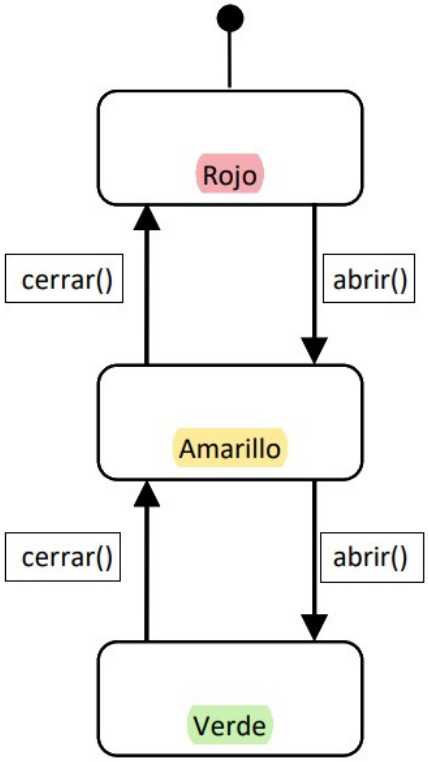
Ingeniería del Software Avanzada

Práctica 2 Patrones de Diseño



Autor: David Ramírez Arco

Fecha: 1 de mayo de 2022

GitHub: https://github.com/Archerd6/Practica-2--design-pattern.git

Índice

Apartados

Apartado a)

a) Descríbase un patrón de diseño que permita implementar de manera satisfactoria dispositivos que, como el mencionado, reaccionan de forma distinta ante el mismo mensaje, dependiendo de su estado interno. Implementar en Java una particularización de dicho patrón de diseño para implementar el dispositivo Biestable descrito.

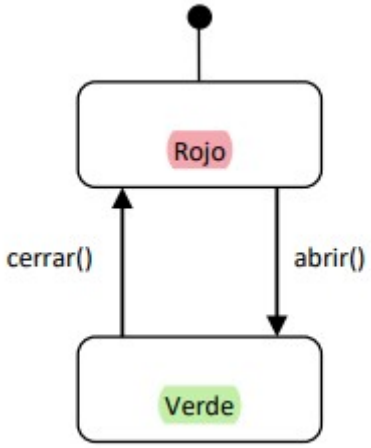


Figura (a) Biestable

Para este problema pienso que el patrón de diseño *Estado* es el más útil, ya que tenemos diferentes eventos (métodos cerrar() y abrir()) en la clase `Semaforo` que hacen variar la situación en la que se encuentra esta misma.

• Clase `Semaforo`

```java
// State Pattern
package semaforo.biestable;

/** Biestable */
public class Semaforo
{
	private static Condition actual;

	public Semaforo(Condition nouveau)
	{
		actual=nouveau;
	}

	public static void setStrategy(Condition nouveau)
	{
		actual=nouveau;
	}

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

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

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

• Interfaz `Condition`

```java
package semaforo.biestable;

public interface Condition
{
	/**Devuelve la cadena “cerrado” cuando está en Rojo y “abierto” cuando está en Verde*/
	public String estado();
	public void cerrar();
	public void abrir();
}
```

```java
package semaforo.biestable;

public class Verde implements Condition
{
	public Verde()
	{

	}

	public String estado()
	{
		return "abierto";
	}

	public void cerrar()
	{
		Semaforo.setStrategy(new Rojo());
	}

	public void abrir()
	{
		// "... Un mensaje abrir() en estado Verde no tendrá efecto ..."
	}
}
```

• Clase `Verde`

```java
package semaforo.biestable;

public class Rojo implements Condition
{
	public Rojo()
	{

	}

	public String estado()
	{
		return "cerrado";
	}

	public void cerrar()
	{
		//  ... no tendrá efecto, ni un mensaje cerrar() en estado Rojo ...
	}

	public void abrir()
	{
		Semaforo.setStrategy(new Verde());
	}
}
```

• Clase `Rojo`

Apartado b)

Supongamos ahora que deseamos implementar un dispositivo Triestable. Tal como muestra la Figura (b), un Triestable incorpora un estado intermedio Amarillo en el que la respuesta al método estado() será la cadena "precaución".

Al haber aplicado un patrón de diseño, se facilita la creación/modificación del apartado b). Solo he tenido que añadir otro estado nuevo, el `Amarillo`.

Para adaptarlo al programa solamente ha habido que cambiar los métodos abrir() y cerrar() de las clases rojo y verde. Esta solución aporta facilidad para la implementación, ya que partimos del anterior Biestable (la reutilización de código es casi total, solo varía el nombre y sus estados de transición al ejecutar los distintos métodos).

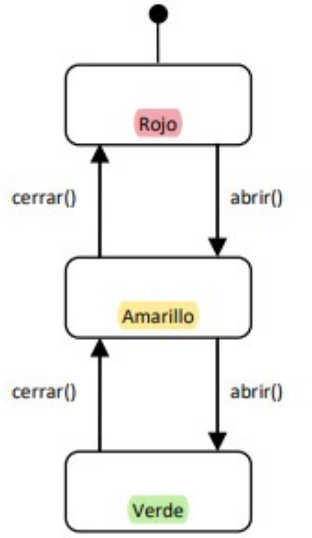


Figura (b) Triestable

Para esta parte he seguido utilizando el patrón Estado (Siendo de utilidad y muy versátil para las modificaciones, por lo que no hay necesidad de implementar otro).

```
1  package semaforo.triestable;
2
3  public class Amarillo implements Condition
4  {
5          public Amarillo()
6          {
7
8          }
9
10         public String estado()
11         {
12                 return "precaución"; //  ... la respuesta al método estado() será la cadena "precaución"...
13         }
14
15         public void cerrar()
16         {
17                 Semaforo.setStrategy(new Rojo());
18         }
19
20         public void abrir()
21         {
22                 Semaforo.setStrategy(new Verde());
23         }
24 }
```

Apartado c)

Supongamos por último que necesitamos realizar una nueva ampliación de nuestro sistema, en el que a la recepción de un mensaje cambio(), un dispositivo Biestable pasará a partir de ese momento a comportarse como un Triestable, y viceversa.

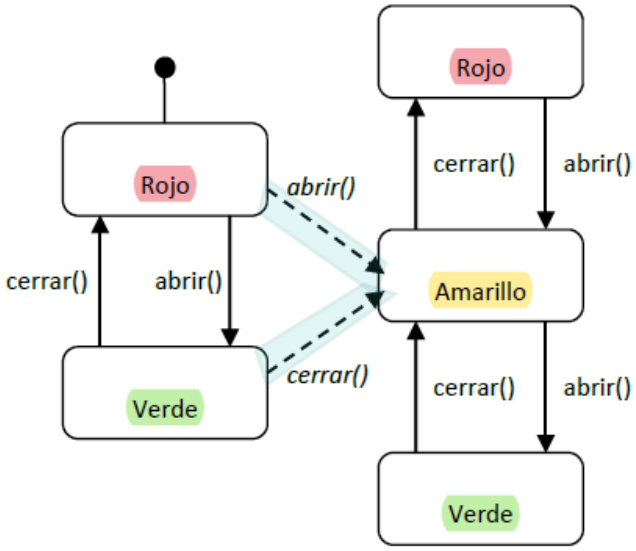


Figura (c) Transición de Biestable a Triestable

Para este apartado he implementado dos nuevas clases que son como las clases `Rojo` y `Verde` solo que estas serían biestables (transicionan entre las dos).

De esta manera la interfaz `Condition` actúa además como un patrón Estrategia que incluye el método cambio() para aplicar una estrategia u otra. Esto sigue siendo una ampliación al patrón estado que ya se había implementado, reutilizando mucha parte del código.

Cuando se use cambio() en `Rojo` o `Verde` pasará a ser `Rojo_biestable` o `Verde_biestable`, sin embargo, cuando se use cambio() en alguno de los estados en modo biestable, pasará a ser amarillo.

A la hora de decidir a qué estado cambiará amarillo he decido que pase a `Rojo_biestable` ya que, ante la duda en un semáforo, mejor pararse.

(El código sería el mismo, pero con el método cambio() y las nuevas clases)

```java
package semaforo.completo;

public interface Condition
{
	/**Devuelve la cadena “cerrado” cuando está en Rojo, “abierto” cuando está en Verde y “precaucion” cuando está en Amarillo*/
	public String estado();
	public void cerrar();
	public void abrir();
	/**A la recepción de un mensaje cambio(), un dispositivo Biestable pasará a partir de ese momento a comportarse como un Triestable, y viceversa.*/
	public void cambio();
}
```

```java
// State Pattern
package semaforo.completo;

/** Completo (Biestable y Triestable) */
public class Semaforo
{
	private static Condition actual;

	public Semaforo(Condition nouveau)
	{
		actual=nouveau;
	}

	public static void setStrategy(Condition nouveau)
	{
		actual=nouveau;
	}

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

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

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

	public void cambio()
	{
		actual.cambio();
	}
}
```

```java
package semaforo.completo;

public class Verde implements Condition
{
	public Verde()
	{

	}

	public String estado()
	{
		return "abierto";
	}

	public void cerrar()
	{
		Semaforo.setStrategy(new Amarillo());
	}

	public void abrir()
	{
		// "... Un mensaje abrir() en estado Verde no tendrá efecto ..."
	}

	public void cambio()
	{
		Semaforo.setStrategy(new Verde_biestable());
	}
}
```

```java
package semaforo.completo;

public class Rojo implements Condition
{

	public Rojo()
	{

	}

	public String estado()
	{
		return "cerrado";
	}

	public void cerrar()
	{
	//  ... no tendrá efecto, ni un mensaje cerrar() en estado Rojo ...
	}

	public void abrir()
	{
		Semaforo.setStrategy(new Amarillo());
	}

	public void cambio()
	{
		Semaforo.setStrategy(new Rojo_biestable());
	}
}
```

```java
package semaforo.completo;

public class Amarillo implements Condition
{

	public Amarillo()
	{

	}

	public String estado()
	{
		return "precaución"; //  ... la respuesta al método estado() será la cadena "precaución"...
	}

	public void cerrar()
	{
		Semaforo.setStrategy(new Rojo());
	}

	public void abrir()
	{
		Semaforo.setStrategy(new Verde());
	}

	public void cambio()
	{
		Semaforo.setStrategy(new Rojo_biestable());
	}
}
```

```java
package semaforo.completo;

public class Verde_biestable implements Condition
{
	public Verde_biestable()
	{

	}

	public String estado()
	{
		return "abierto";
	}

	public void cerrar()
	{
		Semaforo.setStrategy(new Rojo_biestable());
	}

	public void abrir()
	{
		// "... Un mensaje abrir() en estado Verde no tendrá efecto ..."
	}

	public void cambio()
	{
		Semaforo.setStrategy(new Amarillo());
	}
}
```

```java
package semaforo.completo;

public class Rojo_biestable implements Condition
{

	public Rojo_biestable()
	{

	}

	public String estado()
	{
		return "cerrado";
	}

	public void cerrar()
	{
	//  ... no tendrá efecto, ni un mensaje cerrar() en estado Rojo ...
	}

	public void abrir()
	{
		Semaforo.setStrategy(new Verde_biestable());
	}

	public void cambio()
	{
		Semaforo.setStrategy(new Amarillo());
	}
}
```

Conclusión

El patrón de diseño *Estado* se adapta muy bien para este proyecto (ya que permite la modificación y la implementación de forma cómoda).