PRÁCTICA 2
PATRONES DE DISEÑO

PABLO MOLINA SÁNCHEZ
INGENIERÍA DEL SOFTWARE AVANZADA

https://github.com/pablomolinasanchez/PatronDiseno2_pablomolinasanchez

APARTADO A:

El apartado A nos ayuda a situarnos en el ejercicio y generar una base sólida sobre la cual desarrollar el resto de apartados. Por ello, es importante elegir bien el patrón de diseño correspondiente. En mi caso, he elegido el patrón **State** que estará formado por una clase principal *Semáforo* y una clase abstracta *Estado*.

Este patrón de diseño está basado en una clase abstracta, la cual he llamado *Estado*. Esta clase tiene el constructor Estado(semaforo) y Estado(), además de las funcionalidades que los estados, en este caso, Rojo y Verde implementarán, como abrir(), cerrar() y estado() tal y cómo indica el ejercicio. Las clases *Rojo* y *Verde*, que extienden la clase *Estado*, están formadas ambas por sus constructores y la implementación de las funciones anteriormente nombradas en la clase Estado siguiendo las indicaciones del ejercicio de que cuando un semáforo rojo se cierre no haga nada y viceversa con un semáforo en verde. La función estado() devuelve un String indicando el estado en el que se encuentra el semáforo: Si está en rojo devolverá “Cerrado” y si está en verde devolverá “Abierto”.

La clase principal la he definido *Semáforo*, ya que es el ejemplo que hemos usado en clase para una mayor facilidad de entendimiento del ejercicio. Esta clase está formada por el atributo *estado* de tipo Estado. También está formada por su constructor Semáforo() además de las funciones de abrir_semaforo(), cerrar_semaforo() y estado_semaforo() que

según si la variable estado es *Rojo* o *Verde* hace algo distinto, es decir, si estado es rojo y queremos abrir el semáforo, la función abrir_semaforo() llamará al método Rojo.abrir() cambiando estado a Verde. Este proceso se repite en cerrar_semaforo también (llamando a estado.cerrar()) y en estado_semaforo (llamando a estado.estado()).

APARTADO B:

En este apartado B se nos pide que hagamos un triestable reutilizando la mayor cantidad de código posible teniendo en cuenta que en nuestro sistema deberemos disponer tanto de dispositivos biestable como triestable. Para el triestable necesitamos crear un nuevo estado. Por ello, al ser el patrón **State** un patrón de diseño fácilmente extensible, creamos una nueva clase *Amarillo* que también extienden la clase *Estado* al igual que *Rojo* y *Verde*. El método abrir() nos cambiará el estado a verde y el método cerrar() a rojo. El método estado() nos devolverá "Precaución". Como el semáforo puede ser triestable o biestable se añadirán estructuras condicionales al abrir() y cerrar() para realizar funcionalidades distintas según el tipo de semáforo, ya que si un semáforo en rojo es triestable y se abre su estado cambiará a amarillo, mientras que si es biestable cambiará a verde.

En la clase *Semáforo* se creará un nuevo atributo esTriestable de tipo boolean que nos dirá si el semáforo va a ser triestable o biestable, además de dos nuevos constructores: Semaforo(triestable) que nos permitirá crear un semáforo triestable cerrado y *Semaforo(estado, triestable)* que nos permitirá crear un semáforo triestable (o no) en el estado que queramos. Además, se añadirá la función isTriestable() que nos dirá si el semáforo es triestable o es biestable.

APARTADO C:

En el tercer apartado se nos pide que se pueda cambiar de biestable a triestable en cualquier estado del semáforo. Para ello implementamos una nueva función en cada uno de los estados llamada cambio(). Esta función cambiará el atributo esTriestable de true a false o viceversa. Con cambio_semaforo() en la clase *Semáforo* llamaremos a esta función sin importar el estado en el que esté pudiendo así cambiar de biestable a triestable (y al

revés) sin importar si el semáforo está abierto o cerrado. Con esto conseguimos un sistema sencillo y eficaz que se adecua a las exigencias del ejercicio.