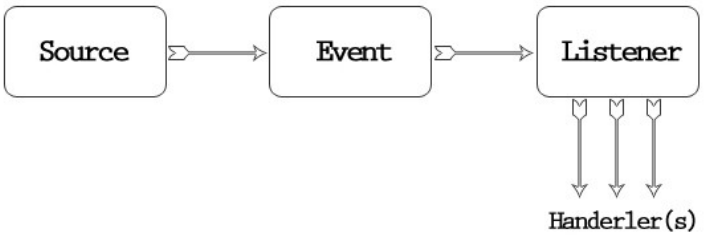
```
}
```

```
class Encargado implements IObservadorMaquina {

 @Override
 public void avisarCambioEstado(Object nuevoestado) {
  System.out.println("Soy en encargado y se ha producido un cambio");
 }
 @Override
 public void avisarFalloElectrico() {
  System.out.println("Soy en encargado y se ha producido un ERROR");
 }

}
```

```
class ResponsableMantenimiento implements IObservadorMaquina {

 @Override
 public void avisarCambioEstado(Object nuevoestado) {
  System.out.println("Soy en responsable y se ha producido un cambio");
 }
 @Override
 public void avisarFalloElectrico() {
  System.out.println("Soy en responsable,ERROR, voy CORRIENDO");
 }
}
```

```
public class PatronnObserver {

 /**
  * @param args the command line arguments
  */
 public static void main(String[] args) {

  Maquina maquinaa = new Maquina();
  Maquina maquinab = new Maquina();
  Encargado encargado = new Encargado();
  ResponsableMantenimiento responsablea = new ResponsableMantenimiento();
  ResponsableMantenimiento responsableb = new ResponsableMantenimiento();
  maquinaa.addListener(encargado);
  maquinaa.addListener(responsablea);
  maquinaa.addListener(responsableb);
  maquinab.addListener(encargado);
```

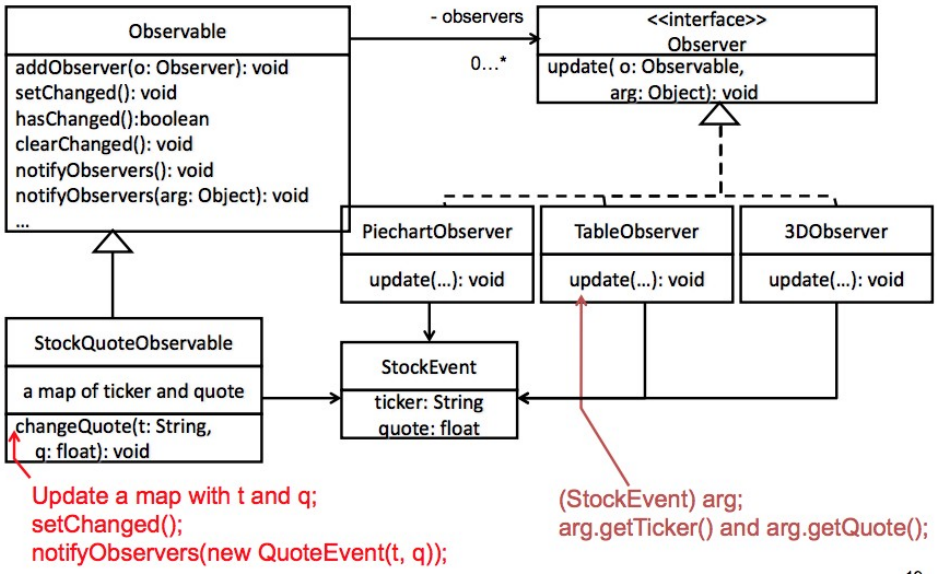
```
  maquinab.addListener(responsablea);
  maquinaa.changeEstado(Estado.ENCENDIDO);
  maquinab.changeEstado(Estado.FALLO_ELECTRICO);
 }

}
```



Event-Driving Programming Model

2.1. Java Observer (Deprecated).

Si bien en la actividad anterior se ha implementado el patrón Observable desde cero, no parece muy lógico que cada desarrollador o desarrolladora implemente de forma independiente el mismo. Java cuenta entre sus librerías y paquetes la implementación del patrón, de forma que solo es necesario implementar las interfaces y heredar de las clases correctas.



Al heredad de Observable, se dispone de una serie de métodos para “avisar” a los objectos suscritos, la estructura de datos con la que se gestiona es implementada por la clase Observable, no siendo necesario gestionarla desde la clase que hereda, de la misma forma en la clase que observa, con la salvedad de que en este caso es una interfaz y es necesario implementar el método update.

Un ejemplo de uso de la implementación del patrón:

```
import java.util.Observable;
public class ObservableValue extends Observable
{
   private int n = 0;
   public ObservableValue(int n){
      this.n = n;
   }
   public void setValue(int n){
      this.n = n;
      setChanged();
      notifyObservers();
   }
   public int getValue(){
      return n;
   }
}

import java.util.Observer;
import java.util.Observable;
public class TextObserver implements Observer
{
   private ObservableValue ov = null;
   public TextObserver(ObservableValue ov){
         this.ov = ov;
   }
   public void update(Observable obs, Object obj)    {
         if (obs == ov){
               System.out.println(ov.getValue());
         }
   }
}

public class Main
{
   public Main()
   {
   ObservableValue ov = new ObservableValue(0);
   TextObserver to = new TextObserver(ov);
   ov.addObserver(to);
   }
   public static void main(String [] args)
   {
   Main m = new Main();
```

```
    }
}
```

2.2. JavaBeans.

La interfaz observable fue marcada como “deprecated” a partir de JDK 9, recomendando el uso de un modelo de componentes denominadao **JavaBeans,** en el que se encapsulan varios objetos (agregación), estableciendose una serie de condiciones como nombre de los métodos, construcción y forma de comportanrse, de forma que sean reusable. Las condiciones son:

- Existir un constructor sin parámetros.
- Atributos privados.
- Los atributos han de ser accesibles con get y set.
- Debe ser serializable.

 ¿Cómo hacer una clase serializable?

La estructura de un JavaBean es:

1. Atributos privados.
2. Constructor por defecto.
3. Métodos, en especial getters y setters.
4. **Eventos.**

Como se puede ver los JavaBeans cumplen con los principios indicados en la introducción orientada a objetos tratados en el presente curso.

Este modelo es usado en múltiples situaciones, por ejemplo y relacionado con el tema, los componentes de la librería gráficas Swing o AWT lo implementan, y en parte JavaFX aunque con matices.

El punto 4, es la novedad con respeto a las clases vistas hasta ahora. Este modelo es similar al anterior “deprecated”, teniendo que definir los siguientes elementos

- Interfaz que han de cumplir los escuchadores. Normalmente se definen métodos para manejar el evento denominados Handles(manejadores).
- Estructuras y métodos para gestionar los escuchadores en el objeto/componente del que se desea escuchar.
- Definición de clase evento, que se creará al producirse el cambio, contendrá información diversa necesaria para tratar el evento.

 ¿Con qué estructura interna se getiona el aviso de los “Listener”?

La documentación oficial de Java indica ejemplos de la sintaxis:

```
public void add<Event>Listener(<Event>Listener a)
public void remove<Event>Listener(<Event>Listener a)
```

Donde Event describe el evento Action,Click,KeyPressed, KeyRelease, Stop, Connect, Disconnect… y cualquier otro que sea necesario definir.

Un ejemplo de uso de los eventos en la clase Jbutton de Swing (representa un botón de la interfaz gráfica):

```
public void addActionListener(ActionListener l);
public void removeActionListener(ActionListener l);
```

Se observa que para añadir y borrar escuchadores se pasa como parámetro objetos que han de implementar una interfaz, se puede definir interfaces desde 0 o extender de la **interfaz EventListener del paquete java.util.EventLinstener,** aunque está interfaz está vacia.

Tambíen se ha de crear una clase que se use para proporcionar información de los eventos a los escuchadores.

- Para definir el evento se hereda de java.util.EventObject, aunque este evento puede ser de cualquier clase que se defina, es recomendable heredar de esta.

La principal característica de está clase es posee un atributo de tipo Object que es el origen del evento.

Constructor Summary

Constructors

Constructor	Description
`EventObject(Object source)`	Constructs a prototypical Event.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`Object`	`getSource()`	The object on which the Event initially occurred.
`String`	`toString()`	Returns a String representation of this EventObject.

Algunas de las clases que heredan dentro del JDK:

AppEvent, AWTEvent, BeanContextEvent, CaretEvent, ChangeEvent, ConnectionEvent, DragGestureEvent, DragSourceEvent, DropTargetEvent, FlavorEvent, HandshakeCompletedEvent, HyperlinkEvent, LineEvent, ListDataEvent, ListSelectionEvent, MenuEvent, NamingEvent, NamingExceptionEvent, NodeChangeEvent, Notification, PopupMenuEvent, PreferenceChangeEvent, PrintEvent, PropertyChangeEvent, RowSetEvent, RowSorterEvent, SSLSessionBindingEvent, StatementEvent, TableColumnModelEvent, TableModelEvent, TreeExpansionEvent, TreeModelEvent, TreeSelectionEvent, UndoableEditEvent, UnsolicitedNotificationEvent

Por ejemplo se está desarrollando un sistema domótico con un conjunto de dispositivos similar a la práctica realizada en temas anteriores, el sistema tiene:

- Light.

- Dimmer.
- Interruptor.
- A/C.
- Speaker.
- PresenceSensor.
- Thermometer.
- SystemLog(almacena todos los eventos que se produce en el sistema para su análisis y control).

Se pide desarrollar el sensor de presencia, y la luz, al detectarse presencia la luz ha de encenderse y apagarse cada segundo, cuando se deje de detectar presencia, se han de apagar la luz se ha de apagar.

Se define en primer lugar el JavaBean para el sensor de presencia,está claro que ha de tener un booleano o un enumerador que indique si ha detectado presencia o no, junto con una estructura para almacenar los escuchadores, se crea la clase para el evento del sensor y la interfaz de los escuchadores:

Clase SensorEvent:

```
public class SensorEvent extends  EventObject{


    public SensorEvent(Object source){
        super(source);
    }
}
```

La interfaz de ISensorListener:

```
public interface ISensorListener extends java.util.EventLinstener{
    public void onSensorChange(SensorEvent event);
}
```

El enumerado con los estados:

```
public enum EBooleanState {
    ON,
    OFF
}
```

Y por ultimo la clase PresenceSensor:

```
public class PresenceSensor {
    private EBooleanState state;
    private Vector<ISensorListener> sensorlisteners;
    public PresenceSensor(){
        this.sensorlisteners= new Vector<>();
    }
```

```
    public void addChangeStateListener(ISensorListener l){
        this.sensorlisteners.add(l);
    }
    public void removeChangeStateListener(ISensorListener l){
        this.sensorlisteners.remove(l);
    }

    public EBooleanState getState() {
        return state;
    }

    public void setState(EBooleanState state) {
        boolean sendevent=false;
        if(state!=this.state)
            sendevent=true;
        this.state = state;
        if(sendevent){
            onChangeState();
        }
    }
    private void onChangeState(){
        SensorEvent event= new SensorEvent(this);
        //se avisa
        this.sensorlisteners.forEach(sl->sl.onSensorChange(event));
    }
}
```

Ahora se define la luz que implementarán la interfaz IsensorListener, además de ser Runnable para poder encenderse y apagarse de forma autónoma:

```
public class Light implements ISensorListener, IDevice, Runnable {

    private EBooleanState state;
    private DeviceState deviceState;
    private Boolean modo_alarma;
    private static long tiempo_sleep = 1000;
    private Thread t;

    public Light() {
        this.deviceState = deviceState.Stopped;
        this.state = EBooleanState.OFF;
        this.modo_alarma = false;
    }
```

```
    public synchronized void setOn() {

        this.state = EBooleanState.ON;
        System.out.println("Se ha encendido la luz");

    }

    public synchronized void setOff() {
        this.state = EBooleanState.OFF;
        System.out.println("Se ha apagado la luz");

    }

    private synchronized void changeDeviceState(DeviceState d) {
        this.deviceState = d;
    }

    public EBooleanState getLightState() {
        return this.state;

    }

    @Override
    public void onSensorChange(SensorEvent event) {
        if (((PresenceSensor) event.getSource()).getState() == state.OFF) {
            this.modo_alarma = false;
            this.setOff();
        } else {
            this.modo_alarma = true;
        }
    }

    @Override
    public void run() {
        while (this.deviceState == deviceState.Running) {
            try {
                Thread.sleep(Light.tiempo_sleep);
                if (this.modo_alarma) {
                    this.Switch();
                }
            } catch (InterruptedException ex) {
                Logger.getLogger(Light.class.getName()).log(Level.SEVERE, null,
ex);
```

```
            }
        }
    }

    private void Switch() {
        if (this.state == EBooleanState.OFF) {
            this.setOn();
        } else {
            this.setOff();
        }
    }

    @Override
    public void Start() {
        if (this.deviceState == deviceState.Stopped) {
            this.changeDeviceState(deviceState.Running);

            this.t = new Thread(this);
            this.t.start();
        }

    }

    @Override
    public void Stop() {
        this.changeDeviceState(deviceState.Stopped);
    }

    @Override
    public DeviceState getDeviceState() {
        return this.deviceState;
    }
}
```

Un ejemplo de programa principal en el que la luz se subscribe a los eventos de la alarma:

```
public class Principal {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
       PresenceSensor sensor= new PresenceSensor();
       Scanner s= new Scanner(System.in);
```

```
        int opcion=-1;
        Light l1= new Light();
        sensor.addChangeStateListener(l1);

        l1.Start();
        do{
            System.out.println("Introduccir opción:");
            System.out.println(" 1. Activar alarma.");
            System.out.println(" 2. Desactivar alarma.");
            System.out.println("0. Salir.");

            opcion=s.nextInt();
            if(opcion==1)
                sensor.setState(EBooleanState.ON);
            else if (opcion==2)
                sensor.setState(EBooleanState.OFF);

        }while (opcion>0);
        l1.Stop();
    }

}
```

 Crear la clase Speaker y subscriberse a los eventos del sensor de presencia, cuando se detecte presencia ha de sonar el altavoz.