

Patrones de diseño – IS Avanzada

Hugo Ávalos de Rorthais

<https://github.com/hugoavalos01/Patrones2-Triestables>

1. Biestable

En esta ocasión vamos a usar el patrón “Estado” ya que la clase principal se puede encontrar en dos situaciones diferentes (Verde o Rojo), y según la situación se comporta de una manera u otra.

He creado entonces la clase Semáforo, una interfaz Estado con los métodos de la clase semáforo y dos clases que implementan dicha interfaz, Verde y Rojo, que implementarán los métodos abrir(), cerrar() y estado() de manera distinta.

Por último tenemos TestSemáforo(), que lo podemos ejecutar como prueba de JUnit para comprobar el funcionamiento de nuestro semáforo.

Clase Semáforo():

```
public class Semáforo {  
    private static Estado estado;  
    public Semáforo(Estado e) {  
        estado = e;  
    }  
    public void abrir() {  
        estado.abrir();  
    }  
    public void cerrar() {  
        estado.cerrar();  
    }  
    public static void setEstado(Estado e) {  
        estado = e;  
    }  
    public String estado() {  
        return estado.estado();  
    }  
}
```

Clase Estado():

```
public interface Estado {  
    public String estado();  
    public void abrir();  
    public void cerrar();  
}
```

Clase Verde() y Rojo():

```
public class Rojo implements Estado {  
    public Rojo() {  
    }  
    @Override  
    public String estado() {  
        // TODO Auto-generated method stub  
        return "cerrado";  
    }  
    @Override  
    public void abrir() {  
        // TODO Auto-generated method stub  
        Semaforo.setEstado(new Verde());  
    }  
    @Override  
    public void cerrar() {  
        // TODO Auto-generated method stub  
    }  
}
```

```
public class Verde implements Estado {  
    public Verde() {  
    }  
    @Override  
    public String estado() {  
        // TODO Auto-generated method stub  
        return "abierto";  
    }  
    @Override  
    public void abrir() {  
        // TODO Auto-generated method stub  
    }  
    @Override  
    public void cerrar() {  
        // TODO Auto-generated method stub  
        Semaforo.setEstado(new Rojo());  
    }  
}
```

TestSemaforo():

Runs: 2/2 Errors: 0 Failures: 0

TestSemaforo [Runner: JUnit 5] (0,001 s)
testCambio() (0,001 s)
testConstructor() (0,000 s)

Failure Trace

```
3 import static org.junit.jupiter.api.Assertions.assertEquals;  
7  
8 public class TestSemaforo {  
9  
10     private Semaforo s;  
11  
12     // Antes de cada test creamos un nuevo semaforo y en verde  
13     @BeforeEach  
14     void init() {  
15  
16         s = new Semaforo(new Verde());  
17     }  
18  
19     // Comprobamos el constructor  
20     @Test  
21     void testConstructor() {  
22  
23         assertEquals("abierto", s.estado());  
24         assertEquals("cerrado", new Semaforo(new Rojo()).estado());  
25     }  
26  
27     // Comprobamos que el semaforo se abre y se cierra  
28     @Test  
29     void testCambio() {  
30  
31         s.cerrar();  
32         assertEquals("cerrado", s.estado());  
33  
34         s.abrir();  
35         assertEquals("abierto", s.estado());  
36     }  
37 }  
38
```

2. Triestable

Ahora se nos pide añadir un estado más al semáforo, el Amarillo.

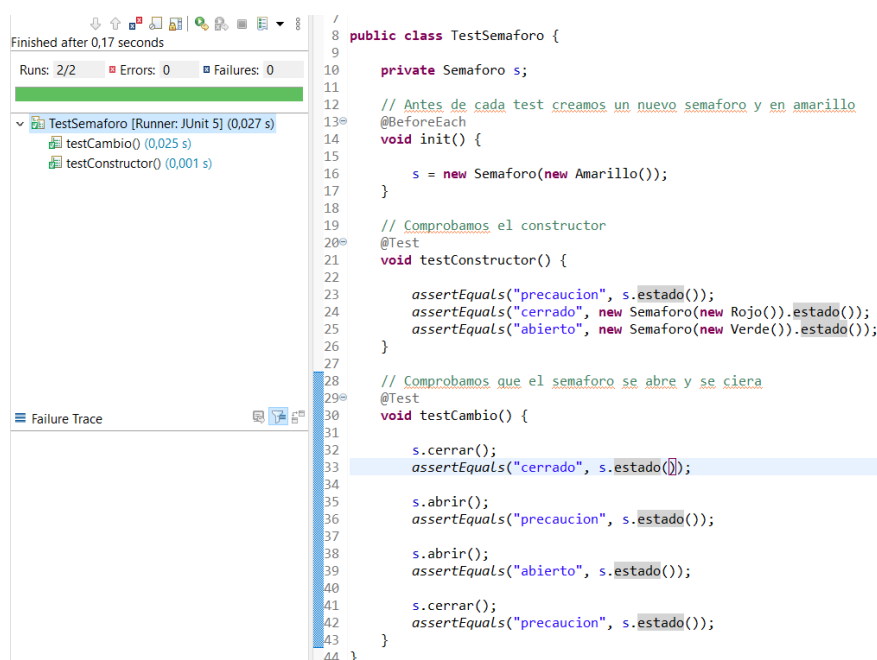
Como anteriormente habíamos aplicado el patrón Estado, ahora simplemente tendríamos que añadir una clase “Amarillo()” que implemente la interfaz “Estado” y definir sus métodos. De esta manera la reutilización del código es casi completa y las modificaciones son mínimas. El único cambio en el código sería en los métodos abrir() y cerrar() de Rojo() y Verde().

Por último, hemos creado la clase TestSemaforo() para comprobar el correcto funcionamiento del proyecto.

Clase Amarillo():

```
public class Amarillo implements Estado {  
  
    public Amarillo() {  
  
    }  
  
    @Override  
    public String estado() {  
        // TODO Auto-generated method stub  
        return "precaucion";  
    }  
  
    @Override  
    public void abrir() {  
        // TODO Auto-generated method stub  
        Semaforo.setEstado(new Verde());  
    }  
  
    @Override  
    public void cerrar() {  
        // TODO Auto-generated method stub  
        Semaforo.setEstado(new Rojo());  
    }  
  
}
```

Clase TestSemaforo():



The screenshot shows an IDE with a test runner on the left and source code on the right. The test runner shows 'Finished after 0,17 seconds', 'Runs: 2/2', 'Errors: 0', and 'Failures: 0'. It lists two tests: 'testCambio()' (0,025 s) and 'testConstructor()' (0,001 s). The source code on the right is for 'TestSemaforo' and includes comments in Spanish explaining the test logic.

```
8 public class TestSemaforo {  
9  
10     private Semaforo s;  
11  
12     // Antes de cada test creamos un nuevo semaforo y en amarillo  
13     @BeforeEach  
14     void init() {  
15  
16         s = new Semaforo(new Amarillo());  
17     }  
18  
19     // Comprobamos el constructor  
20     @Test  
21     void testConstructor() {  
22  
23         assertEquals("precaucion", s.estado());  
24         assertEquals("cerrado", new Semaforo(new Rojo()).estado());  
25         assertEquals("abierto", new Semaforo(new Verde()).estado());  
26     }  
27  
28     // Comprobamos que el semaforo se abre y se cierra  
29     @Test  
30     void testCambio() {  
31  
32         s.cerrar();  
33         assertEquals("cerrado", s.estado());  
34  
35         s.abrir();  
36         assertEquals("precaucion", s.estado());  
37  
38         s.abrir();  
39         assertEquals("abierto", s.estado());  
40  
41         s.cerrar();  
42         assertEquals("precaucion", s.estado());  
43     }  
44 }
```

3. Biestable y Triestable

Se nos pide implementar un método “cambio()” para pasar de biestable a triestable.

Para el modo biestable he creado dos clases nuevas, “Rojo_Bi” y “Verde_Bi”, que actúan igual que las clases “Rojo” y “Verde” del semáforo biestable que desarrollamos previamente.

Para el modo triestable, el código es igual que el del apartado anterior, no hemos tenido que cambiar nada.

Por último, he añadido el método “**cambio()**” a la interfaz y a las diferentes clases. Este método cambia entre “Rojo” y “Rojo_Bi” y entre “Verde” y “Verde_Bi”.

Cuando se invoque a este método y el semáforo esté en Amarillo, este pasará a un estado auxiliar, “**Transición**”, que devuelve el mismo mensaje que Amarillo sólo que al invocar a los métodos abrir o cerrar, pasará a “Verde_Bi” o “Rojo_Bi”, cambiando así al modo biestable. Con este estado auxiliar nos ahorramos tener que tomar una decisión de si poner el semáforo Rojo o Verde cuando hagamos el cambio desde el color Amarillo.

Gracias a esto, la reutilización del código es completa y sólo hemos tenido que añadir el método cambio() a nuestro proyecto. Podríamos decir que la interfaz Estado actúa también similar a un patrón “Estrategia”, cambiando de una a otra al invocar a cambio().

Por último, he desarrollado TestSemaforo() para comprobar el funcionamiento del proyecto.

Clase Rojo_Bi / Verde_Bi:

```
public class Rojo_Bi implements Estado {  
    public Rojo_Bi() {  
    }  
    @Override  
    public String estado() {  
        // TODO Auto-generated method stub  
        return "cerrado";  
    }  
    @Override  
    public void abrir() {  
        // TODO Auto-generated method stub  
        Semaforo.setEstado(new Verde_Bi());  
    }  
    @Override  
    public void cerrar() {  
        // TODO Auto-generated method stub  
    }  
    @Override  
    public void cambio() {  
        // TODO Auto-generated method stub  
        Semaforo.setEstado(new Rojo());  
    }  
}
```

Clase Transicion():

```
public class Transicion implements Estado{

    public Transicion() {

    }

    @Override
    public String estado() {
        // TODO Auto-generated method stub
        return "precaucion";
    }

    @Override
    public void abrir() {
        // TODO Auto-generated method stub
        Semaforo.setEstado(new Verde_Bi());
    }

    @Override
    public void cerrar() {
        // TODO Auto-generated method stub
        Semaforo.setEstado(new Rojo_Bi());
    }

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

    }

}
```

Clase Amarillo():

```
public class Amarillo implements Estado {

    public Amarillo() {

    }

    @Override
    public String estado() {
        // TODO Auto-generated method stub
        return "precaucion";
    }

    @Override
    public void abrir() {
        // TODO Auto-generated method stub
        Semaforo.setEstado(new Verde());
    }



    @Override
    public void cerrar() {
        // TODO Auto-generated method stub
        Semaforo.setEstado(new Rojo());
    }

    @Override
    public void cambio() {
        // TODO Auto-generated method stub
        Semaforo.setEstado(new Transicion());
    }


}
```


Clase TestSemaforo():


```
public class TestSemaforo {  
  
    private Semaforo s;  
  
    // Antes de cada test creamos un nuevo semaforo y en amarillo  
    @BeforeEach  
    void init() {  
  
        s = new Semaforo(new Amarillo());  
    }  
  
    // Comprobamos el constructor  
    @Test  
    void testConstructor() {  
  
        assertEquals("precaucion", s.estado());  
        assertEquals("cerrado", new Semaforo(new Rojo()).estado());  
        assertEquals("abierto", new Semaforo(new Verde()).estado());  
  
    }  
  
    // Comprobamos que el semaforo se abre y se cierra  
    @Test  
    void testAbreCierra() {  
  
        s.cerrar();  
        assertEquals("cerrado", s.estado());  
  
        s.abrir();  
        assertEquals("precaucion", s.estado());  
  
        s.abrir();  
        assertEquals("abierto", s.estado());  
  
    }  
  
    //Comprobamos que cambia correctamente de modo desde cualquier estado posible  
    @Test  
    void testCambio() {  
  
        //De amarillo a verde biestable al abrir  
        s.cambio();  
        s.abrir();  
        assertEquals("abierto", s.estado());  
  
        //De verde biestable a verde triestable  
        s.cambio();  
        assertEquals("abierto", s.estado());  
  
        //De rojo triestable a rojo biestable  
        s.cerrar();  
        s.cerrar();  
        s.cambio();  
        assertEquals("cerrado", s.estado());  
  
        //De rojo biestable a triestable  
        s.cambio();  
        assertEquals("cerrado", s.estado());  
  
    }  
}
```

Runs: 3/3  Errors: 0  Failures: 0

TestSemaforo [Runner: JUnit 5] (0,038 s)

 testAbreCierra() (0,032 s)

 testCambio() (0,003 s)

 testConstructor() (0,001 s)