Triestables

INFORME PRÁCTICA 3

Nerea martín Serrano | 3º Ingeniería de la Salud

Índice

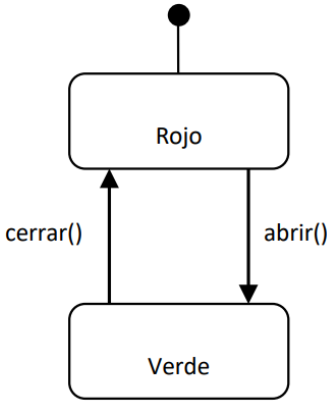
Introducción

En esta práctica vamos a implementar varios dispositivos software similares un semáforo: un biestable y un triestable, y al final de un dispositivo que pueda cambiar de biestable a triestable y viceversa.

Este proyecto esta alojado en el siguiente repositorio de GitHub: https://github.com/nmartinse/Triestable

Biestable

El dispositivo biestable seguirá el siguiente esquema:



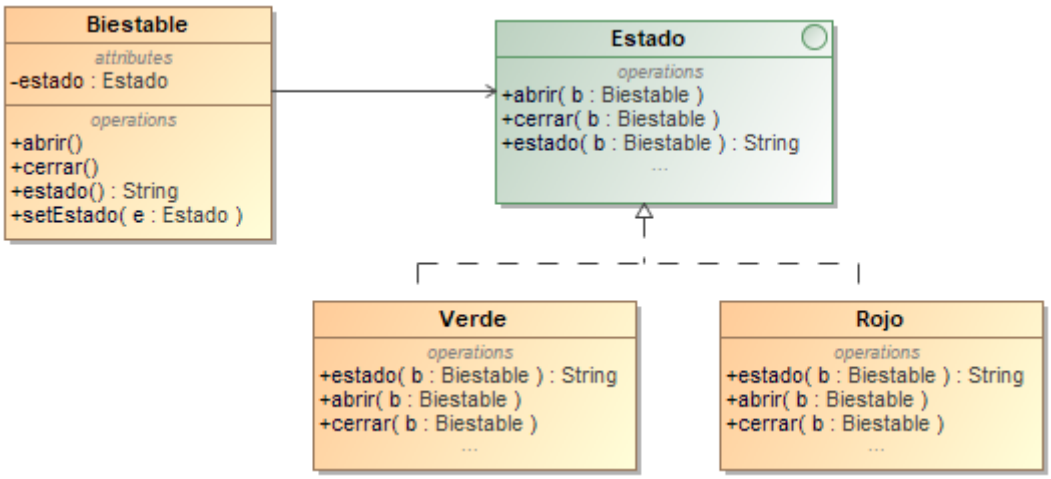
Tenemos dos estados: Rojo y Verde. El método *cerrar()* pasa del estado Verde al Rojo y el método *abrir()* del Rojo al Verde. Si aplicamos *abrir()* al estado Verde, no tendrá ningún efecto, al igual que si aplicamos *cerrar()* sobre Rojo.

Vamos a aplicar el patrón de diseño Estado. Este patrón nos va a permitir que el comportamiento de nuestro objeto cambie según el estado que se encuentre. Además, si en un futuro surge otro estado se puede añadir sin necesidad de hacer grandes modificaciones en el código.

Aunque, este no es el único patrón que se aplica, posteriormente también se usará el patrón Fábrica.

DIAGRAMA DE CLASES

Antes de empezar a diseñar las pruebas o a implementar en Java vamos a crear un diagrama de clases para tener un esquema del que partir.



Vamos a tener una clase que sea el biestable. Después tenemos una interfaz para manejar los distintos estados, que en este caso son rojo y verde.

La clase **Biestable** va a tener un atributo *estado*, que será una instancia de la interfaz ***Estado***. Este atributo va a servir para guardar el estado en el que se encuentra en ese momento el biestable. Como el atributo *estado* es privado, vamos a crear un método para poder modificar es estado, *setEstado()* Además, esta clase conta de otros tres métodos que delega en ***Estado*** (tal y como indica el patrón Estado).

Al principio del informe hemos dicho que se iba a usar el patrón Estado para resolver el problema de tener un objeto con varios estados. Pero también se ha usado el patrón Fábrica, pues en los métodos de los estados se realizan cambios de estado, por lo tanto delegamos la creación de los objetos a los estados. La clase **Biestable** no crea un constructor para la variable *estado*, sino que se invoca a la interfaz ***Estado***. Entonces, la interfaz ***Estado*** se ha definido tanto para controlar los estados como para crear los objetos que hagan referencia al estado (las clases **Rojo** y **Verde**).

En la interfaz ***Estado*** se define los tres métodos que van a tener en común todos los estados. Estos tres métodos tienen como argumento una instancia de la clase **Biestable**, para poder tener acceso a los métodos de esta clase.

La clase **Rojo** y **Verde** son las clases encargadas de modular los estados. Los dos implementan la interfaz ***Estado***. Por lo tanto, los dos tienen los mismo tres métodos, pero en cada estado tienen una finalidad distinta.

	Verde	**Rojo**
abrir()	No tiene efecto	Cambia al estado verde
cerrar()	Cambia al estado rojo	No tiene efecto
estado()	Devuelve “abierto”	Devuelve “cerrado”

PRUEBAS

Antes de implementar este objeto en Java vamos a crear las pruebas. Para ello vamos a usar *Cucumber*.

Clase Verde

Primero, vamos a hacer las pruebas para la clase **Verde**. Entonces, creamos un escenario para cada uno de los métodos que implementa esta clase. Creamos un *.feature* para que compruebe el correcto funcionamiento de los métodos de esta clase:

```
3  @test_verde
4  Feature: Test Verde
5
6    @test_abrir
7    Scenario: abrirVerdeOk
8      Given el biestable en estado Verde
9      When quiero seleccionar abrir
10     Then no se modifica el estado
11
12   @test_cerrar
13   Scenario: cerrarVerdeOk
14     Given el biestable esta en Verde
15     When quiero seleccionar cerrar
16     Then se modifica el estado a rojo
```

Una vez que tenemos el *.feature* tenemos que implementar el StepDefinition para cada uno de los escenarios

Método abrir()

Ahora debemos de definir el escenario correspondiente a este método en el *StepDefinitions*:

```
12    @Given("el biestable en estado Verde")
13    public void el_biestable_en_estado_verde() {
14        b.setEstado(new Verde());
15    }
16    @When("quiero seleccionar abrir")
17    public void quiero_seleccionar_abrir() {
18        b.abrir();
19    }
20    @Then("no se modifica el estado")
21    public void no_se_modifica_el_estado() {
22        assertEquals("abierto", b.estado());
23    }
```

Siendo b: `private Biestable b = new Biestable();`

En este escenario lo que hacemos es comprobar si estando en el estado Verde e invocamos el método *abrir()*, este no tiene ningún efecto. Es decir, cambia el estado, por lo tanto comprobamos que si después de invocar a este método el estado sigue siendo verde.

Método cerrar()

```
25    @Given("el biestable esta en Verde")
26    public void el_biestable_esta_en_verde() {
27        b.setEstado(new Verde());
28    }
29
30    @When("quiero seleccionar cerrar")
31    public void quiero_seleccionar_cerrar() {
32        b.cerrar();
33    }
34    @Then("se modifica el estado a rojo")
35    public void se_modifica_el_estado_a_rojo() {
36        assertEquals("cerrado", b.estado());
37    }
```

Primero, establecemos que el biestable esté en estado verde. Después, le pasamos el método *cerrar()*. Este método debería de cambiar el estado a rojo, por lo que comprobamos que esto sea cierto.

Clase Rojo

A continuación, vamos a realizar las pruebas para la clase **Rojo**. Lo primero es definir los escenarios;

```
3  @test_rojo
4  Feature: Test Rojo
5
6    @test_abrir
7    Scenario: abrirRojoOk
8      Given el biestable en estado rojo
9      When selecciono abrir
10     Then se modifica el estado a verde
11
12   @test_cerrar
13   Scenario: cerrarRojoOk
14     Given el biestable esta en rojo
15     When slecciono cerrar
16     Then el estado no se modifica
```

Implementamos estos escarrios en el *stepDefinition*. En este caso debemos de comprobar que el biestable esta en el estado Rojo e invocamos al método *abrir()* el biestable cambiará su estado a verde, Sin embargo, si invocamos el método *cerrar()* el estado no se modificará.

```
@Given("el biestable en estado rojo")
public void el_biestable_en_estado_rojo() {
    b.setEstado(new Rojo());
}

@When("selecciono abrir")
public void selecciono_abrir() {
    b.abrir();
}

@Then("se modifica el estado a verde")
public void se_modifica_el_estado_a_verde() {
    assertEquals("abierto", b.estado());
}

@Given("el biestable esta en rojo")
public void el_biestable_esta_en_rojo() {
    b.setEstado(new Rojo());
}

@When("slecciono cerrar")
public void slecciono_cerrar() {
    b.cerrar();
}

@Then("el estado no se modifica")
public void el_estado_no_se_modifica() {
    assertEquals("cerrado", b.estado());
}
```

IMPLEMENTACIÓN EN JAVA

Ahora pasamos a la implementación en Java. Para ello seguimos el esquema que hemos creado al principio.

- **Clase biestable**.

```java
public class Biestable {

    private Estado estado;

    /**
     * Método set, para cambiar de estado
     *
     * @param e: estado al que se desea cambiar
     */
    public void setEstado(Estado e) {
        estado = e;
    }

    public void abrir() {
        estado.abrir(this);
    }

    public void cerrar() {
        estado.cerrar(this);
    }

    public String estado() {
        return estado.estado(this);
    }
}
```

- Interfaz ***Estado***

```java
public interface Estado {

    void abrir(Biestable biestable);
    void cerrar(Biestable biestable);
    String estado(Biestable biestable);
}
```

- Clase **Rojo**

```java
public class Rojo implements Estado {

    public void abrir(Biestable biestable) {
        biestable.setEstado(new Verde());
    }

    public void cerrar(Biestable biestable) {

    }

    public String estado(Biestable biestable) {
        return "cerrado";
    }
}
```

- Clase **Verde**

```java
public class Verde implements Estado {

    public void abrir(Biestable biestable) {

    }

    public void cerrar(Biestable biestable) {
        biestable.setEstado(new Rojo());

    }

    public String estado(Biestable biestable) {
        return "abierto";
    }
}
```

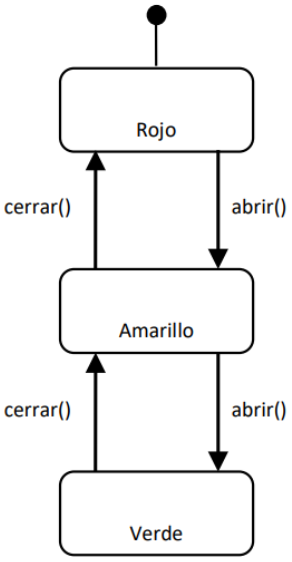
Para comprobar si hemos realizado correctamente esta implementación, ejecutemos los test que hemos creado con anterioridad. La salida de estas pruebas es :

```
Results:

Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

------------------------------------------------
BUILD SUCCESS
------------------------------------------------
```

Esto quiere decir que la implementación ha funcionado como se esperaba.

Triestable

Este dispositivo va a seguir el siguiente esquema:



Como podemos comprar este esquema es muy parecido al dispositivo anterior, el biestable. Pero esta vez se ha añadido un estado intermedio, el amarillo.

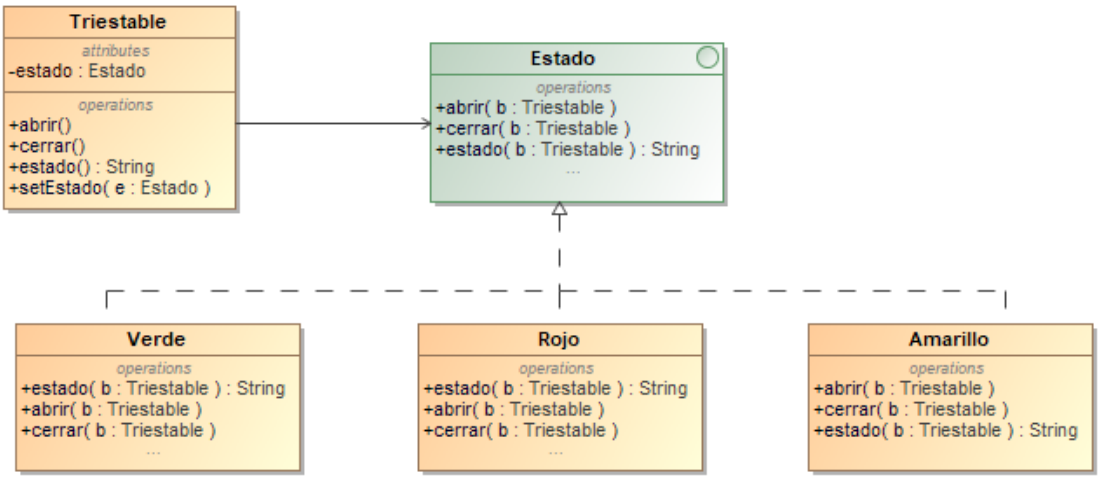
Al haber implementado el patrón Estado al desarrollar el biestable, el añadir un nuevo estado no debería de suponer muchos cambios. En un principio nada más que debemos de crear una nueva case que haga referencia al estado amarillo. Aunque, posteriormente veremos que surgen algunos problemas.

En el triestable se mantienen los patrones aplicados en el anterior dispositivo, el patrón Estrategia y el patrón Fabrica.

Además, estos patrones siguen teniendo la misma finalidad. En el caso del patrón Estado se va a usar para poder modular los distintos estados en lo que se puede encontrar el triestable. El patrón Fábrica se va a usar para poder delegar la ceración de objetos a estos estados (que solo los estados puedan crear otros estados)

DIAGRAMA DE CLASES

Como en el anterior dispositivo vamos a comenzar por crear el diagrama de clase Básicamente debemos de mantener el mismo diagrama, solo tenemos que añadir la nueva clase **Amarillo** que implemente de la interfaz ***Estado***.



Como podemos ver son las mismas clases, el problema es que ahora algunos métodos tienen una finalidad distinta, por ejemplo ahora si aplicamos el método *cerrar()* al triestable en estado verde este pasa a estado amarrillo, y no a rojo como hacía en el biestable.

	Verde	**Amarillo**	**Rojo**
abrir()	No tiene efecto	Cambia al estado verde	Cambia al estado amarillo
cerrar()	Cambia al estado amarillo	Cambia al estado rojo	No tiene efecto
estado()	Devuelve “abierto”	Devuelve “precaución”	Devuelve “cerrado”

IMPLEMENTACIÓN EN JAVA

Esta vez vamos a empezar por la implementación en Java para ir comentando los cambios en el código. Para desarrollar el triestable vamos a reutilizar el código del biestable, es decir vamos a Java y copiamos el proyecto donde hemos desarrollado el biestable.

Empezamos por la clase **Triestable**, para ello cogemos el código de la clase **Biestable** anterior, seleccionamos el nombre de esta clase y le damos a refactorizar, cambiando el nombre de la clase a **Triestable.** Entonces, el código quedaría:

```java
public class Triestable {

    private Estado estado;

    /**
     * Método set, para cambiar de estado
     *
     * @param e: estado al que se desea cambiar
     */
    public void setEstado(Estado e) {
        estado = e;
    }

    public void abrir() {
        estado.abrir(this);
    }

    public void cerrar() {
        estado.cerrar(this);
    }

    public String estado() {
        return estado.estado(this);
    }
}
```

Los métodos de la interfaz ***Estado*** tenían como argumento una instancia de la clase **Biestable**. Al haber cambiado el nombre de la clase con refactorización, los argumentos de estos métodos han cambiado, y ahora son una instancia de la clase **Triestable**. Entonces, el único cambio que vamos a hacer en esta interfaz es cambiar el nombre de estas variables de método, para felicitar la lectura del código.

Por lo tanto, vamos a volver a realizar refactorización, esta vez con el nombre de las variables de método, para cambiarla de *biestable* a *triestable*.

```java
public interface Estado {

    void abrir(Triestable triestable);
    void cerrar(Triestable triestable);
    String estado(Triestable triestable);
}
```

Las clases que más cambios significativos van a ser la clase **Rojo** y **Verde**, debido a que ahora ha aparecido el estado intermedio amarrillo. Por lo tanto, hay que modificar los métodos que provocan un cambio de estado, pues ahora si invocamos al método *cerrar()* en verde ya no hay que cambiar al estado a rojo, sino que hay que cambia a amarillo.

Entonces, primero necesitamos crear ese estado amarillo intermedio. Creamos una clase **Amarillo** que implemente la interfaz ***Estado***.

```java
public class Amarillo implements Estado {

    public void abrir(Triestable triestable) {
        triestable.setEstado(new Verde());
    }

    public void cerrar(Triestable triestable) {
        triestable.setEstado(new Rojo());
    }

    public String estado(Triestable triestable) {
        return "preucacion";
    }
}
```

Por último, quedan hacer los cambios pertinentes en **Rojo** y **Verde**. Como hemos mencionado anteriormente, de estas clases lo que hay que cambiar con lo método que provoquen un cambio de estado. En lo métodos donde estas dos clases cambiar de estado, deben de cambiar al estado amarillo.

```java
public class Verde implements Estado {

    public void abrir(Triestable triestable) {

    }

    public void cerrar(Triestable triestable) {
        triestable.setEstado(new Amarillo());
    }

    public String estado(Triestable triestable) {
        return "abierto";
    }
}
```

```java
public class Rojo implements Estado {

    public void abrir(Triestable triestable) {
        triestable.setEstado(new Amarillo());
    }

    public void cerrar(Triestable triestable) {

    }

    public String estado(Triestable triestable) {
        return "cerrado";
    }
}
```

Se planteado aplicar el patrón Adaptador, pero no ha sido necesario pues no ha habido que crear una nueva interfaz ni modificar significativamente la que ya tenemos (pues solo se ha cambiado el nombre de los argumentos de los métodos).

PRUEBAS

Los miso pasa con las pruebas, el código se puede reciclar es su mayoría. Debemos de crear un nuevo escenario para la clase **Amarillo** y modificar un par de detalles en las clases **Rojo** y **Verde**.

```java
public class StepDefinitions {

    private Triestable t = new Triestable();

    @Given("el triestable en estado Amarillo")
    public void el_triestable_en_estado_amarillo() {
        t.setEstado(new Amarillo());
    }
    @When("invoco al metodo abrir")
    public void invoco_al_metodo_abrir() {
        t.abrir();
    }
    @Then("se modifica el estado a Verde")
    public void se_modifica_el_estado_a_verde() {
        assertEquals("abierto", t.estado());
    }


    @Given("el triestable esta en Amarillo")
    public void el_triestable_esta_en_amarillo() {
        t.setEstado(new Amarillo());
    }
    @When("invoco al metodo cerrar")
    public void invoco_al_metodo_cerrar() {
       t.cerrar();
    }
    @Then("se modifica el estado a Rojo")
    public void se_modifica_el_estado_a_rojo() {
        assertEquals("cerrado", t.estado());
    }
    @Given("el triestable en estado rojo")
    public void el_triestable_en_estado_rojo() {
        t.setEstado(new Rojo());
    }
    @When("selecciono abrir")
    public void selecciono_abrir() {
        t.abrir();
    }
    @Then("se modifica el estado a amarillo")
    public void se_modifica_el_estado_a_amarillo() {
        assertEquals("preucacion", t.estado());
    }


    @Given("el triestable esta en rojo")
    public void el_triestable_esta_en_rojo() {
        t.setEstado(new Rojo());
    }
    @When("slecciono cerrar")
    public void slecciono_cerrar() {
        t.cerrar();
    }
    @Then("el estado no se modifica")
    public void el_estado_no_se_modifica() {
        assertEquals("cerrado", t.estado());
    }
```

```
71      @Given("el triestable en estado Verde")
72      public void el_triestable_en_estado_verde() {
73          t.setEstado(new Verde());
74      }
75      @When("quiero seleccionar abrir")
76      public void quiero_seleccionar_abrir() {
77         t.abrir();
78      }
79      @Then("no se modifica el estado")
80      public void no_se_modifica_el_estado() {
81          assertEquals("abierto", t.estado());
82      }
83
84
85      @Given("el triestable esta en Verde")
86      public void el_triestable_esta_en_verde() {
87         t.setEstado(new Verde());
88      }
89      @When("quiero seleccionar cerrar")
90      public void quiero_seleccionar_cerrar() {
91          t.cerrar();
92      }
93      @Then("se modifica al estado amarillo")
94      public void se_modifica_al_estado_a_amarillo() {
95          assertEquals("preucacion", t.estado());
96      }
97  }
```

Probamos a ejecutar estas pruebas para ver si hemos implementado el triestable correctamente:

```
Results:

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0

------------------------------------------------------
BUILD SUCCESS
```

Todos los test que hemos implementado se han ejecutado perfectamente, por lo tanto, nuestro sistema funciona correctamente.

Sistema intercambiable

Este es un sistema más complejo donde tenemos un biestable que se puede cambiar a Triestable.

IMPLEMENTACIÓN EN JAVA

- Clase principal **Sistema**

```
public class Sistema {

    private Tipo t = new Biestable();

    public void cambio() {
        if (t instanceof Biestable) {
            t = new Triestable();
            t.setEstado(new Amarillo());
        } else
            t = new Biestable();
    }

    public void abrir() {
        t.abrir();
    }

    public void cerrar() {
        t.cerrar();
    }

    public String t() {
        return t.estado();
    }

    public void setEstado(Estado e) {
        t.setEstado(e);

    }

    public String estado() {
        return t.estado();
    }
}
```

- Interfaz ***Tipo***

```
2  public interface Tipo {
3
4      void abrir();
5
6      void cerrar();
7
8      String estado();
9
10     void setEstado(Estado e);
11
12 }
```

- Clase **Biestable**

```
7  public class Biestable implements Tipo {
8
9      private Estado estado;
10
11     public void setEstado(Estado e) {
12         estado = e;
13     }
14
15     public void abrir() {
16         estado.abrir(this);
17     }
18
19     public void cerrar() {
20         estado.cerrar(this);
21     }
22
23     public String estado() {
24         return estado.estado(this);
25     }
26 }
```

- Clase **Triestable**

```
7  public class Triestable implements Tipo {
8
9      private Estado estado;
10
11     public void setEstado(Estado e) {
12         estado = e;
13     }
14
15     public void abrir() {
16         estado.abrir(this);
17     }
18
19     public void cerrar() {
20         estado.cerrar(this);
21     }
22
23     public String estado() {
24         return estado.estado(this);
25     }
26 }
```

- Interfaz **Estado**

```
2  public interface Estado {
3
4      void abrir(Object o);
5
6      void cerrar(Object o);
7
8      String estado(Object o);
9  }
```

- Clase **Rojo**

```
2  public class Rojo implements Estado {
3
4      public void abrir(Object o) {
5          if (o instanceof Triestable)
6              ((Triestable) o).setEstado(new Amarillo());
7          else
8              ((Biestable) o).setEstado(new Verde());
9
10     }
11
12     public void cerrar(Object o) {
13
14     }
15
16     public String estado(Object o) {
17         return "cerrado";
18     }
19 }
```

- Clase **Verde**

```
2  public class Verde implements Estado {
3
4      public void cerrar(Object o) {
5          if (o instanceof Triestable)
6              ((Triestable) o).setEstado(new Amarillo());
7          else
8              ((Biestable) o).setEstado(new Rojo());
9
10     }
11
12     public void abrir(Object o) {
13
14     }
15
16     public String estado(Object o) {
17         return "abierto";
18     }
19 }
```

- Clase **Amarillo**

```
2  public class Amarillo implements Estado {
3
4      public void abrir(Object o) {
5          if (o instanceof Triestable)
6              ((Triestable) o).setEstado(new Verde());
7
8      }
9
10     public void cerrar(Object o) {
11         if (o instanceof Triestable)
12             ((Triestable) o).setEstado(new Rojo());
13
14     }
15
16     public String estado(Object o) {
17         return "preucacion";
18     }
19 }
```

En esta ocasión lo que se ha hecho es tratar a la clase **Biestable** y **Triestable** como otras dos estrategias. Entonces, en este código hay dos patrones estrategias aplicados, uno para los tipos de sistema as que tenemos y otro para los tipos de estado.

Además es destacable que en esta sección no hemos podido reutilizar tanto código como en la sección anterior.