Patrones de Diseño

Ingeniería del Software Avanzada

Jesús Aldana Martín

Práctica 2. Triestables

a) Biestable

Para este primer apartado el ejercicio nos pide implementar un dispositivo biestable con dos estados, rojo y verde. Desde el estado rojo y mediante el método abrir() pasaremos a verde y volveremos a rojo con cerrar(), indicando en todo momento tras cada acción el estado del dispositivo.

Tras leer la práctica al completo he optado por utilizar el patrón de diseño *Estado,* este patrón se basa en la creación de clases nuevas para cada estado posible del objeto extrayendo todos los métodos del estado, colocándolos en la clase nueva, este patrón es muy similar al *Estrategia* con la salvedad de que aquí los diferentes estados si que se conocen entre sí permitiendo transiciones entre ellos cuando con el otro patrón las estrategias son independientes.

El código realizado para el biestable sería con la siguiente estructura:

- La clase Biestable posee todas las cualidades del dispositivo, encontramos un atributo de tipo Estado con el que implementamos todos los métodos: estado() nos indicará en formato string si está abierto o cerrado, los métodos abrir() y cerrar() cambiarán a rojo o verde según el estado en el que estemos y cambioColor() cambiará al estado enviado como parámetro.

```
package biestables;

public class Biestable {

    private Estado estado;

    public Biestable() {
        estado = new Rojo(this);
    }

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

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

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

    public void cambioColor(Estado state) {
        this.estado = state;
    }
}
```

- He creado una clase abstracta Estado con un constructor al que le llegará un biestable como parámetro y con los tres métodos que dispondrán las subclases.

```java
package biestables;

public abstract class Estado {

    protected Biestable bs;

    public Estado(Biestable biestable) {
        this.bs = biestable;
    }

    public abstract void abrir();
    public abstract void cerrar();
    public abstract String estado();

}
```

- Por último las clases Rojo y Verde que heredan de la clase Estado todos los métodos, en la clase Rojo el método abrir() cambiará a Verde y cerrar() está vacío y de manera complementaria se ha implementado la clase Verde.

```java
package biestables;


public class Rojo extends Estado {

    public Rojo(Biestable biestable) {
        super(biestable);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void abrir() {
        // TODO Auto-generated method stub
        super.bs.cambioColor(new Verde(super.bs));

    }

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

    }

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

```java
package biestables;

public class Verde extends Estado {

        public Verde(Biestable biestable) {
            super(biestable);
            // TODO Auto-generated constructor stub
        }

        @Override
        public void abrir() {
            // TODO Auto-generated method stub
            super.bs.cambioColor(new Rojo(super.bs));

        }

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

        }

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

b) Triestable

Al haber utilizado un patrón de diseño acorde a las características de nuestro problema, la ampliación a un modelo de dispositivo triestable será muy sencilla y tendremos la posibilidad de reutilizar prácticamente el código al completo. De esta manera solo necesitaremos añadir la clase Amarillo y reutilizar las clases del apartado anterior adaptandolas al formato triestable. Implementación del triestable en java :

- La clase Triestable hereda la clase Biestable por tanto lo único necesario será modificar el constructor para que el estado inicial sea el Rojo_triestable y el método cambioColor() para que este cambie al rojo del modelo triestable.

```
package triestables;

import biestables.Biestable;

public class Triestable extends Biestable {

    public Triestable() {
        cambioColor(new Rojo_triestable(this));
    }

}
```

- Para los estados Rojo_triestable y Verde_triestable ambos heredarán de sus respectivas clases padre y lo único que hay habría que implementar es que ahora el abrir() Rojo_triestable el cambio será a Amarillo y lo mismo a la hora de cerrar Verde_triestable.

```
package triestables;

import biestables.Biestable;
import biestables.Rojo;
import biestables.Verde;

public class Rojo_triestable extends Rojo{

    public Rojo_triestable(Biestable bs) {
        super(bs);
    }

    @Override
    public void abrir() {
        // TODO Auto-generated method stub
        bs.cambioColor(new Amarillo(bs));

    }
}
```

```
package triestables;

import biestables.Biestable;
import biestables.Rojo;
import biestables.Verde;

public class Verde_triestable extends Verde {

    public Verde_triestable(Biestable bs) {
        super(bs);
    }

    @Override
    public void cerrar() {
        // TODO Auto-generated method stub
        bs.cambioColor(new Amarillo(bs));

    }
}
```

- A continuación la gran diferencia respecto al biestable, simplemente he añadido lun estado Amarillo intermedio actuando igual que Rojo y Verde. Al abrir(), se cambia al Verde_triestable y al cerrar() se cambia al Rojo_triestable.

```java
package triestables;

import biestables.Biestable;
import biestables.Estado;

public class Amarillo extends Estado {

    public Amarillo(Biestable biestable) {
        super(biestable);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void abrir() {
        // TODO Auto-generated method stub
        super.bs.cambioColor(new Verde_triestable(super.bs));
    }

    @Override
    public void cerrar() {
        // TODO Auto-generated method stub
        super.bs.cambioColor(new Rojo_triestable(super.bs));
    }

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

c) Transición de Biestable a Triestable

Por último necesitamos ampliar el sistema para que se pueda transitar de Biestable a Triestable en cualquier momento al igual que en el apartado anterior al haber elegido un patrón de diseño abierto a cambios con tan solo añadir el método cambio() a la clase Biestable y a la clase abstracta Estado.

- Añadimos a Biestable el método cambio().

```java
package biestables;

public class Biestable {

    private Estado estado;

    public Biestable() {
        estado = new Rojo(this);
    }

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

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

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

    public void cambioColor(Estado state) {
        this.estado = state;
    }

    public void cambio() {
        this.estado.cambio();
    }
}
```

- Añadimos a Estado el método cambio().

```java
package biestables;

public abstract class Estado {

    protected Biestable bs;

    public Estado(Biestable biestable) {
        this.bs = biestable;
    }
    public abstract void abrir();
    public abstract void cerrar();
    public abstract String estado();

    protected abstract void cambio();

}
```

- Para los estados Rojo y Verde tan solo tenemos que añadir el método cambio() , haciendo el cambio al color análogo del triestable pasando de Rojo a Rojo_triestable y de la misma manera de Verde a verde_triestable.

```java
package biestables;
import triestables.Rojo_triestable;

public class Rojo extends Estado {

    public Rojo(Biestable biestable) {
        super(biestable);
        // TODO Auto-generated constructor stub
    }

    @Override
    public void abrir() {
        // TODO Auto-generated method stub
        super.bs.cambioColor(new Verde(super.bs));

    }

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

    }

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

    @Override
    protected void cambio() {
        // TODO Auto-generated method stub
        super.bs.cambioColor(new Rojo_triestable(super.bs));

    }
}
```

```java
package biestables;
import triestables.Verde_triestable;

public class Verde extends Estado {

        public Verde(Biestable biestable) {
            super(biestable);
            // TODO Auto-generated constructor stub
        }

        @Override
        public void abrir() {
            // TODO Auto-generated method stub
            super.bs.cambioColor(new Rojo(super.bs));

        }

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

        }

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

        @Override
        protected void cambio() {
            // TODO Auto-generated method stub
            super.bs.cambioColor(new Verde_triestable(super.bs));
        }
}
```

Github

https://github.com/jesusaldanamartin/designPatterns_Triestable