Módulo Programación.

1º DAW.

PRÁCTICA 0: Patrón observer. Eventos en Java.
UNIDAD DE TRABAJO 8.
Profesor: Pedro Antonio Santiago Santiago.

1. Introducción.

El patrón observer es ampliamente utilizado en la creación de software, sirviendo incluso de base para la programación reactiva, actualmente utilizada en los frameworks más conocidos como React.

Ya sea de forma nativa en el lenguaje o con librerías externas la implementación de este patrón no se realiza desde 0, sino que se utiliza implementaciones ya realizadas.

En esta práctica se hace uso del sistema de gestión de eventos que implementa JavaFX, en otros lenguajes es similar, y los fundamentos los mismos.

Se tiene un sistema domótico con diferentes elementos que se pueden interconectar entre si:

Ligth	Se puede encender y apagar por parte de terceros.
Switch	Interruptor, al pulsar se debe de realizar una acción en algún otro dispositivo, por ejemplo encender una luz o un altavoz.
Dimmer	Posee un nivel máximo y mínimo, cada vez que se modifica emite un evento con el nuevo nivel.
Speker	Altavoz, posee un nivel de sonido que se modifica con eventos, de igual forma su encendido y apagado.

2. Desarrollo de la práctica.

2.1. Parte 1. Definiendo el switch.

Definir un interruptor sencillo es fácil, simplemente ha de tener un atributo que indique si se encuentra pulsado o no, en este caso se utiliza un enumerado, aunque se puede usar un booleano:

```
public class Switch {

    private EBooleanState state;
    public Switch() {
    }
    public EBooleanState getState() {
        return state;
```

```
    }

    public void setState(EBooleanState state) {

        this.state = state;

    }
}
```

El enumerado:

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

Pero esta clase no puede “avisar” a nadie de sus cambios, es decir producir evento. Además se desea que todo dispositivo puede estar encendido o apagado, definiendose una interfaz para todos los dispositivos:

```
public interface IDevice {

    public void Start();


    public void Stop();

    public DeviceState getDeviceState();
}
```

Los estados de un dispositivo son un enumerado:

```
public enum DeviceState {
    Stopped,
    Running,
    Error

}
```

También se quiere tener diferentes tipos de dispositivos que puedan emitir interrupciones de tipo on/off, cierto/falso..., por lo que se define una interfaz comun para todos ellos:

```
public interface IDeviceBoolean extends IDevice {
     public EBooleanState getState();
```

```
    public void setState(EBooleanState state);
}
```

Por lo que se hace que el interruptor implemente esta interfaz:

```
public class Switch implements IDeviceBoolean {

    private EBooleanState state;
    private DeviceState deviceState;

    public Switch() {

    }

    @Override
    public EBooleanState getState() {
        return state;
    }

    @Override
    public void setState(EBooleanState state) {
        this.state = state;
    }

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

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

    }

    @Override
    public void Start() {
        this.deviceState = DeviceState.Stopped;
    }
}
```

Aún no puede emitir eventos, pero pronto podrá. Lo primero es definir **quién** puede recibir eventos del interruptor, recuriendo de nuevo a una interfaz, **que los que quieran escuchar interruptores habrán de implementar**, normalmente estas interfaces terminan con Listener para indicar que son escuchadores:

```
public interface ISwitchListener extends java.util.EventListener {
    public void onSwitchChange(SwitchEvent event);
}
```

Posee un único método que recibe un evento de tipo SwitchEvent (definido para esta aplicación):

```
public class SwitchEvent extends EventObject {

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

Hereda de EventObject, y recibe como parámetro el origien del evento, en este caso será un switch.

Por último se añade al interrupotor un **Vector de** escuchadores, métodos para añadir y eliminar escuchadores y cuando se produce un cambio en su estado se recorre el vector de escuchadores avisando del cambio:

```
public class Switch implements IDeviceBoolean {

    private EBooleanState state;
    private DeviceState deviceState;
    //listado de escuchadores
    private Vector<ISwitchListener> switchlisteners;

    public Switch() {
        this.switchlisteners = new Vector<>();
    }
    public void addChangeStateListener(ISwitchListener l) {
        this.switchlisteners.add(l);
    }

    public void removeChangeStateListener(ISwitchListener l) {
        this.switchlisteners.remove(l);
```

```
    }

    public EBooleanState getState() {
        return state;
    }

    //cuando se produce el cambio de estado se emite el evento
    public void setState(EBooleanState state) {
        boolean sendevent = false;
        if (state != this.state) {
            sendevent = true;
        }
        this.state = state;
        if (sendevent) {
            //avisar
            onChangeState();
        }
    }
    //avisar a los escuchadores
    private void onChangeState() {
        //se crea el evento
        SwitchEvent event = new SwitchEvent(this);
        //se recorre los avisadores
        this.switchlisteners.forEach(sl -> sl.onSwitchChange(event));
    }

    @Override
    public void Start() {
        this.deviceState = DeviceState.Running;
    }

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

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

```
    }
}
```

2.2. Parte 2. Interfaz gráfica del interruptor.

Para probar el funcionamiento de los eventos se crea una interfaz gráfica sencilla en JavaFX usando un grid en el que se colocarán los diferentes elementos.

Se envuelve el interruptor en un SwitchComponent que hereda de ImageView (similar a img de html) y se inserta en un grid. El Switch component posee internamente un Switch. El código:

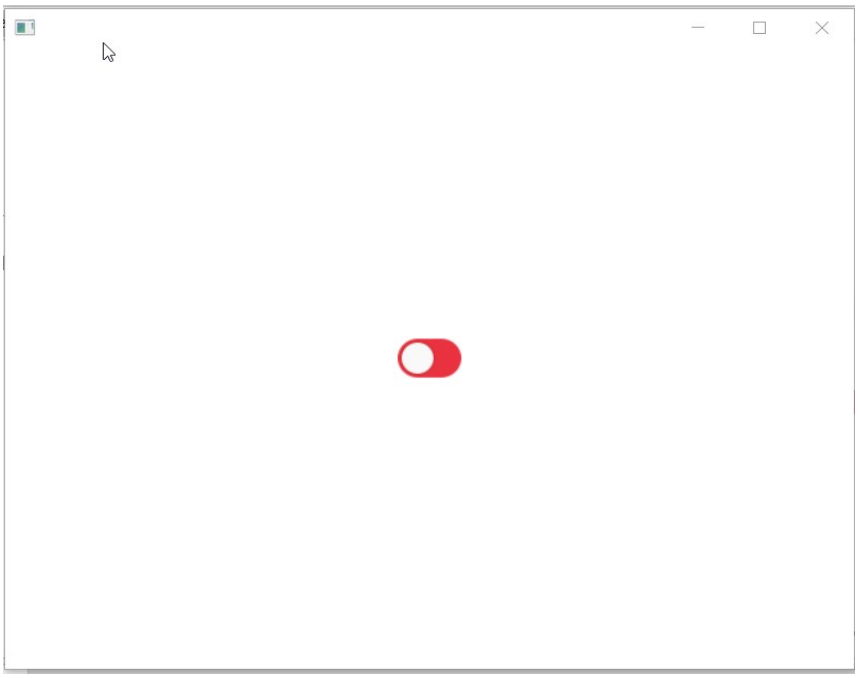
```
public class SwitchComponent extends ImageView {

    private Switch sswitch;

    public SwitchComponent() {
        super();
        this.sswitch = new Switch();
        this.setImage(Resources.getInstance().getImage("switch_off"));
        //evento de la imagen al pulsar, cambia la imagen y se cambia el estado
        //del switch interno
        this.setOnMousePressed((t) -> {
            if (this.sswitch.getState() == EBooleanState.ON) {
                this.sswitch.setState(EBooleanState.OFF);
                this.setImage(Resources.getInstance().getImage("switch_off"));
            } else {
                this.sswitch.setState(EBooleanState.ON);
                this.setImage(Resources.getInstance().getImage("switch_on"));

            }
        });
    }
    public Switch getSwitch() {
        return this.sswitch;
    }
}
```

Como se puede comprobar se están aplicando los conocimientos adquiridos durante todo el curso.

Ahora en la clase principal se añade un grid y un SwiftComponent, además de una lista de dispositivos que será necesario, por ejemplo para iniciarlos:

```
public class App extends Application {

    private GridPane grid;
    private ArrayList<IDevice> devices;
    @Override
    public void start(Stage stage) {
        var javaVersion = SystemInfo.javaVersion();
        var javafxVersion = SystemInfo.javafxVersion();

        this.devices= new ArrayList<>();
        this.createGridView();
        var scene = new Scene(this.grid, 640, 480);
        stage.setScene(scene);
         stage.setOnCloseRequest(arg0 -> {
            //se paran los hilos
            this.devices.forEach(d->{d.Stop();});
         });
        stage.show();
    }
    public void createGridView() {
        this.grid = new GridPane();
        this.grid.setMinSize(640, 480);
        this.grid.setAlignment(Pos.CENTER);
        var sswitch= new SwitchComponent();
        this.devices.forEach( c-> {c.Start();});
        this.devices.add(sswitch.getSwitch());
        this.grid.add( sswitch, 1, 0);
    }
    public static void main(String[] args) {
        launch();
    }
}
```

El resultado es:



2.3. Parte 3. Escuchador Light.

Ya se tiene un elemento que lanza o “dispara” eventos, ya se puede definir elementos que escuchen eventos de tipo SwitchEvent, por ejemplo, una bombilla.

La clase básica de una que implementa el Idevice si escuchar eventos es:

```
public class Light implements IDevice {

    private EBooleanState state;
    private DeviceState deviceState;

    public Light() {
        this.deviceState = deviceState.Stopped;
        this.state = EBooleanState.OFF;

    }

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

            this.changeDeviceState(deviceState.Running);


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

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

}
```

Ahora se ha de preparar para poder recibir los eventos de un Switch, por tanto ha de implementar la interfaz IswitchListener:

```
public class Light implements ISwitchListener, IDevice{

    private EBooleanState state;
    private DeviceState deviceState;

    public Light() {
        this.deviceState = deviceState.Stopped;
        this.state = EBooleanState.OFF;

    }

    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;
    }
    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);

        }
    }

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

    }

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

    @Override
    public void onSwitchChange(SwitchEvent event) {
          if (((IDeviceBoolean) event.getSource()).getState() ==
EBooleanState.OFF) {
            this.setOff();

        } else {
            this.setOn();
        }
     }
}
```

Observar el método onSwitchChange que recibie un SwitchEvent y obtiene el “source” y el estado de este.

2.4. Parte 4. Interfaz gráfica de Light.

Similar a la del pulsador, se envuelve en un ImageView:

```
1. public class LightComponent extends ImageView{
2.     private Light ligth;
3.
4.     public LightComponent(){
5.         super();
```

```
6.          this.ligth= new Light();
7.          this.ligth.setLc(this);
8.          this.setImage(Resources.getInstance().getImage("light_off"));
9.
10.           }
11.           public void setOn(){
12.
   this.setImage(Resources.getInstance().getImage("light_on"));
13.
14.           }
15.           public void setOff(){
16.
   this.setImage(Resources.getInstance().getImage("light_off"));
17.
18.           }
19.           public Light getLight(){
20.               return this.ligth;
21.           }
22.
23.       }
```

Aunque no es del todo correcto, se le pasa a la bombilla, la representación gráfica para que cambie (línea 7), añadiendo los getters y setters en “Ligth y cambiando el setOn y setOff para que modificar el valor de la bombilla gráfica.

```
    public synchronized void setOn() {
        this.state = EBooleanState.ON;
        if(this.lc!=null){
            this.lc.setOn();
        }
        System.out.println("Se ha encendido la luz");
    }

    public synchronized void setOff() {
        this.state = EBooleanState.OFF;
        if(this.lc!=null){
            this.lc.setOff();
        }
        System.out.println("Se ha apagado la luz");
    }
```

2.5. Parte 5. Escuchando.

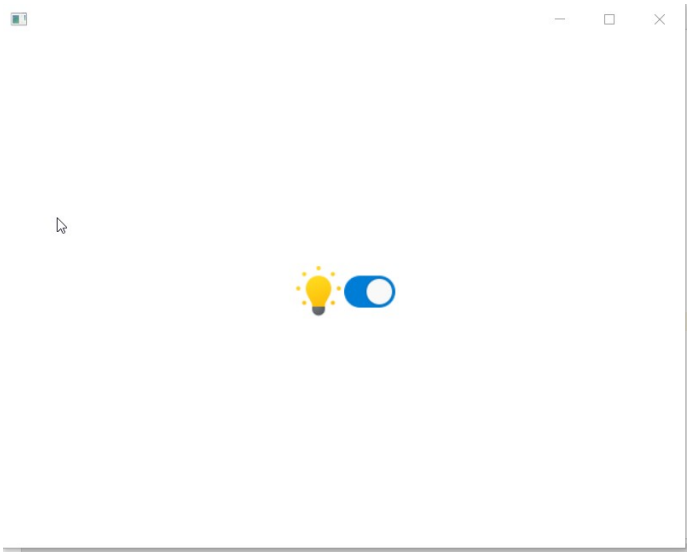
Ahora se ha de “unir” el interruptor con la bombilla, aunque una bombilla puede estar escuchando a varios interruptores y un interruptor controlar varias bombillas.

El el programa principal, se crea la bombilla y se añade como escuchador del interruptor anterior:

```
public class App extends Application {

    private GridPane grid;
    private ArrayList<IDevice> devices;
    @Override
    public void start(Stage stage) {
        var javaVersion = SystemInfo.javaVersion();
        var javafxVersion = SystemInfo.javafxVersion();

        this.devices= new ArrayList<>();
        this.createGridView();
        var scene = new Scene(this.grid, 640, 480);
        stage.setScene(scene);
         stage.setOnCloseRequest(arg0 -> {

         });
        stage.show();
    }

    public void createGridView() {
        this.grid = new GridPane();
        this.grid.setMinSize(640, 480);
        this.grid.setAlignment(Pos.CENTER);
        var ligth = new LightComponent();
        var sswitch= new SwitchComponent();
        sswitch.getSwitch().addChangeStateListener(ligth.getLight());
        this.devices.add(ligth.getLight());
        this.devices.add(sswitch.getSwitch());
        this.devices.forEach( c-> {c.Start();});
        this.grid.add(ligth, 0, 0);
        this.grid.add( sswitch, 1, 0);
    }
```

```
    public static void main(String[] args) {
        launch();
    }

}
```

El resultado:



2.6. Parte 6. Probando los eventos.

Añadir varios interruptores y varias bombillas que se enciendan y apagen, por ejemplo un interruptor que controle varias bombillas y una bombilla controlada por varios interruptores.

2.7. Parte 7. Ampliando los elementos.

- Diseñar un dimmer, que contenga un valor máximo y un valor mínimo, al modificarse el valor ha de saltar un evento. Usar el componente gráfico slider. Es necesario:
  - Definir el dimmer.
  - Definir la interfaz escuchadora del dimmer.
  - Definir el evento que recibiran los escuchadores
- Crear un reproductor de música que deberá poder subscribirse a un interruptor para encenderse y apagarse, junto con la subscripción al dimmer anterior para controlar el volumen del sonido. Ha de implementar la interfaz de SwitchListener y DimmerListener. No es necesario tener un componente gráfico para que funcione.

El esqueleto de la clase Player, en la que se tiene que añadir las interfaces escuchadoras:

```
public class Player implements IDevice {
    private EBooleanState state;
    private DeviceState deviceState;
    private MediaPlayer mediaplayer;
    public Player() {
        this.deviceState = deviceState.Stopped;
        this.state = EBooleanState.OFF;
        this.mediaplayer = Resources.getInstance().getSound("sonido");
        //entrar en bucle, evento que salta al terminar
        this.mediaplayer.setOnEndOfMedia(new Runnable() {
            @Override
            public void run() {
                mediaplayer.seek(Duration.ZERO);
                mediaplayer.play();
            }
        });

    }
    public synchronized void setOn() {
        this.state = EBooleanState.ON;
        this.mediaplayer.play();
        System.out.println("Se ha encendido el reproductor");
    }

    public synchronized void setOff() {
        this.state = EBooleanState.OFF;
        this.mediaplayer.stop();
        System.out.println("Se ha apagado el reproductor");
    }

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

    @Override
    public void Start() {

        this.changeDeviceState(deviceState.Running);

    }
```

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

    }

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

El resultado: