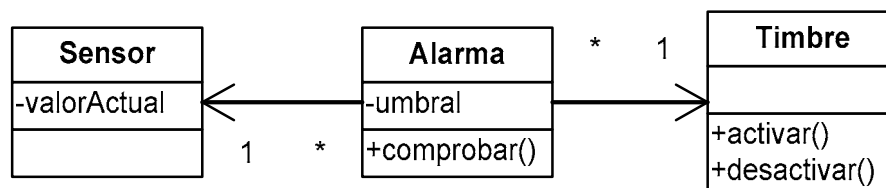


Clases y objetos

Ejercicio resuelto

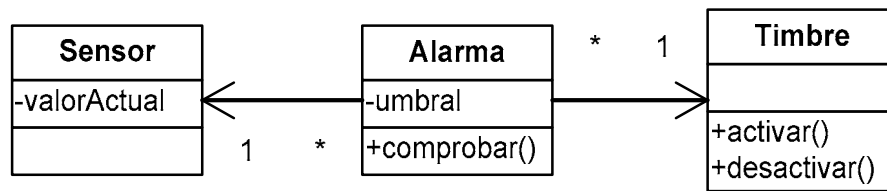
Cree una clase denominada **Alarma** cuyos objetos activen un objeto de tipo **Timbre** cuando el valor medido por un **Sensor** supere un umbral preestablecido:



Implemente en Java todo el código necesario para el funcionamiento de la alarma, suponiendo que la alarma comprueba si debe activar o desactivar el timbre cuando se invoca el método `comprobar()`.

Cree una subclase de **Alarma** denominada **AlarmaLuminosa** que, además de activar el timbre, encienda una luz (que representaremos con un objeto de tipo **Bombilla**).

NOTA: Procure eliminar la aparición de código duplicado al crear la subclase de **Alarma** y asegúrese de que, cuando se activa la alarma luminosa, se enciende la luz de alarma y también suena la señal sonora asociada al timbre.



```

/**
 * Alarma
 */

public class Alarma
{

    private Sensor sensor;
    private Timbre timbre;
    private double umbral;

    /**
     * Constructor
     */

    public Alarma
        (Sensor sensor, Timbre timbre, double umbral)
    {
        this.sensor = sensor;
        this.timbre = timbre;
        this.umbral = umbral;
    }

    /**
     * Comprobar estado de la alarma
     */

    public void comprobar ()
    {
        if (sensor.getValorActual() > umbral) {
            timbre.activar();
        } else {
            timbre.desactivar();
        }
    }
}

```

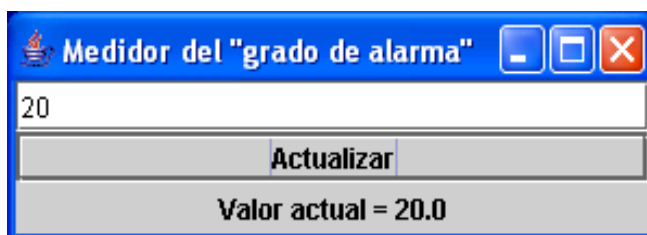
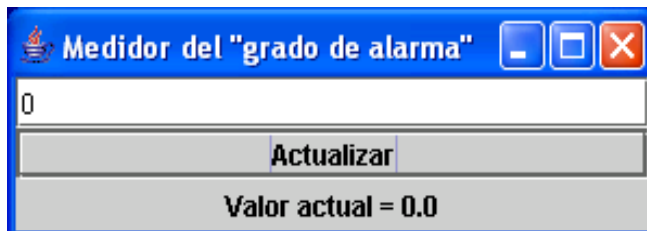
Programa principal:

```
public class Programa
{
    // Programa principal

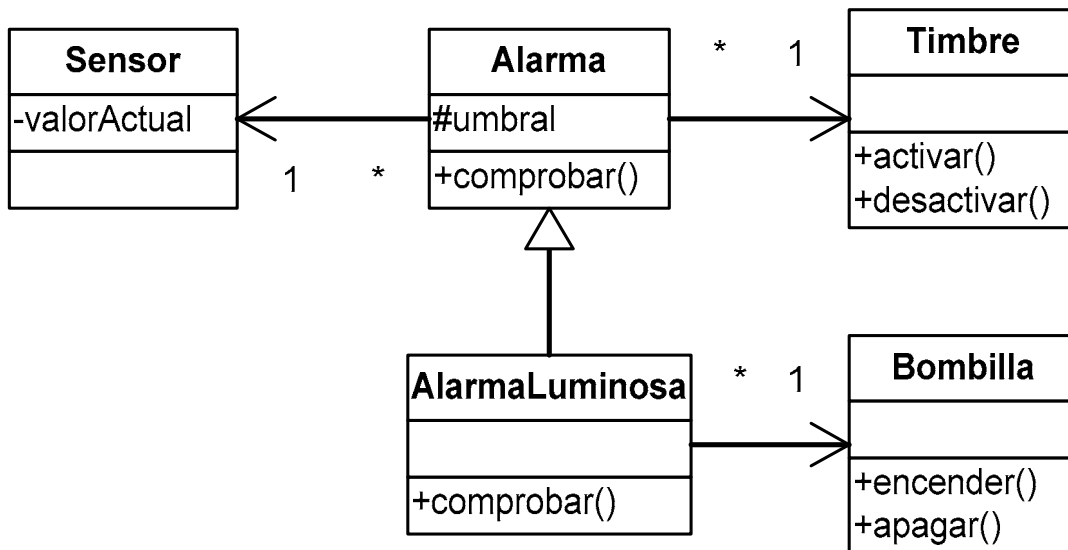
    public static void main(String[] args)
    {
        SensorSwing sensor = new SensorSwing();
        Timbre timbre = new Timbre();
        Alarma alarma = new Alarma (sensor, timbre, 0.0);

        sensor.setAlarma(alarma);
    }
}
```

- + SensorSwing es un tipo particular de Sensor que se encarga de llamar al método comprobar() de la alarma en cuanto se produce un cambio en el valor medido por el sensor.



Versión 1: Redefinición del método comprobar()



```
public class Alarma
{
    protected Sensor sensor;
    protected Timbre timbre;
    protected double umbral;

    /**
     * Constructor
     */

    public Alarma
        (Sensor sensor, Timbre timbre, double umbral)
    ...

    /**
     * Comprobar estado de la alarma
     */

    public void comprobar ()
    {
        if (sensor.getValorActual() > umbral) {
            timbre.activar();
        } else {
            timbre.desactivar();
        }
    }
}
```

```

/**
 * Alarma luminosa (versión 1)
 */

public class AlarmaLuminosa extends Alarma
{
    private Bombilla bombilla;

    /**
     * Constructor
     */

    public AlarmaLuminosa
        ( Sensor sensor, Timbre timbre,
          Bombilla bombilla, double umbral)
    {
        super(sensor,timbre,umbral);

        this.bombilla = bombilla;
    }

    /**
     * Redefinición del método comprobar()
     */

    public void comprobar ()
    {
        super.comprobar();

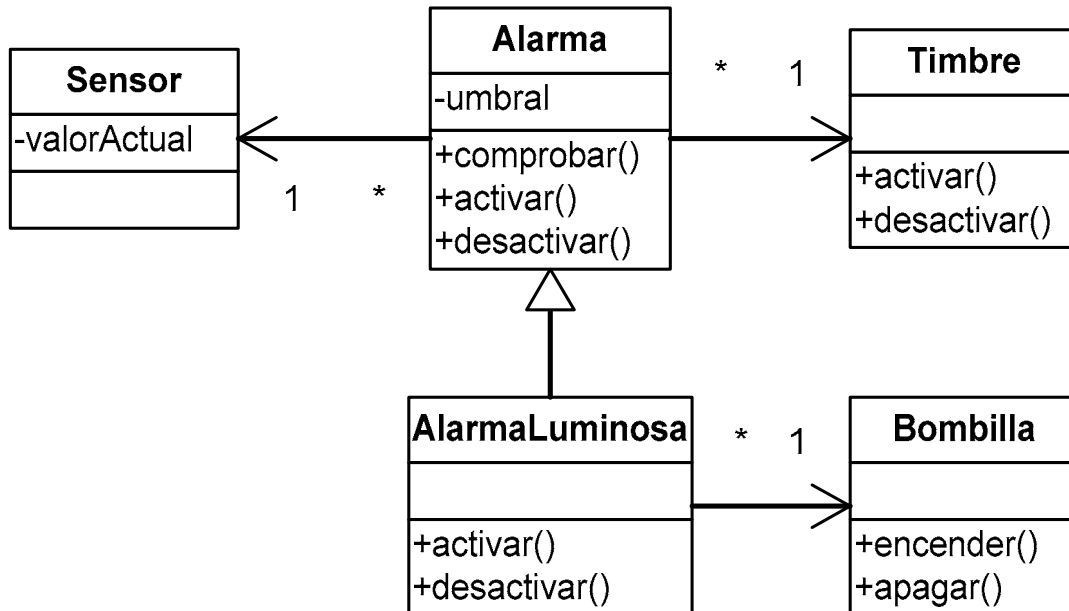
        if (sensor.getValorActual()>umbral) {
            bombilla.encender();
        } else {
            bombilla.apagar();
        }
    }
}

```

- ✖ Acceso a miembros protegidos
("se rompe" la encapsulación de los objetos).
- ✖ Existencia de código duplicado (if...)

NOTA: Hemos de acordarnos de llamar al método `comprobar()` de la clase `Alarma` para que también suene el timbre.

Versión 2: Uso correcto de la redefinición de métodos



La redefinición de los métodos `activar()` & `desactivar()` nos permitirá:

- ✚ Reutilizar la lógica incluida en el método `comprobar()` sin tener que duplicar código (la clase **Alarma** es la única responsable de la activación de la alarma, hecho que conllevará distintas consecuencias en función del tipo particular de alarma).
- ✚ Eliminar una posible fuente de errores (anteriormente, si cambiasen las condiciones bajo las cuales se activa una **Alarma**, también tendríamos que revisar la implementación de `comprobar()` en la clase **AlarmaLuminosa**).
- ✚ Mantener la encapsulación de los objetos de la clase **Alarma** (no se podrá acceder a sus variables de instancia desde el exterior de la clase)

```

/**
 * Alarma
 */

public class Alarma
{
    private Sensor sensor;
    private Timbre timbre;
    private double umbral;

    /**
     * Constructor
     */

    public Alarma
        (Sensor sensor, Timbre timbre, double umbral)
    {
        this.sensor = sensor;
        this.timbre = timbre;
        this.umbral = umbral;
    }

    /**
     * Comprobar estado de la alarma
     */

    public final void comprobar ()
    {
        if (sensor.getValorActual()>umbral) {
            activar();
        } else {
            desactivar();
        }
    }

    public void activar ()
    {
        timbre.activar();
    }

    public void desactivar ()
    {
        timbre.desactivar();
    }
}

```

```

/**
 * Alarma luminosa (versión 2)
 */

public class AlarmaLuminosa extends Alarma
{
    private Bombilla bombilla;

    /**
     * Constructor
     */

    public AlarmaLuminosa
        ( Sensor sensor, Timbre timbre,
          Bombilla bombilla, double umbral)
    {
        super(sensor,timbre,umbral);
        this.bombilla = bombilla;
    }

    /**
     * Redefinición del método activar()
     */

    public void activar ()
    {
        super.activar();
        bombilla.encender();
    }

    /**
     * Redefinición del método desactivar()
     */

    public void desactivar ()
    {
        super.desactivar();
        bombilla.apagar();
    }
}

```

NOTA: Hemos de acordarnos de llamar a los métodos correspondientes de la clase Alarma para que el timbre siga sonando...

Apéndice: Clases auxiliares

Una posible implementación de sensores, timbres y bombillas

Sensor.java

```
/**
 * Clase genérica para representar un sensor
 */

public class Sensor
{
    /**
     * Valor actual medido por el sensor
     */

    private double valorActual;

    /**
     * Acceso al valor medido por el sensor
     */

    public double getValorActual ()
    {
        return valorActual;
    }

    /**
     * Modificación del valor del sensor
     * (algo de lo que se encargarán
     * las subclases particulares de Sensor)
     */

    protected void setValorActual (double valor)
    {
        valorActual = valor;
    }
}
```

SensorSwing.java

```
/**
 * Sensor "ficticio"
 * (demostración del uso de SWING)
 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SensorSwing extends Sensor
    implements ActionListener
{
    private JFrame      frame;
    private JPanel      panel;
    private JButton      button;
    private JTextField editor;
    private JLabel      info;

    private Alarma      alarma;

    /**
     * Constructor
     */
    public SensorSwing()
    {
        // Ventana (JFrame)

        frame = new JFrame("Medidor de... ");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300,100);

        // Panel (JPanel)
        // ...donde se colocan los controles de la interfaz
        panel = new JPanel(new GridLayout(3, 1));

        // Controles de la interfaz de usuario
        // -----
        editor = new JTextField(5);
        info   = new JLabel("...", SwingConstants.CENTER);
        button = new JButton("Actualizar");
    }
}
```

```

        // Manejador de eventos asociado al botón
        button.addActionListener(this);

        // Colocar los controles en el panel
        panel.add(editor);
        panel.add(button);
        panel.add(info);

        // Botón por defecto
        frame.getRootPane().setDefaultButton(button);

        // Añadir el panel a la ventana
        frame.getContentPane().add(panel);

        // Mostrar la ventana
        frame.setVisible(true);
    }

    /**
     * Alarma asociada
     */

    public void setAlarma (Alarma alarma)
    {
        this.alarma = alarma;
    }

    /**
     * Pulsación del botón.
     */

    public void actionPerformed(ActionEvent event)
    {
        double valor = Double.parseDouble(editor.getText());

        setValorActual(valor);
        info.setText("Valor actual = " + valor);

        if (alarma!=null) {
            alarma.comprobar();
        }
    }
}

```

Timbre.java

```
import java.awt.Toolkit;

/**
 * Timbre
 */

public class Timbre
{
    private static final int PITIDOS = 5;

    /**
     * Activar el timbre
     */

    public void activar ()
    {
        int i;

        for (i=0; i<PITIDOS; i++) {
            Toolkit.getDefaultToolkit().beep();

            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
            }
        }
    }

    /**
     * Desactivar el timbre
     */

    public void desactivar ()
    {
    }
}
```

Bombilla.java

Una ventana roja o verde en función del estado de la bombilla...

```
import java.awt.*;
import javax.swing.*;

/**
 * Bombilla
 */

public class Bombilla
{
    private JFrame frame;

    /**
     * Constructor
     */

    public Bombilla()
    {
        frame = new JFrame("Bombilla");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200,200);
        frame.setVisible( true );

        apagar();
    }

    /**
     * Encender la bombilla
     */

    public void encender ()
    {
        frame.getContentPane().setBackground( Color.red );
    }

    /**
     * Apagar la bombilla
     */

    public void apagar ()
    {
        frame.getContentPane().setBackground( Color.green );
    }
}
```