PATRÓN COMMAND

Objetivo: «Encapsular una petición como un objeto, de modo que puedan parametrizarse otros objetos con distintas peticiones o colas de peticiones y proporcionar soporte para realizar operaciones que puedan deshacerse».

El patrón *command* u *orden*, es un patrón de comportamiento que tiene como objetivo **encapsular la invocación de un método**.

Entendiendo el patrón Command

Vamos "desmenuzando" el objetivo del patrón para ir entendiéndolo mejor!

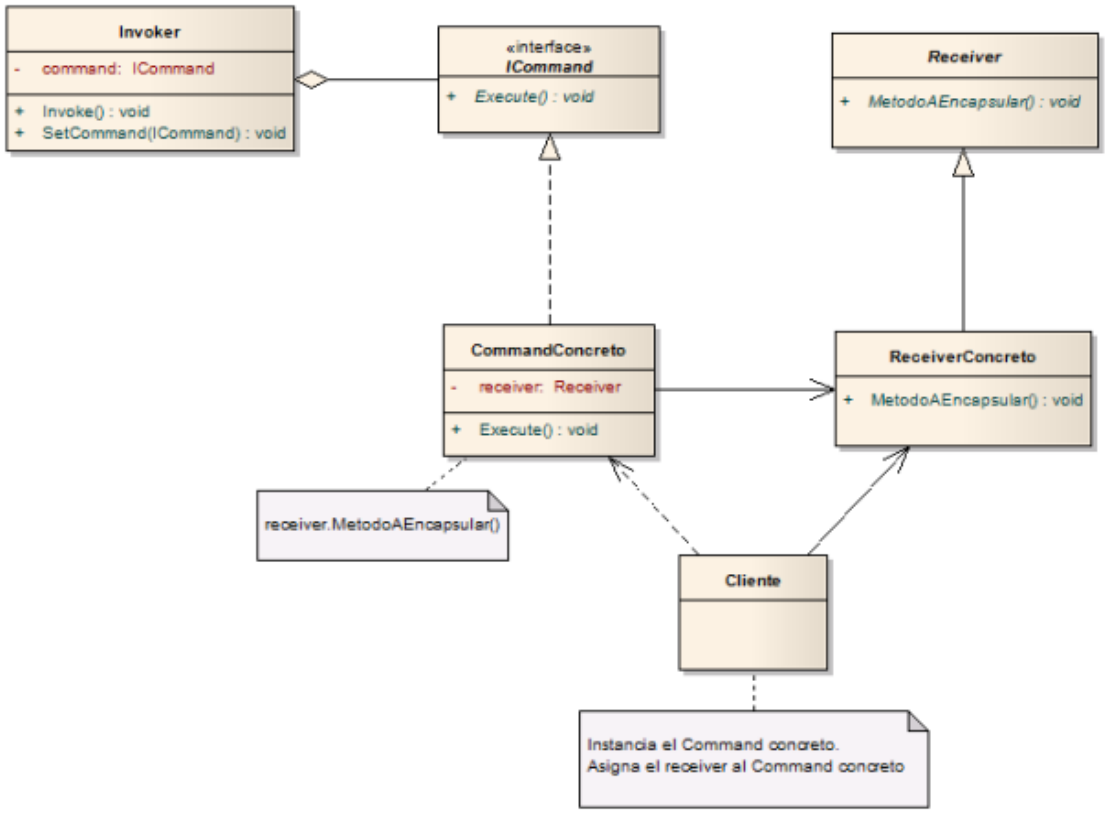
«*Utilizar una interfaz con un método genérico de modo que los objetos que la implementen incluyan una referencia al objeto cuyo método queremos abstraer. Al invocar el método genérico se invocará este método, por lo que el objeto receptor no tiene por qué conocer el objeto original, bastándole una referencia a la interfaz para ejecutar el método*«.

¿Mejor así? Quizás sea todavía un poco complicado, … Intentemos desgranarlo un poco más.

Si tenemos una interfaz común *ICommand* con un método genérico *execute()*, cualquier objeto que la implemente podrá agregar dentro del código de *execute()* la llamada al método que queremos abstraer. De este modo, el objeto receptor queda completamente desacoplado del objeto invocado. Le bastará con invocar *ICommand.execute()* para que ocurra la magia.

El cliente será el encargado de implementar las clases adecuadas para asegurar la funcionalidad que el objeto receptor espera. Nuestro objeto receptor puede ser una aplicación web esperando ejecutar una operación sobre una base de datos. Lo ideal sería que nuestra aplicación no tuviera que preocuparse ni qué objeto usar para realizar la operación ni el nombre de ésta. Si la interfaz *ICommand* proporciona un método genérico para realizar esta operación, nuestros programadores podrán implementar tantos conectores a distintas bases de datos como quieran, y la aplicación simplemente "se conectará a una base de datos, si saber, ni cuál es, ni cómo conectarse, delegando en el método *ICommand.execute()* el resto del trabajo".

Por lo tanto, podemos afirmar que el **patrón *Command* es la quintaesencia de una interfaz**: ofrece la firma de un método genérico que las clases que la implementan se encargarán de completar. Resumiendo, **el patrón *Command* hace que la implementación del método de una interfaz encapsule la invocación del método de otro objeto distinto.**



Si nos centramos en el diagrama de clases del diseño del patrón, podremos identificar los siguientes actores:

- **ICommand**: interfaz que expone el método genérico (*execute()*).
- **CommandGenerico**: implementa *ICommand* y posee una referencia al objeto cuyo método *execute()* tendrá que encapsular. Este objeto recibe el nombre de *Receiver*.
- **Receiver**: como acabamos de decir, es el objeto que implementa la funcionalidad real. Alguno de sus métodos será encapsulado por *ICommand.execute()*.
- **Invoker**: clase encargada de invocar el método *ICommand.execute()*. Posee una referencia (o varias) a *ICommand*, y su método *SetCommand* le permite cambiar su funcionalidad en tiempo de ejecución. Ofrecerá también un método que invoque el método *ICommand.Execute()* que, a su vez, invocará *Receiver.MetodoAEncapsular()*.

Implementando el patrón Command

Volvamos a nuestra fábrica de vehículos. Se nos ha encargado programar parte de la centralita de un nuevo modelo, concretamente de la activación y desactivación de las luces. Sabemos que contaremos con tres tipos de luces: posición, cortas y largas, pero la implementación de los métodos encargados de este proceso aún no están definidas,

ya que se han subcontratado a una empresa coreana. Sin embargo, nuestro cliente nos exige que, pese a ello, debemos avanzar con el diseño de este módulo.

El patrón Command es perfecto en este escenario:

- contamos con un entorno en el que **debemos ejecutar un método que aún no conocemos o que puede cambiar con el tiempo**, por lo que crearemos una interfaz *ICommand* que exponga un método *Execute()*.
- Posteriormente, una vez que los requisitos estén claros y que sepamos **qué** hay que ejecutar, implementaremos clases que se acoplen entre nuestra centralita y el componente que la empresa coreana nos proporcionará.

Por lo tanto, comenzaremos creando una interfaz *ICommand* que se encargará de ofrecer una firma para el método que encapsulará el resto de los métodos: *Execute()*:

1. **Vea el código de la interfaz ICommand.**

A continuación, modelaremos las clases que serán encapsuladas por el objeto *Command*. O más específicamente, las clases cuyos métodos serán encapsulados por el método *Execute()* del objecto *Command()*. Crearemos una clase abstracta de la que heredarán el resto de las clases que serán encapsuladas, aunque en realidad esto no es estrictamente necesario (un objeto *Command* podría albergar diversos objetos no necesariamente relacionados entre sí). **Esta clase es conocida como *Receiver***, y se caracteriza porque se encargará de **albergar la lógica concreta del método**.

2. **Vea el código de la clase abstracta LucesReceiver.**

A continuación, implementaremos las clases concretas cuyos métodos serán encapsulados por *ICommand.Execute()*. Dado que heredan de la clase abstracta *LucesReceiver*, únicamente tendremos que implementar de forma explícita el método *Encender()*, ya que el resto de la funcionalidad será común a los tres tipos de luces. Comenzaremos por las luces de posición:

3. **Vea el código de la clase LucesPosicion.**

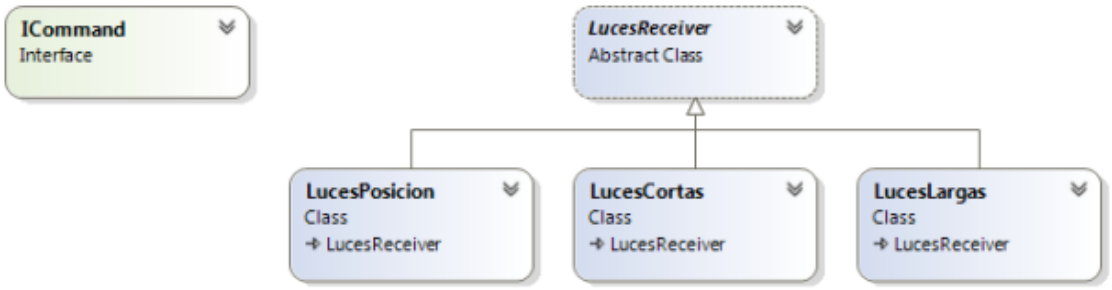
A continuación, hacemos lo mismo con las luces cortas.

4. **Vea el código de la clase LucesCortas.**

Finalmente, realizamos el mismo proceso con las luces largas.

5. **Vea el código de la clase LucesLargas.**

Esto se resumiría ahora mismo en el siguiente esquema:



Ya tenemos implementado uno de los elementos de nuestro patrón. Ahora haremos lo propio con la parte cliente, es decir, nuestra centralita**. O más concretamente, el módulo encargado de encender y apagar las luces.** Este se corresponderá con el elemento *Invoker*, que es aquel que tendrá como objetivo invocar la acción encapsulada por el objeto *Command*. Comenzaremos implementando una interfaz que ofrezca dos operaciones:

- Una operación *SetCommand(ICommand)* que permitirá a nuestro módulo cambiar el *ICommand* a ejecutar.
- Una operación *Invoke()* que invoque el método *ICommand.Execute()* que esté asignado en el momento actual.

6. **Vea el código de la interfaz IInvoker.**

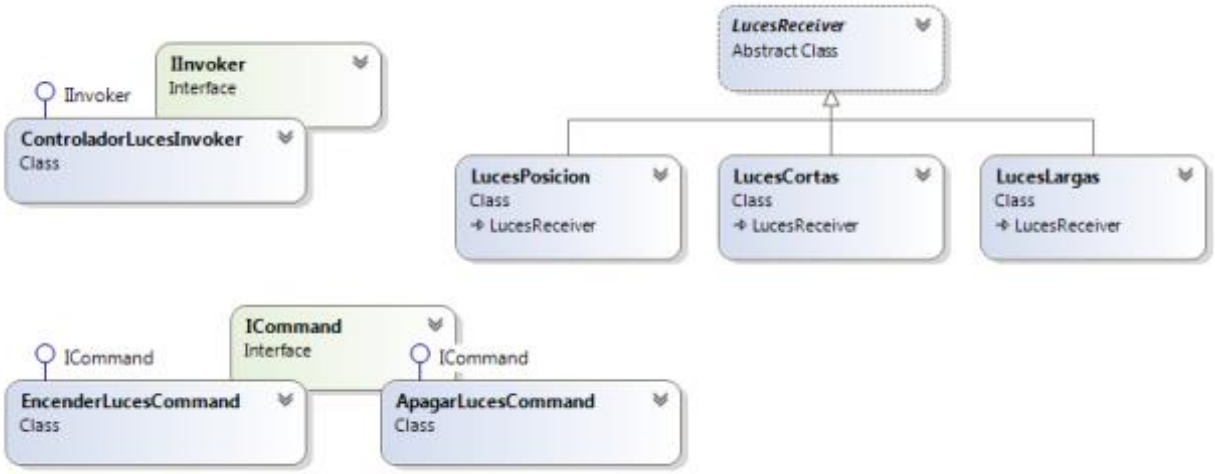
A continuación, implementaremos la interfaz. Lo haremos a través de la clase *ControladorLucesInvoker*. La clase realizará justo aquello que acabamos de definir en la interfaz. Además, incluirá una referencia a un objeto que implemente la interfaz *ICommand*

7. **Vea el código de la clase ControladorLucesInvoker.**

Ya casi hemos completado nuestro patrón. Falta lo más importante: las clases que implementan la clase *ICommand* y que simbolizan **las operaciones que se quieren encapsular**. Estas operaciones serán, **encender y apagar**. Por lo tanto, crearemos las clases **EncenderLucesCommand** y **ApagarLucesCommand**:

8. **Vea el código de las clases EncenderLucesCommand y ApagarLucesCommand.**

Esto completará nuestro diagrama de clases, dejándolo de la siguiente manera:



Finalmente, haremos el código que hará uso del patrón que acabamos de implementar. En él crearemos los tres tipos de luces (posición, cortas y largas) y utilizaremos el método *SetCommand()* de cada uno de los objetos destinados a encender y a apagar las luces para cambiar en tiempo de ejecución el tipo de luz a manejar.

9. **Vea el código del Main en Program.cs.**

```
C:\Ort2021\Programacion 2\Patron COMMAND\ProyectoPatronCommand\PatronCommand\bin\Debug\net5.0\PatronCommand.exe
Encendiendo luces (PatronCommand.LucesPosicion). Alumbrando a una distancia de 1 metros.
Apagando las luces
Encendiendo luces (PatronCommand.LucesCortas). Alumbrando a una distancia de 40 metros.
Apagando las luces
Encendiendo luces (PatronCommand.LucesLargas). Alumbrando a una distancia de 200 metros.
Apagando las luces
```

¿Cuándo utilizar este patrón? Ejemplos reales

Dado que este patrón permite encapsular la ejecución de un método como un objeto, una de las funcionalidades que nos ofrece es la de **encadenar invocaciones**. Si creamos una serie de objetos de este tipo y los almacenamos como objetos, éstos pueden pasarse como parámetros a un método que los vaya ejecutando de forma secuencial.

El segundo escenario en el que este patrón es útil es aquel que hemos visto en el ejemplo, es decir, donde **el invocador del método deba ser desacoplado del objeto que maneja la invocación**. Esto puede darse por necesidades del diseño o bien porque las especificaciones del módulo que maneja la invocación no estén definidas o existe previsión de que vaya a variar con el tiempo.

Recordemos además que la definición del patrón establecía que debía *proporcionar soporte para deshacer operaciones*. ¿Cómo realizar esto? Dado que estas operaciones se encapsulan como objetos, sería perfectamente posible mantener en memoria una pila con elementos ejecutados y otra que almacene objetos que modelen los distintos estados correspondientes a los instantes anteriores a ejecutar cada uno de los *Commands* de la primera pila. De este modo, realizar la opción *deshacer* sería tan sencillo como realizar un *pop* del primer objeto de cada pila y sobreescribir el estado actual del programa con el contenido del primer objeto almacenado en la pila de estados.

Otros ejemplos reales de este patrón serían, por ejemplo:

- Grabación de macros (secuencia de operaciones)
- Asistentes de instalación (cada pantalla implementaría un *Command* que realizaría una operación *Execute()* al pulsar el botón «*Siguiente*»)