ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Práctica 2: Patrones de Diseño

Realizado por
Florín Babusca Voicu

UNIVERSIDAD DE MÁLAGA
MÁLAGA, ABRIL DE 2022

Apartado A

Para la implementación del biestable, emplearemos el **patrón Estado** el cual permite cambiar el comportamiento de un objeto dependiendo de su estado interno.

Para aplicar este patrón, crearemos una interfaz llamada **ColorXEstable**, la cual tendrá como métodos abstractos abrir() y cerrar(). Esta interfaz será implementada en dos clases, una a la que llamaremos **ColorRojo** y otra llamada **ColorVerde**, las cuales implementarán los métodos mencionados anteriormente.

También tendremos la clase abstracta **XEstable**, la cual tendrá un objeto de ColorXEstable el cuál será empleado para realizar las llamadas cerrar() y abrir().

A la hora de implementar dichos métodos, se nos planteará el problema de cómo referenciar a los diferentes estados entre sí. Es decir, si el atributo que almacena a ColorXEstable es BiestableRojo y usamos el método abrir(), ¿cómo podemos realizar una vinculación dinámica entre los estados sin que estos se conozcan entre sí? Esta pregunta es justo la que responde el **patrón Mediador**, el cual nos permite referenciar a los diferentes estados sin que estos tengan que conocerse.

Para implementar dicho patrón en nuestro problema, crearemos una interfaz llamada **Mediador**, la cual tiene un método que indicará el siguiente estado al que debe cambiar el XEstable.

Para un XEstable de dos estados, implementaremos un MediadorBiestable el cual referenciará al estado complementario del que reciba. Este mediador lo establecerá la clase **Biestable**, la cual es un hijo de XEstable, que inicializará el atributo privado de XEstable que almacena el mediador (elMediador) con el mediador mencionado anteriormente, e informará de dicho mediador a su estado como un parámetro en los métodos abrir() y cerrar().

Dicho estado será quien emplee a MediadorBiestable para que le informe del nuevo estado al que tiene que referenciar.

Una de las desventajas de la implementación es que el Mediador para referenciar al siguiente estado crea el estado. Por ello, quizás hubiese sido interesante implementar el patrón Singular para los estados para así evitar crear constantemente nuevos objetos de estado. No obstante, gracias a que Java posee un recolector de basura para objetos no referenciados, y como nuestro Mediador provee del siguiente estado correcto, el estado anterior es automáticamente eliminado, y por ende, no he considerado necesario implementar dicho patrón.

Finalmente, se ha creado un test en JUnit para comprobar el correcto funcionamiento del biestable.

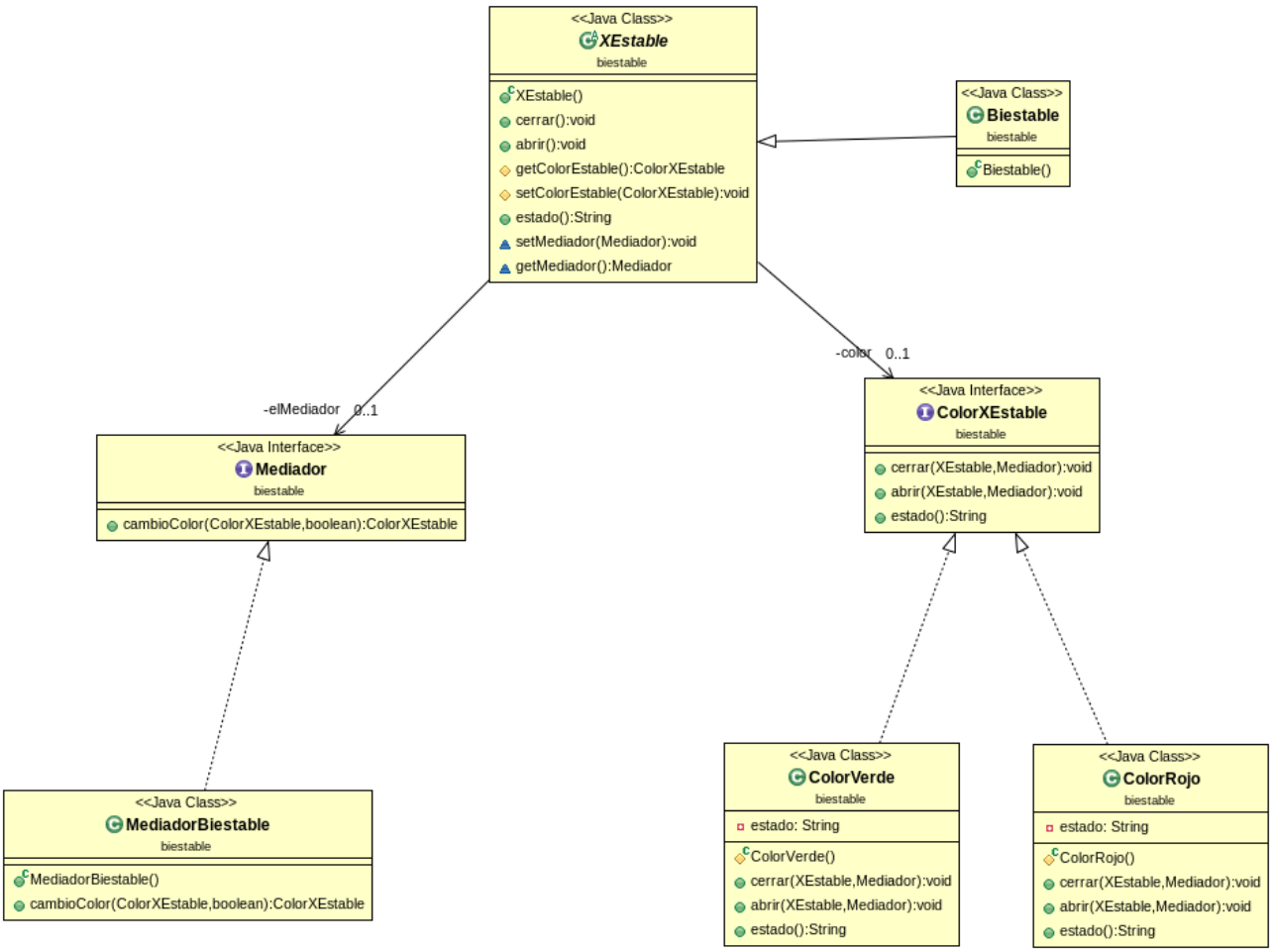

Diagrama UML del apartado A

Apartado B

Gracias a que se ha empleado el patrón Estado y Mediador, extender el código para crear un triestable es sumamente sencillo.

Para ello, simplemente deberemos:

1. Crear una implementación de la interfaz ColorXEstable para representar el estado amarillo.
2. Implementar un Mediador para un triestable.
3. Crear una clase Triestable que extienda a XEstable en la cual se establezca el nuevo mediador creado en el constructor con el método setMediador(Mediador e).

Como podemos observar, gracias a los patrones usados hemos conseguido crear un código que resulta muy fácil de extender y **no requiere ninguna modificación del código ya existente** para ampliar su funcionalidad. Esta característica es previsible ya que estamos intentando resolver un problema muy parecido al del apartado anterior, y gracias a los patrones arquitectónicos empleados, hemos conseguido obtener un código muy reutilizable para problemas semejantes.

En concreto, el **patrón Estado** nos ha brindado la facilidad de **crear tantos estados de un XEstable como necesitemos** sin la necesidad de modificar el código; y **el patrón Mediador** nos ha permitido crear tantos tipos de xestables como queramos sin la necesidad de modificar la estructura de ningún xestable ni de ningún estado ya que los diferentes estados se referencian entre sí gracias al mediador, luego tenemos **desacoplados los estados y los xestables**, lo que conlleva una alta reutilización de código para los diferentes xestables con sus diferentes estados.

No obstante, el código tiene la limitación de que si deseamos añadir un nuevo estado, deberemos o bien modificar o bien implementar un nuevo Mediador que lo controle ya que de lo contrario ningún otro mediador tendría en cuenta al estado nuevo y por ende, el estado no podrá ser usado en un XEstable.

Finalmente, como en el apartado anterior, se ha creado un test que permita probar el nuevo triestable.

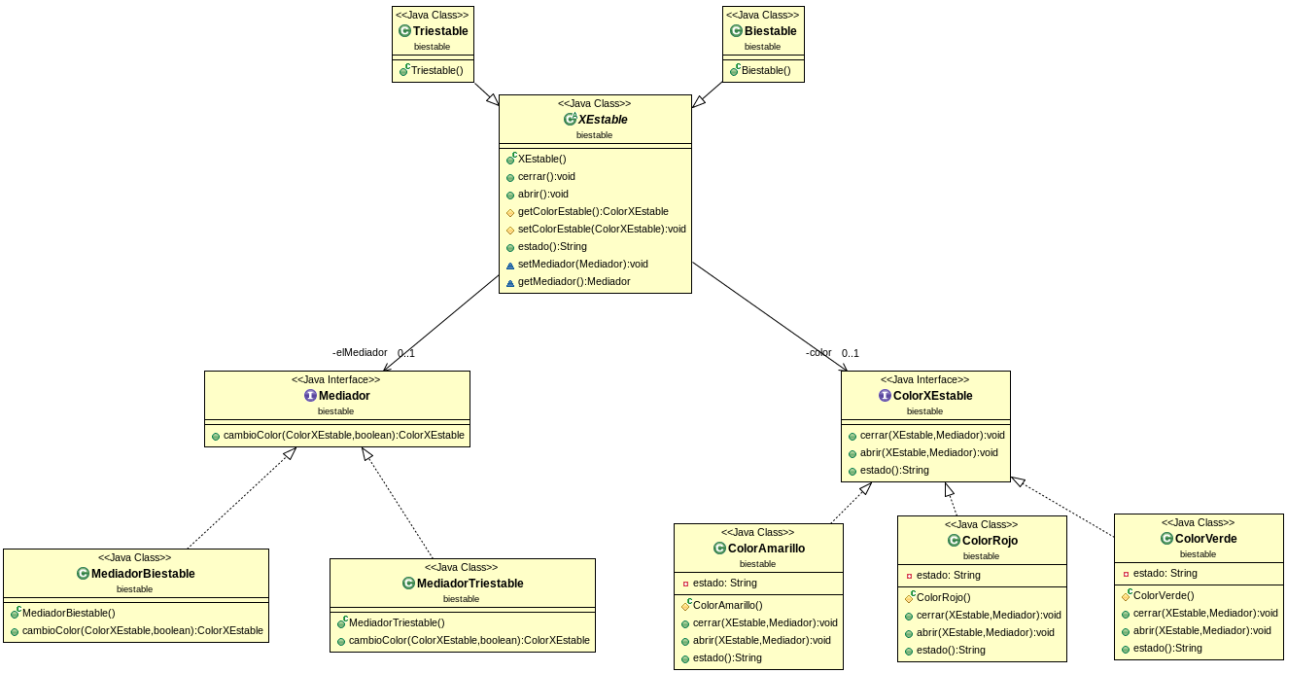

Diagrama UML para Biestables y Triestables

Apartado C

Gracias a que se ha empleado el patrón Mediador, crear un XEstable dinámico se reduce a cambiar el mediador de XEstable. Para ello:

1. Creamos una clase hijo de XEstable, el cual establecerá el mediador MediadorBiestable en su constructor.
2. Tendrá un nuevo método público cambio() en el cual cambiará el mediador y se decidirá a qué estado cambiar cuando el XEstable esté en amarillo.

Como podemos ver, gracias a los patrones anteriores no es necesario modificar ningún aspecto de nuestro código sino simplemente extender XEstable un una nueva clase el cual nos permita alternar en tiempo de ejecución el Mediador usado. Por ello, podemos afirmar que **hemos reutilizado completamente el código ya existente**.

Esta fácil implementación ha sido posible gracias a que tanto los estados como los xestables están desacoplados, luego es trivial extender el código para este problema en concreto.

Para tratar el problema durante el cambio de un triestable a un biestable cuando este esté en color amarillo, considero razonable que se cambie el color a rojo en el biestable ya que este es el color de inicialización del mismo. Para ello, en el método cambio() se comprueba si el estado del triestable es Amarillo, y de ser así, se cambia al rojo. Además, conceptualmente tiene sentido pensar en que si un sistema semáforo está en amarillo se traduzca al color rojo en un semáforo de dos colores ya que la propia DGT afirma que el color amarillo debe ser interpretado como una advertencia de que el semáforo cambiará a rojo pronto.

Finalmente, como en el apartado anterior, se ha creado un test que permita probar el nuevo objeto creado.

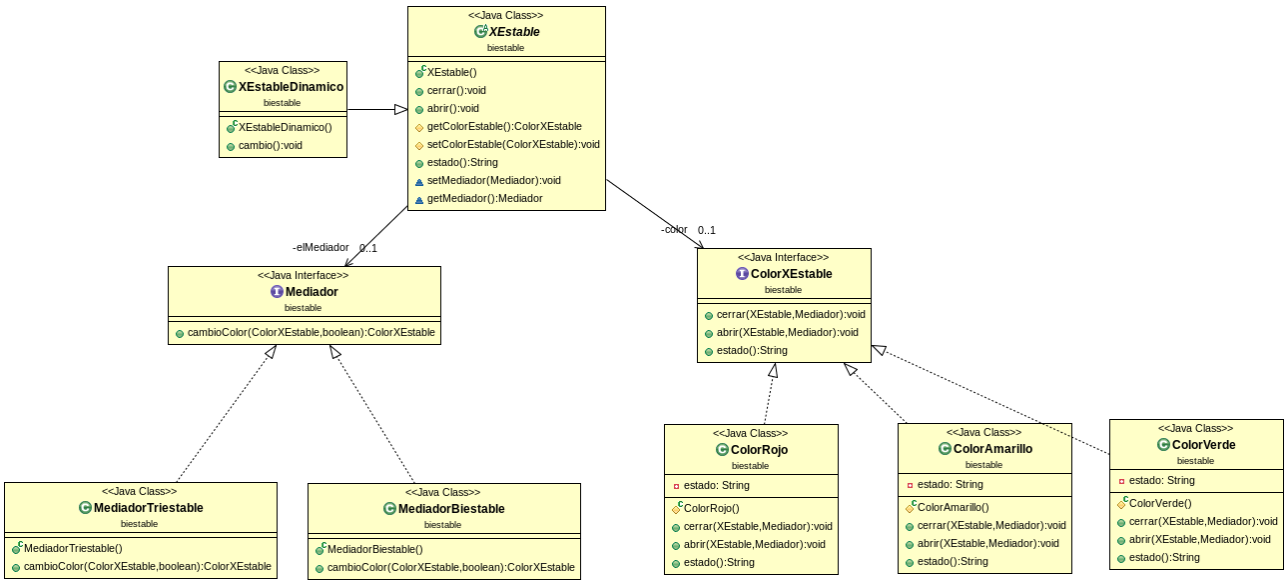

Diagrama UML de un XEstable dinámico

Enlaces de Interés

Repositorio de github para el apartado A: https://github.com/FlorinUMA/Practica-2-A

Repositorio de github para el apartado B: https://github.com/FlorinUMA/Practica-2-B

Repositorio de github para el apartado C: https://github.com/FlorinUMA/Practica-2-C