Control de temperatura

Patrones que usamos en esta práctica:

- **Patrón estado:**

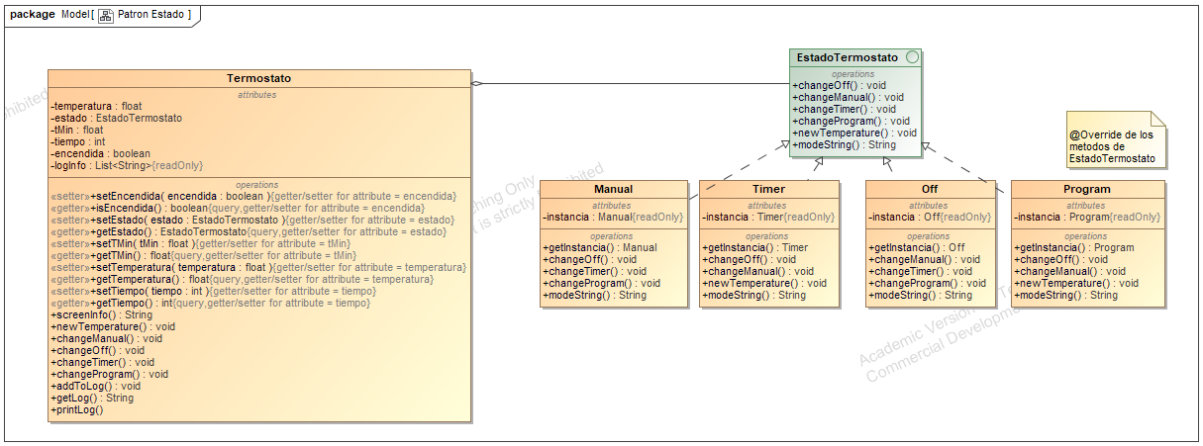
Tenemos una clase Termostato que es la que varía entre los diferentes estados Off, Manual, Program y Timer.
Estos estados son clases que implementan de una interfaz EstadoTermostato donde están declarados sus métodos de variación entre estados, que son default e inicializados a UnsupportedOperationException.
Cada uno de los estados hace un Override de los distintos métodos que implementan ya que cada uno funciona de una manera distinta.

Este estado lo usamos para poder trabajar con los distintos estados de forma independiente y limpia, sin necesidad de mezclar todos los estados en una clase demasiado extensa. Además nos aseguramos de que cada estado cumpla unos requisitos mínimos. También da lugar a una fácil implementación de un nuevo estado en el Termostato, ya que solo tendríamos que crear un nuevo estado que extienda de EstadoTermostato y añadir su condición en cada observador. La creación de un nuevo estado no conlleva a la modificación del código en gran medida.

Hemos decidido optar por el patrón estado porque al menos inicialmente no tiene muchos estados y el número de elementos afectados por los estados es simplemente un termostato.

Una desventaja de la implementación del patrón estado es que es menos compacto que una solución basada en sentencias condicionales, pero como hemos dicho anteriormente esto es el precio de que el programa sea más abierto a cambios. Es mucho más fácil transicionar entre los diferentes estados y saber su funcionamiento.

Nota: Al usar un Patrón estado estamos usando directamente un patrón de instancia única también.



- **Patrón Observador.**

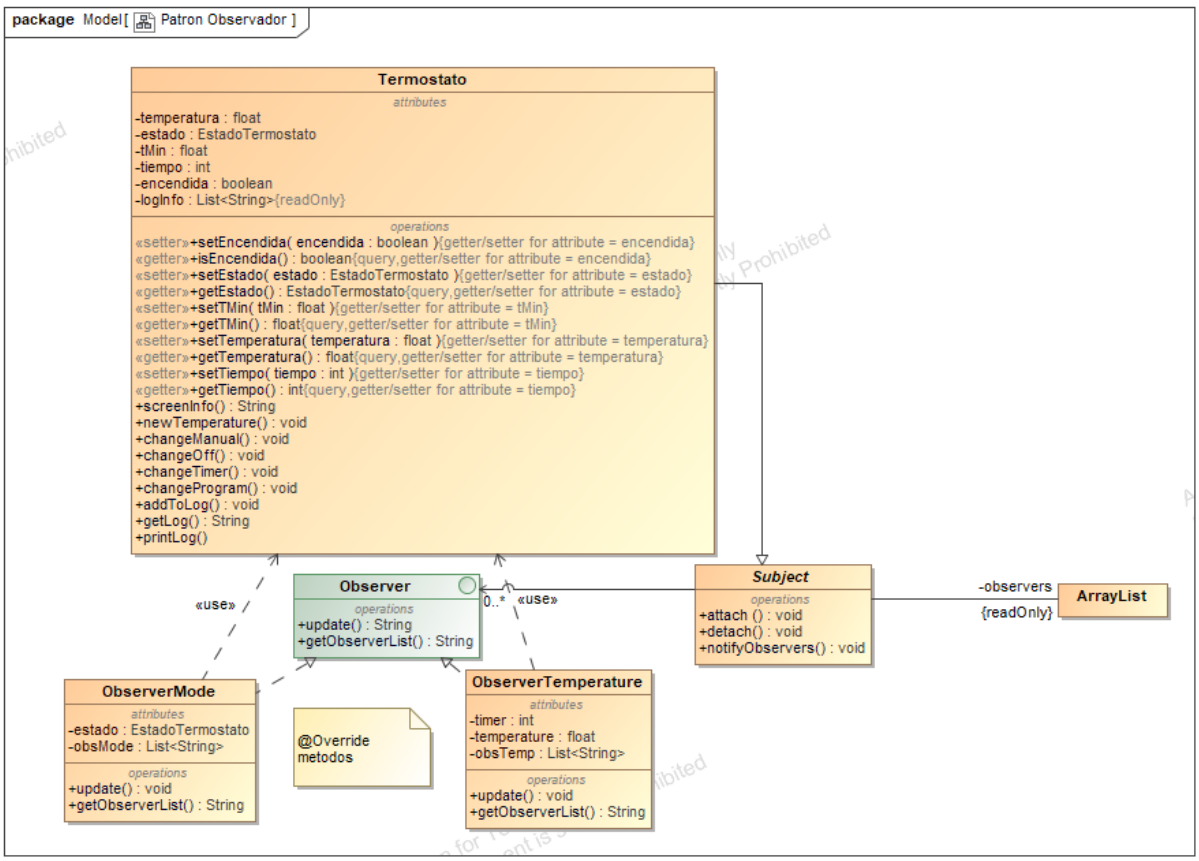
Aplicamos un patrón observador donde el termostato trabaja como sujeto. Tenemos un observador para los estados del termostato y otro para saber si la calefacción está encendida o apagada y comunicar el
estado en el que se encuentra. Cada vez que modificamos la temperatura o hacemos un cambio de estado el termostato notifica a todos los observadores de la acción. Los observadores analizan cuál es el cambio
que se ha realizado y lo imprimen por pantalla.

Hemos usado este patrón porque es la forma más sencilla y limpia de poder analizar los cambios de estado de un objeto, separarlo por casos y además ayuda en la
fácil implementación de futuras nuevas acciones u operadores.

Uno de los inconvenientes que nos hemos encontrado con el Patrón Observador es el hecho de que no tienen ningún tipo de relación entre ellos, en este caso es sencillo controlarlos porque sus funciones son limitadas. En concreto nos obligan a crear una lista de todos los logs en la clase Termostato ya que no podemos informar de un observador a otro. También nos obligan en el propio update del observador a que ellos mismos se den cuenta de si ha cambiado o no algo del termostato.

Hemos usado un pull model.

Las clases implicadas en este patrón son los dos observadores, el sujeto, la interfaz Observer y el propio termostato, que hereda de Subject.



Principios que usamos en esta práctica:

- Principio de la responsabilidad única:

Cumplimos este principio con los observadores, ya que a la hora de lanzar el notify se aplica a los dos observadores que tenemos pero solo uno está interesado en esa llamada y solo uno notifica.
Funciona porque cuando se notifican los observadores lo primero que hacen es comprobar si se ha realizado algún cambio en los estados de los que son responsables, si esto no ha pasado entonces simplemente no hacen nada.
En caso contrario de que se haya producido un cambio comprueba cual se ha producido y lo notifica por pantalla.

- Principio de abierto-cerrado:

Cumplimos este principio tanto en los observadores como en los estados.
En el caso de los estados tenemos una interfaz EstadoTermostato que implementan todos los estados, que a su vez implementan todos los métodos definidos en la propia interfaz de EstadoTermostato pudiendo modificar el funcionamiento de cada uno de ellos con un Override.
En el caso de los observadores también tenemos una interfaz Observer con el método update definido. Los observadores implementan esta interfaz pero cada uno hace un update diferente ya que cada observador redefine update.

- Principio de inversión de dependencia:

Consiste en que las clases e interfaces no dependan de clases concretas que puedan cambiar de funcionamiento y nos obliguen a cambiar la mayor parte de nuestro código.
Los usamos mayoritariamente en el termostato, que cuenta con múltiples inyecciones de dependencia mediante métodos setter o constructores.

Diagrama de Clases y secuencia: