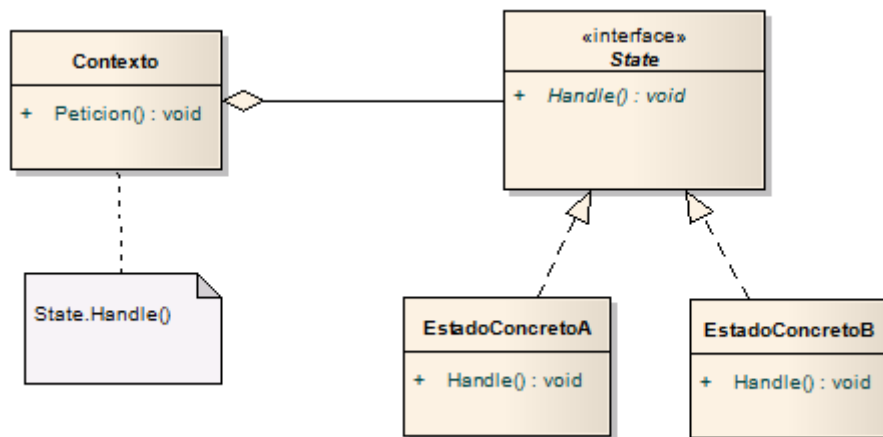


PATRÓN STATE

Objetivo:

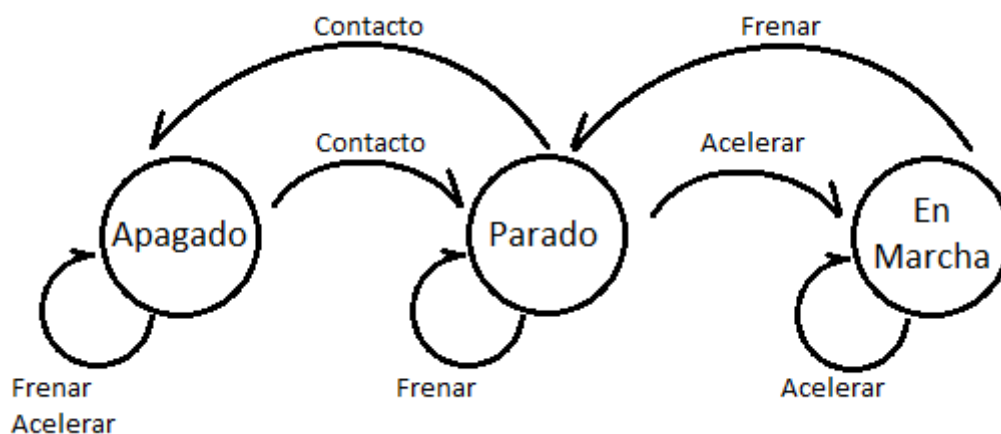
«Permitir que un objeto modifique su comportamiento cuando su estado interno cambie. Parecerá que el objeto cambia de clase».



El patrón *State* tiene la misión fundamental de encapsular el comportamiento de un objeto dependiendo del estado en el que éste se encuentre. ¿A qué nos referimos con estado? Estrictamente hablando, podemos definir el estado de un objeto como el conjunto actual de los valores de los atributos de un objeto.

Por ejemplo, un semáforo, puede estar en estado “avanzar”, “no avanzar” y “precaución”.

Tenemos el concepto de «máquina de estados», que lo podemos resumir con la siguiente imagen, y refiriéndonos al estado que puede tener un automovil:



- Como podemos observar, contamos con una serie de círculos que representan los **estados** (Apagado, Parado, En Marcha).
- Cada uno de esos estados se caracteriza por los valores de una serie de atributos (por ejemplo, para que el vehículo se encuentre *En Marcha*, se debe dar el caso de que el vehículo esté arrancado y con una velocidad distinta de cero).
- ¿Cómo se producen los cambios de estado? A través de las llamadas **transiciones**. Estas transiciones podrían compararse con nuestros **métodos y funciones**, que actúan sobre los valores del objeto haciendo que éste cambie de estado. O no.

Por tanto, tenemos un conjunto limitado de estados (Apagado, Parado, En Marcha) y otro de transiciones (Contacto, Acelerar, Frenar). El factor determinante se encuentra en que **dependiendo en qué estado nos encontremos, las transiciones actuarán de forma distinta**, realizando ciertas operaciones, transiciones a otros estados... o no realizando absolutamente nada.

Como ejemplo, pisar el acelerador con el vehículo apagado no realizará ningún cambio, mientras que si pisamos el acelerador con el vehículo arrancado, el coche se pondrá en movimiento, ganando velocidad (y realizando una transición del estado PARADO al estado EN_MARCHA).

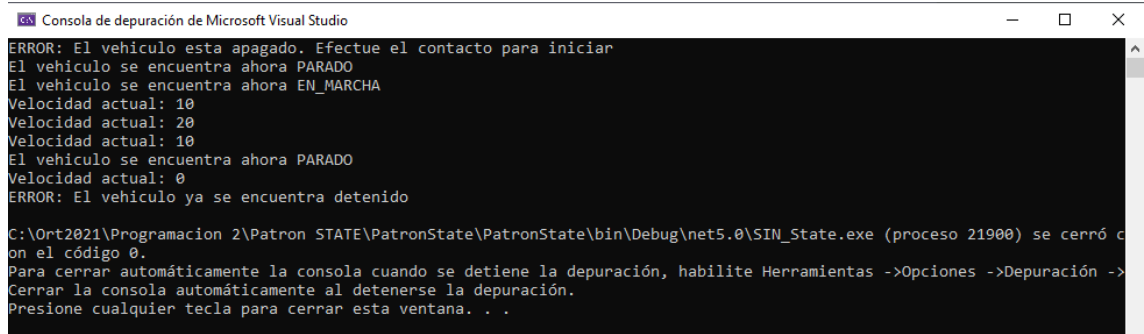
Un pequeño ejemplo “sin usar el patrón STATE”

Modelaremos este ejemplo con una clase a la que llamaremos «VehiculoBasico». Su funcionamiento será sencillo: contará con una variable entera que almacenará el **estado actual**, otra que almacenará la **velocidad**, y un conjunto de métodos que serán los encargados de realizar las **transiciones**. ¿Cómo lo controlamos? Pues a priori, en esta primera parte “sin aplicar el patrón State”, con instrucciones *switch* o *if...else*, que comprueben el estado actual y actúen de forma distinta dependiendo del estado en el que nos encontremos.

Vamos viendo en el proyecto **SIN_State**, todos los siguientes elementos:

1. `public class VehiculoBasico` ver ESTADOS y ATRIBTOS
2. `public void Contacto()`
3. `public void Acelerar()`
4. `public void Frenar()`
5. Codificamos un pequeño programa que hace uso de la clase.
Ver y ejecutar en `static void Main(string[] args)`, para ver todos los cambios

de estado.



```
Consola de depuración de Microsoft Visual Studio
ERROR: El vehículo esta apagado. Efectue el contacto para iniciar
El vehículo se encuentra ahora PARADO
El vehículo se encuentra ahora EN_MARCHA
Velocidad actual: 10
Velocidad actual: 20
Velocidad actual: 10
El vehículo se encuentra ahora PARADO
Velocidad actual: 0
ERROR: El vehículo ya se encuentra detenido

C:\Ort2021\Programacion 2\Patron STATE\PatronState\PatronState\bin\Debug\net5.0\SIN_State.exe (proceso 21900) se cerró c
on el código 0.
Para cerrar automáticamente la consola cuando se detiene la depuración, habilite Herramientas ->Opciones ->Depuración ->
Cerrar la consola automáticamente al detenerse la depuración.
Presione cualquier tecla para cerrar esta ventana. . .
```

Encapsulando el estado

Todo esto resulta muy sencillo, pero ¿qué ocurriría si, por ejemplo, los requisitos de nuestro programa nos exigieran agregar un nuevo estado? Como podemos imaginar, cada uno de los métodos tendrá que agregar una nueva línea a su *switch* para contemplar esta posibilidad, y seguramente también tengamos que modificar el código ya existente para «incluir» el nuevo estado dentro de la funcionalidad del programa.

Por ejemplo, si imaginamos un estado «Sin Combustible» que indique que nos hemos quedado sin combustible, implicaría:

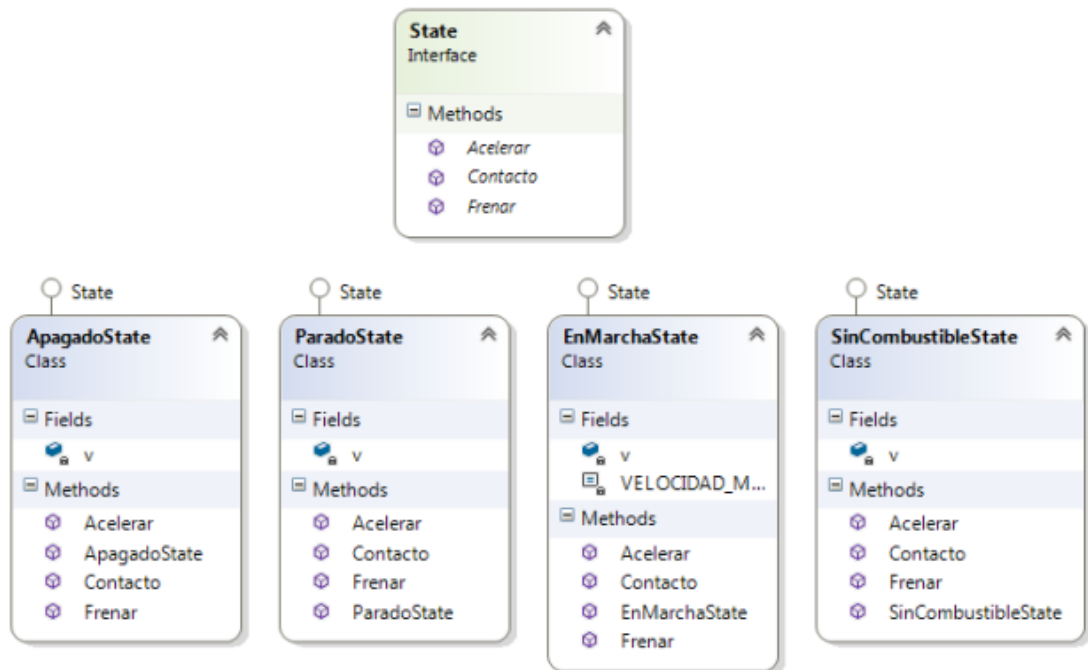
- Agregar una nueva constante `SIN_COMBUSTIBLE` para modelar el estado.
- Agregar un nuevo *case* que contemple este estado en cada uno de los métodos existentes (Acelerar, Frenar y Contacto)
- Modificar los métodos para contemplar que el vehículo no arrancará sin combustible, y que no podrá aumentar su velocidad si nos encontramos en marcha.

“Es mejor!”, seguir el principio de *encapsular lo que cambia* . Y como lo que cambia parecen ser los estados, los vamos a encapsular.

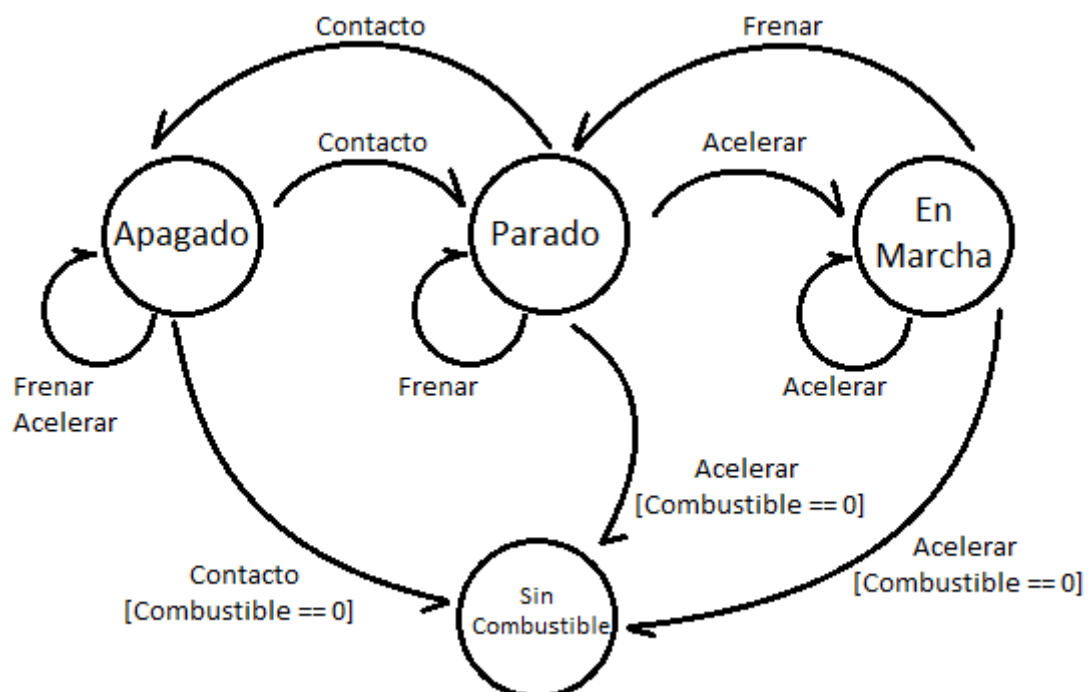
Ver proyecto CON_State:

1. Lo primero que deberemos hacer será crear una interfaz común que todos los estados deberán implementar. Esta interfaz, por lo tanto, deberá exponer los métodos *Acelerar*, *Frenar* y *Contacto*.
2. Lo siguiente que haremos será crear una clase llamada **clase de contexto**, encargada de “mantener” el estado actual junto al resto de los atributos que se consideren oportunos. En nuestro caso concreto, será la clase **vehículo**.

3. Debemos implementar la interfaz con **una clase por cada estado existente en el sistema**. Es decir crear una clase por cada uno de los “estados concretos”, que implemente los métodos de la interfaz *IState*. Además, agregamos un nuevo estado *Sin Combustible*, que será un estado final (una vez alcanzado, no podrá cambiarse de estado). (Ver **DiagramaClaseConState.png**)



La idea de este nuevo modelo es modificar el diagrama anterior para contemplar el estado *SinCombustible*, de forma similar a la siguiente:



4. Implementamos las clases de estado:
 - a. Primero *ApagadoState*
 - b. Creamos otras tres clases similares que implementen *IState*:
ParadoState, *EnMarchaState* y *SinCombustibleState*.

Como vemos, **cada método es independiente e implementa su propia funcionalidad, incluyendo la transición de estados**. Para ello realiza un *new* pasando como parámetro al constructor la referencia al contexto (el objeto de la clase **Vehiculo**). El resto de estados seguirán un patrón similar.

5. Finalmente, disponemos de un fragmento de código que hace del patrón. Ver y ejecutar en `static void Main(string[] args)`, para ver todos los cambios de estado.