Informe e1

Principios SOLID:

Principio abierto/cerrado:

¿Qué es?

El principio se basa en que una clase o método debe de estar abierta a extensiones pero cerrada a modificaciones. Esto quiere decir que si se quiere añadir algo nuevo no debería haber un motivo por el cual el código anterior sea cambiado.

¿Dónde se usa?

En nuestro código hemos usado este principio en todas las clases de Estado (Off, Manual, Programar y Timer) ya que, en caso de que quisiésemos añadir un quinto estado a nuestro termostato, solo habría que implementar una nueva clase correspondiente al estado y un método en la interfaz, por lo que no estaríamos sobrescribiendo nuestro código anterior. Otra ventaja es que la clase contexto (Termostato) tampoco se vería modificada.

Principio de responsabilidad Única:

¿Qué es?

El principio se basa en que cada objeto debe tener una responsabilidad única dentro de una clase y todos sus servicios están relacionados con esa responsabilidad.

¿Dónde se usa?

Este principio se usa en cada clase Estado, ya que todas esas clases solo tienen una única responsabilidad, la de cambiar de estado.

Principio de inversión de dependencia

¿Qué es?

El principio se basa en que los módulos de alto nivel no deberían depender de los de bajo nivel, lo que quiere decir es que los elementos de alto nivel se comunicarán con los de bajo mediante una interfaz en vez de componentes concretos. A su vez los de bajo nivel deben implementar esa interfaz y sobrescribir los métodos.

¿Dónde se usa?

En nuestro ejercicio la clase alto nivel y concreta es Termostato, esta se relaciona con cada estado mediante una interfaz (EstadoTermostato) por lo que esta clase solo llama a los métodos sin importar lo que ocurre en las clases de bajo nivel.

Es decir, la clase alto nivel que usa la interfaz es Termostato mientras que las de bajo nivel (Off, Manual, Time y Program) sobrescriben los métodos de la interfaz.

Patrones de Diseño:

Instancia Única (Singleton)

¿Qué es?

Este patrón puede hacer que una clase tenga solo una instancia y, globalmente, poder acceder a ella, es decir, estar utilizando el mismo objeto en todo momento.

Este patrón se aplica añadiendo un campo estático privado, donde almacenaremos la instancia, un método estático público donde se obtiene (getter) y hacer un constructor privado, haciendo que solo el método estático de la clase sea capaz de invocarlo.

¿Por qué usarlo?

Hemos usado este patrón para definir solamente una instancia para cada estado y estar seguros de que solo existe una por cada clase y, aparte, poder acceder desde cualquier lugar del programa a ella.

Patrón estado

¿Qué es?

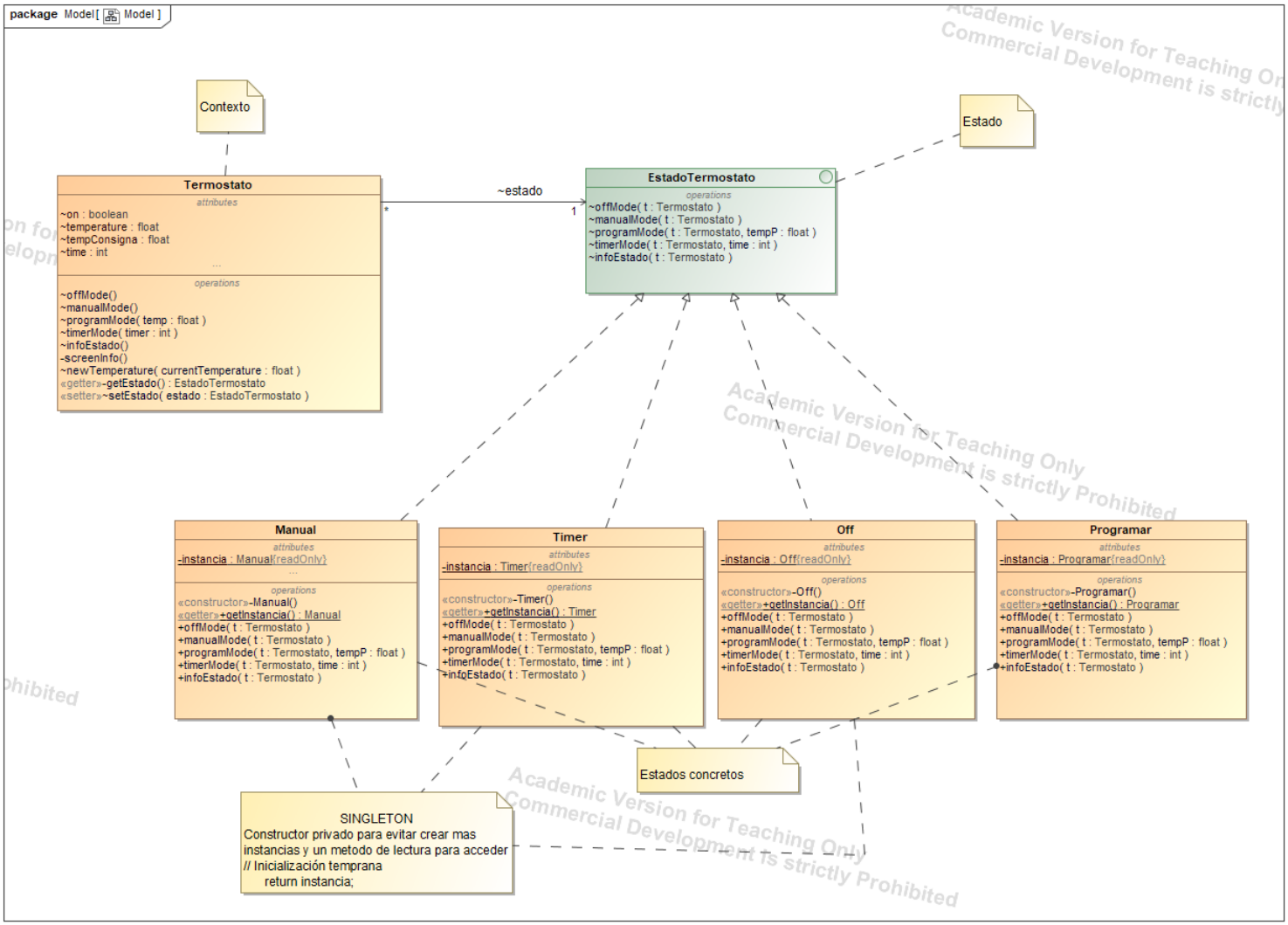
El patrón estado es un patrón de comportamiento que permite modificar este dependiendo del estado en el que se encuentre.

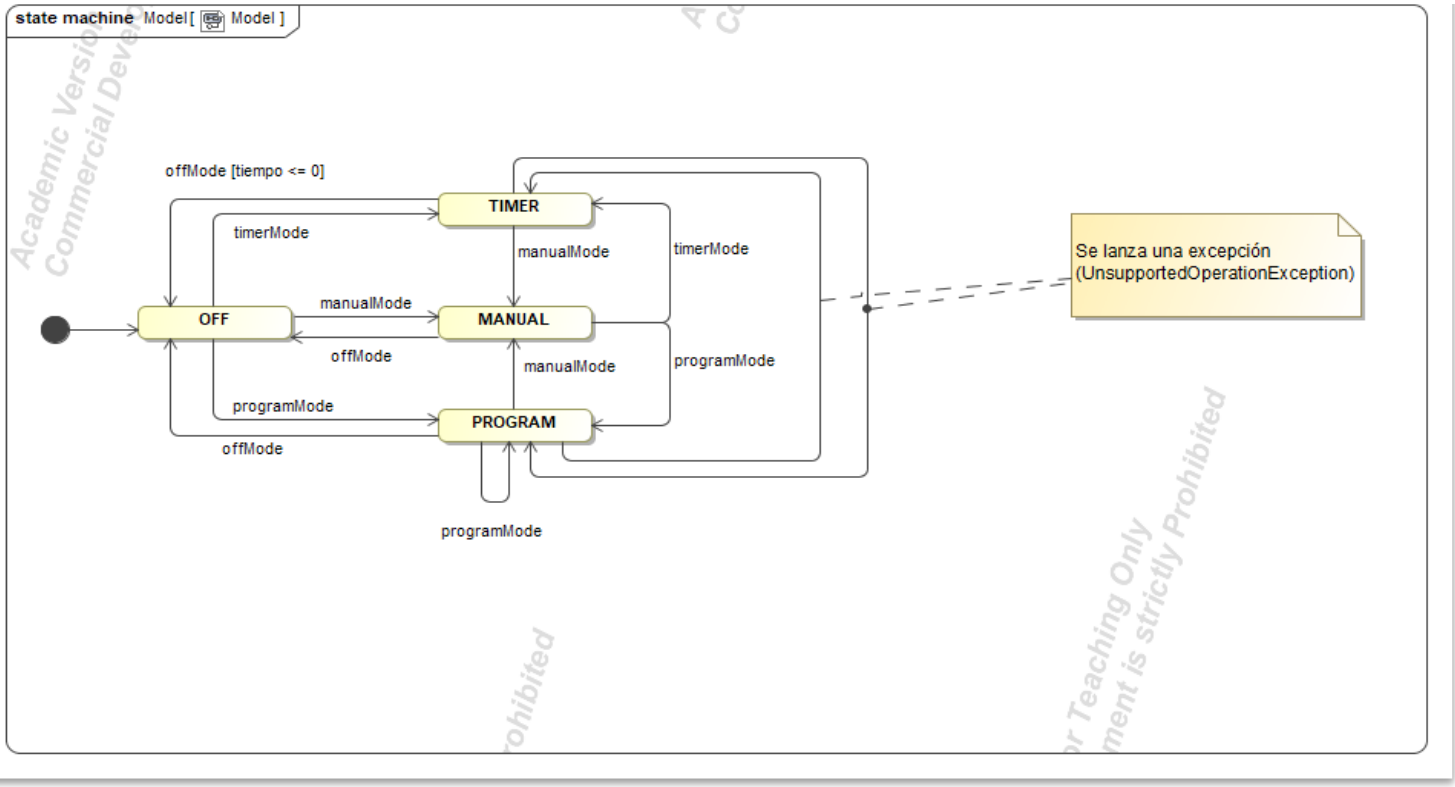
El patrón está estrechamente relacionado con la máquina finita de estados, en este caso la clase, que cambia de un estado a otro (dependiendo de en cual esté).

Para implementarlo se ha usado una interfaz de estado con los métodos que permiten la transición de un estado a otro y creando una clase (pública) por cada estado. Además, se ha definido un modificador (setter) en el objeto principal con el que se sobrescribe el estado.

¿Por qué usarlo?

En el ejercicio se definía un objeto (el termostato) que se comportaba de diferentes maneras dependiendo del estado en el que se encontrase y así no tener que repetir código, ya que hay estados que, aun siendo similares, difieren en algunas líneas.


package Model [Model]
Contexto
Estado
Termostato
attributes
~on : boolean
~temperature : float
~tempConsigna : float
~time : int
operations
~offMode()
~manualMode()
~programMode(temp : float)
~timerMode(timer : int)
~infoEstado()
-screenInfo()
~newTemperature(currentTemperature : float)
«getter»-getEstado() : EstadoTermostato
«setter»~setEstado(estado : EstadoTermostato)
~estado
*
1
EstadoTermostato
operations
~offMode(t : Termostato)
~manualMode(t : Termostato)
~programMode(t : Termostato, tempP : float)
~timerMode(t : Termostato, time : int)
~infoEstado(t : Termostato)
Manual
attributes
-instancia : Manual{readOnly}
operations
«constructor»-Manual()
«getter»+getInstancia() : Manual
+offMode(t : Termostato)
+manualMode(t : Termostato)
+programMode(t : Termostato, tempP : float)
+timerMode(t : Termostato, time : int)
+infoEstado(t : Termostato)
Timer
attributes
-instancia : Timer{readOnly}
operations
«constructor»-Timer()
«getter»+getInstancia() : Timer
+offMode(t : Termostato)
+manualMode(t : Termostato)
+programMode(t : Termostato, tempP : float)
+timerMode(t : Termostato, time : int)
+infoEstado(t : Termostato)
Off
attributes
-instancia : Off{readOnly}
operations
«constructor»-Off()
«getter»+getInstancia() : Off
+offMode(t : Termostato)
+manualMode(t : Termostato)
+programMode(t : Termostato, tempP : float)
+timerMode(t : Termostato, time : int)
+infoEstado(t : Termostato)
Programar
attributes
-instancia : Programar{readOnly}
operations
«constructor»-Programar()
«getter»+getInstancia() : Programar
+offMode(t : Termostato)
+manualMode(t : Termostato)
+programMode(t : Termostato, tempP : float)
+timerMode(t : Termostato, time : int)
+infoEstado(t : Termostato)
Estados concretos
SINGLETON
Constructor privado para evitar crear mas instancias y un metodo de lectura para acceder
// Inicialización temprana
return instancia;


state machine Model [Model]
offMode [tiempo <= 0]
TIMER
timerMode
manualMode
timerMode
OFF
manualMode
MANUAL
offMode
manualMode
programMode
programMode
PROGRAM
offMode
programMode
Se lanza una excepción
(UnsupportedOperationException)