Clase Abstracta - Definición

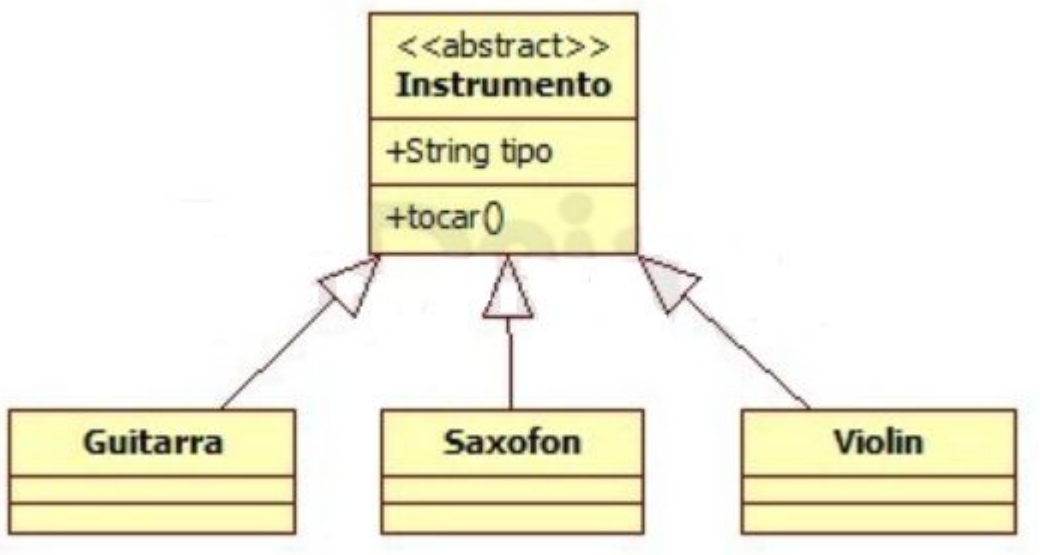
El propósito de una clase abstracta es proporcionar una superclase apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común (Deitel 9° Edición).

- Es una clase que no se puede instanciar (no se pueden crear objetos)
- Se usa únicamente para definir subclases.
- Es una clase que contiene uno o más métodos abstractos (métodos que no tienen implementación).
- Se usa para definir *tipos amplios de comportamientos* en la raíz de la jerarquía de clases y usar sus subclases para proporcionar los detalles de implementación de la clase abstracta.

Clase Abstracta - uso

- Las subclases extienden la misma clase abstracta y proporcionan diferentes implementaciones para los método abstractos.
- Otra clase (clase concreta) tiene que proporcionar la implementación de los métodos abstractos.
- La clase concreta tiene que implementar todos los métodos abstractos de la clase abstracta para que sea usada para instanciarla.
- La primer subclase concreta que herede de una clase abstract debe implementar todos los métodos de la superclase.

Clase Abstracta – Ejemplo I (de la web)



Cada una de las clases concretas (**Guitarra**, **Saxofon** y **Violin**) implementan el método **tocar()** y le dan la funcionalidad dependiendo de como se toque el ***Instrumento***.

Todos los instrumentos musicales se pueden tocar (es un método abstracto).

tocar() es un proceso común en todos los instrumentos sin importar el detalle de como se tocan.

Pero sabemos que una **Guitarra** no se toca de la misma manera que el **Saxofón** y de un **Violín**.

Por lo tanto, al heredar de la clase ***Instrumento***, todas sus clases hijas están obligadas a implementar este método y darle la funcionalidad que le corresponda...

Herencia
Polimorfismo

Clase Abstracta

Clase Abstracta – Ejemplo II (de la web)

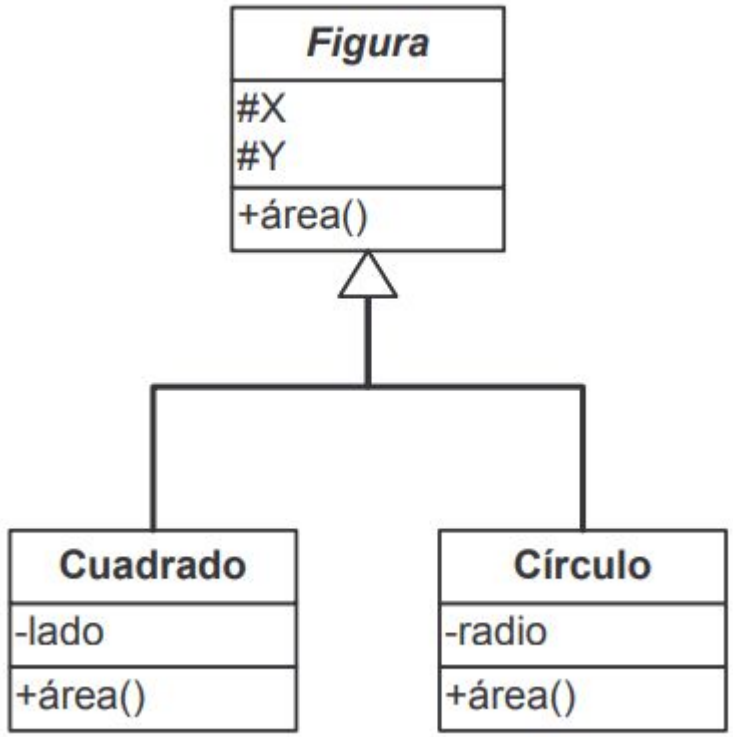


Figura es la clase abstracta.
No tiene sentido calcular su **área()**, si podemos hacerlo del **Cuadrado** o del **Circulo**.

¿En que método/s de que clase/s va a estar la implementación (código) de **área()**?

Herencia
Polimorfismo

Clase Abstracta

Interfaces- Definición (Deitel 8° edition)

- Una interfaz describe a un conjunto de métodos que pueden llamarse en un objeto, pero no proporciona implementaciones concretas para todos ellos.

- Podemos declarar clases que implementen a (es decir, que proporcionen implementaciones concretas para los métodos de) una o más interfaces.

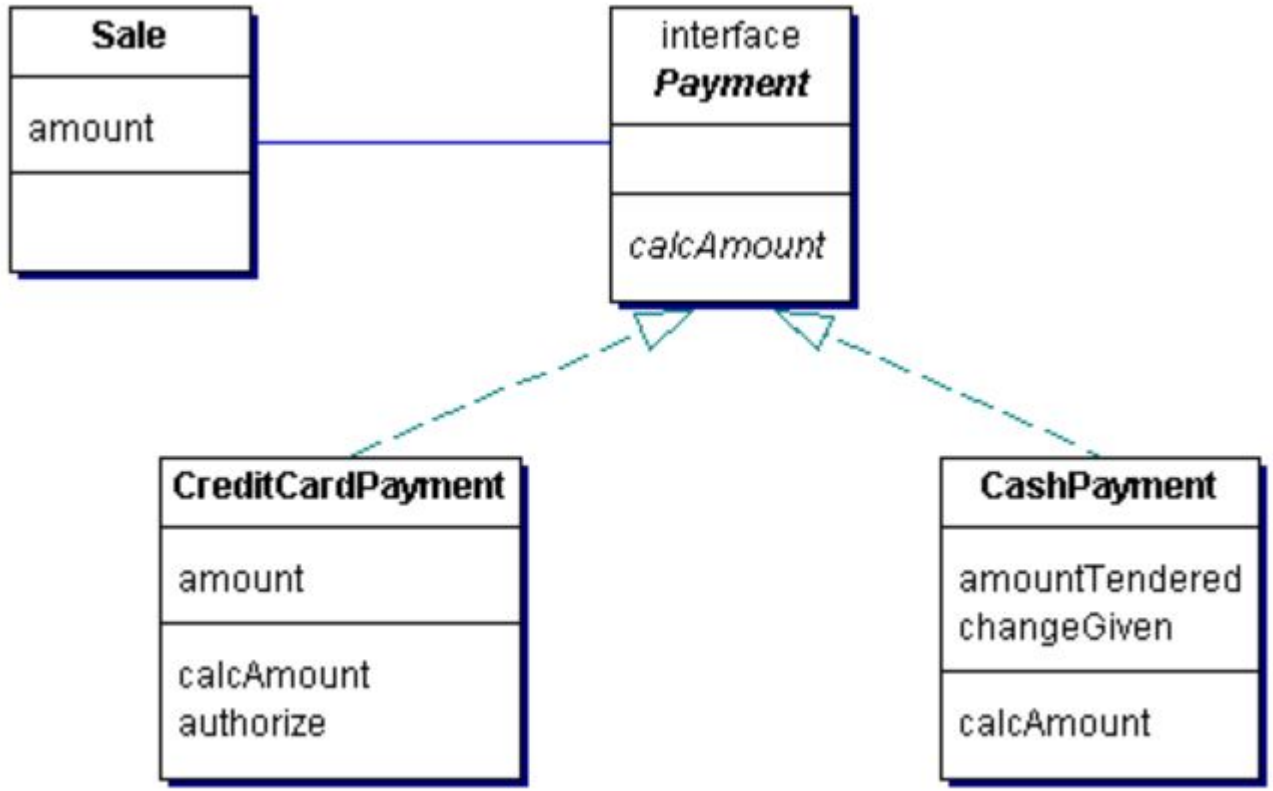
- Cada método de una interfaz debe declararse en todas las clases que implementen a la interfaz.

Interfaces- Definición (Deitel 8° edition)

- Una interfaz describe a un conjunto de métodos que pueden llamarse en un objeto, pero no proporciona implementaciones concretas para todos ellos.
- Una vez que una clase implementa a una interfaz, todos los objetos de esa clase tienen una relación ***es un*** con el tipo de la interfaz, y se garantiza que todos los objetos de la clase proporcionarán la funcionalidad descrita por la interfaz.

Las interfaces son útiles para asignar <u>funcionalidad común</u> a clases que posiblemente no estén relacionadas.

Interfaces – Ejemplo I (de la web)



Patrón State

Permite que un objeto modifique su comportamiento cada vez que cambie su estado.

Propósito		
De Creación	**Estructural**	De Comportamiento
Factory Method	Adapter	**Interpreter**
		Template Method
Abstract Factory	Adapter	**Chain of Responsability**
Builder	Bridge	**Command**
Prototype	Composite	**Iterator**
Singleton	Decorator	**Mediator**
	Facade	**Memento**
	Flyweight	**Observer**
	Proxy	**State**
		Strategy
		Visitor

Patrón de Comportamiento: **State**

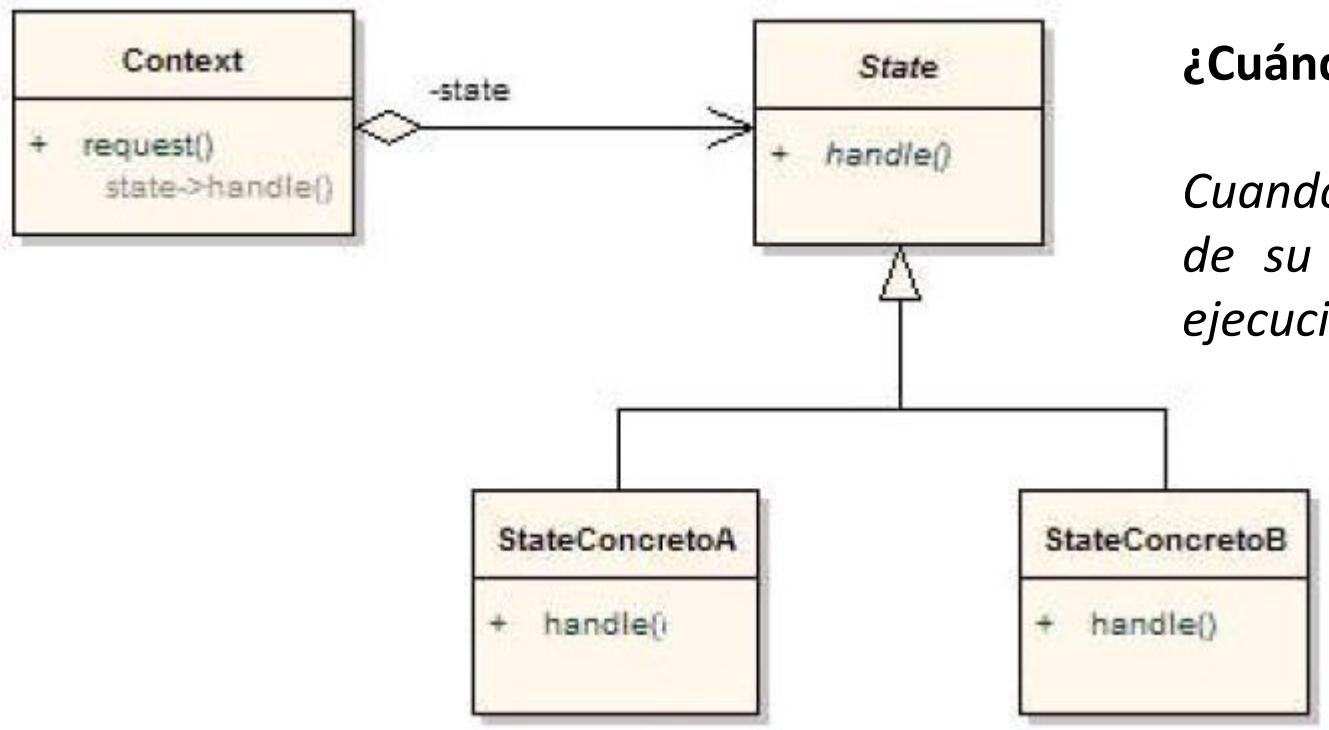
Patrón State

- Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Es decir, busca que un objeto pueda reaccionar (se pueda comportar) según su estado interno.
- Esto se podría solucionar con una variable, y el uso de muchísimos IF-ELSE generando un código ilegible de baja calidad lo que dificulta su mantenimiento.
- Desacopla el estado de la clase en cuestión.
- El patrón State propone una solución a esta situación creando un objeto por cada estado posible.
- En muchas ocasiones se requiere que un objeto tenga diferentes comportamientos según el estado en que se encuentre.

Participantes !

Patrón State

Permite que un objeto modifique su comportamiento cada vez que cambie su estado.



¿Cuándo lo usamos?

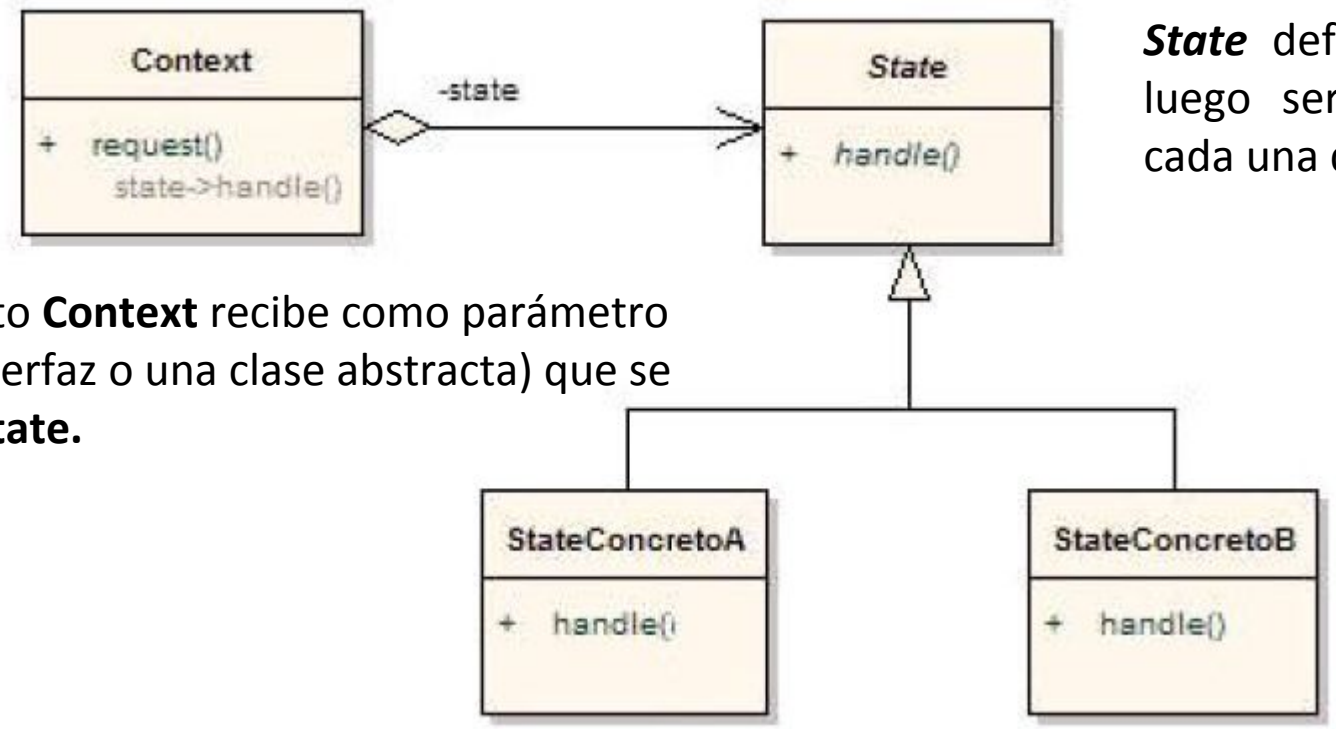
Cuando el comportamiento del objeto depende de su estado y debe cambiar en tiempo de ejecución según el comportamiento del estado.

Patrón de Comportamiento: **State**

Participantes !

Patrón State

Permite que un objeto modifique su comportamiento cada vez que cambie su estado.



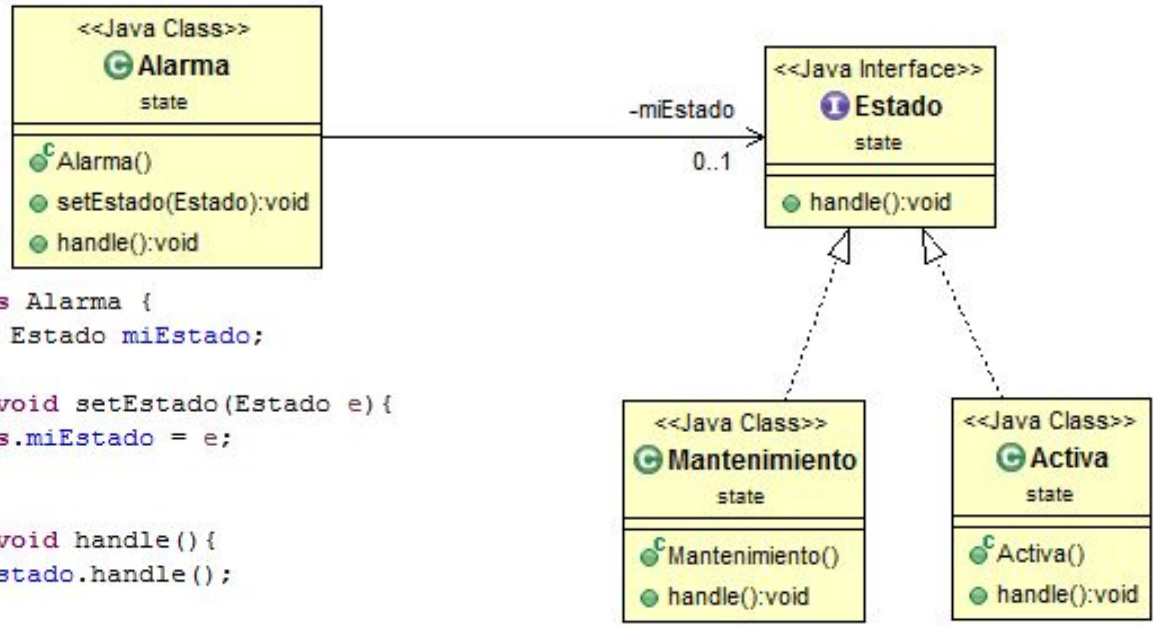
State define un único método **handle()** que luego será sobre-escrito (implementado) en cada una de sus sub-clases.

EL objeto **Context** recibe como parámetro (una interfaz o una clase abstracta) que se llama **State.**

StateConcretoA
StateConcretoB
Son los estados específicos (concretos) son los estados propiamente dicho que contienen el comportamiento que realizará el objeto de acuerdo al estado en el que se encuentre.

Patrón State (Ejemplo de la Web)

Permite que un objeto modifique su comportamiento cada vez que cambie su estado.




```
public class Alarma {
    private Estado miEstado;

    public void setEstado(Estado e){
        this.miEstado = e;
    }

    public void handle(){
        miEstado.handle();
    }
}
```

¿Qué sucede en el main()?
(tiempo de ejecución)

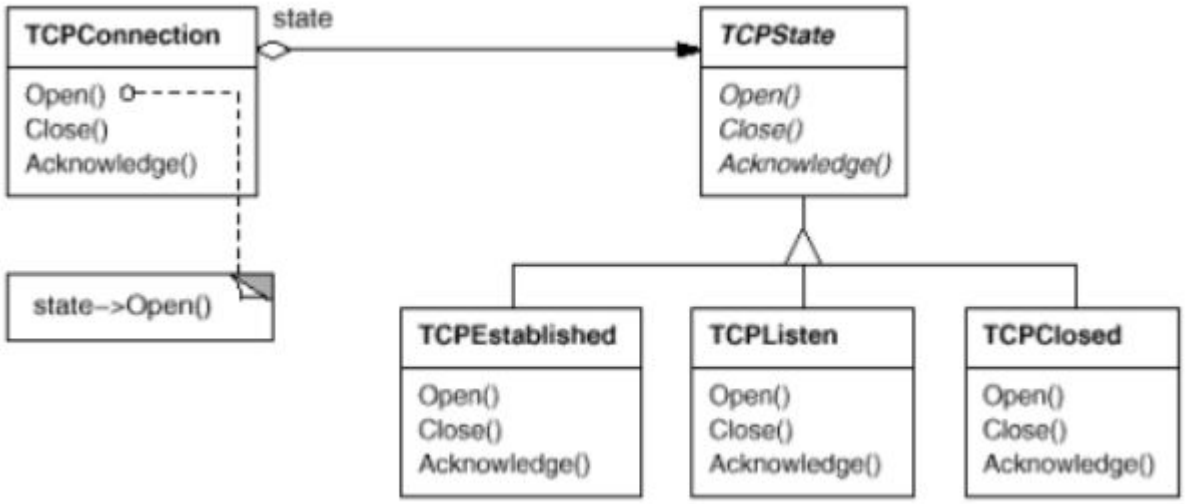
```
public static void main(String[] args) {
    // TODO Auto-generated method stub

    Alarma miAlarma = new Alarma();

    miAlarma.setEstado(new Activa());
    miAlarma.handle();

    miAlarma.setEstado(new Mantenimiento());
    miAlarma.handle();

}
```

```
public class Mantenimiento implements Estado{

    public void handle(){
        System.out.println("Alarma en mantenimiento");
        System.out.println("Avisar al dueño");
    }

}
```

```
public class Activa implements Estado{

    public void handle(){
        System.out.println("Alarma Activada !!!");
    }

}
```

Patrón State (otro ejemplo)

Permite que un objeto modifique su comportamiento cada vez que cambie su estado.



Las subclases implementan el comportamiento específico de cada estado.

Considerar una clase **TCPConnection** que representa una conexión de red.

Un objeto de esta clase puede estar en varios estados (establecido, escuchando, cerrado). Cuando ese objeto recibe solicitudes de otros objetos, responde de forma diferente dependiendo de su estado.

La idea clave de este patrón es introducir una clase abstracta llamada TCPState para representar los estados de la red. Las subclases de esta clase implementan el comportamiento específico de cada estado.

El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de ese estado.

Patrón Strategy

Permite realizar un cambio de estrategia (el algoritmo) en tiempo de ejecución.

Propósito		
De Creación	**Estructural**	De Comportamiento
Factory Method	Adapter	**Interpreter**
		Template Method
Abstract Factory	Adapter	**Chain of Responsability**
Builder	Bridge	**Command**
Prototype	Composite	**Iterator**
Singleton	Decorator	**Mediator**
	Facade	**Memento**
	Flyweight	**Observer**
	Proxy	**State**
		Strategy
		Visitor

Patrón de Comportamiento: **Strategy**

Patrón Strategy

- Este patrón define un conjunto de algoritmos, los encapsula y los hace intercambiables. Permite que los comportamientos de los clientes sean determinados en tiempo de ejecución.

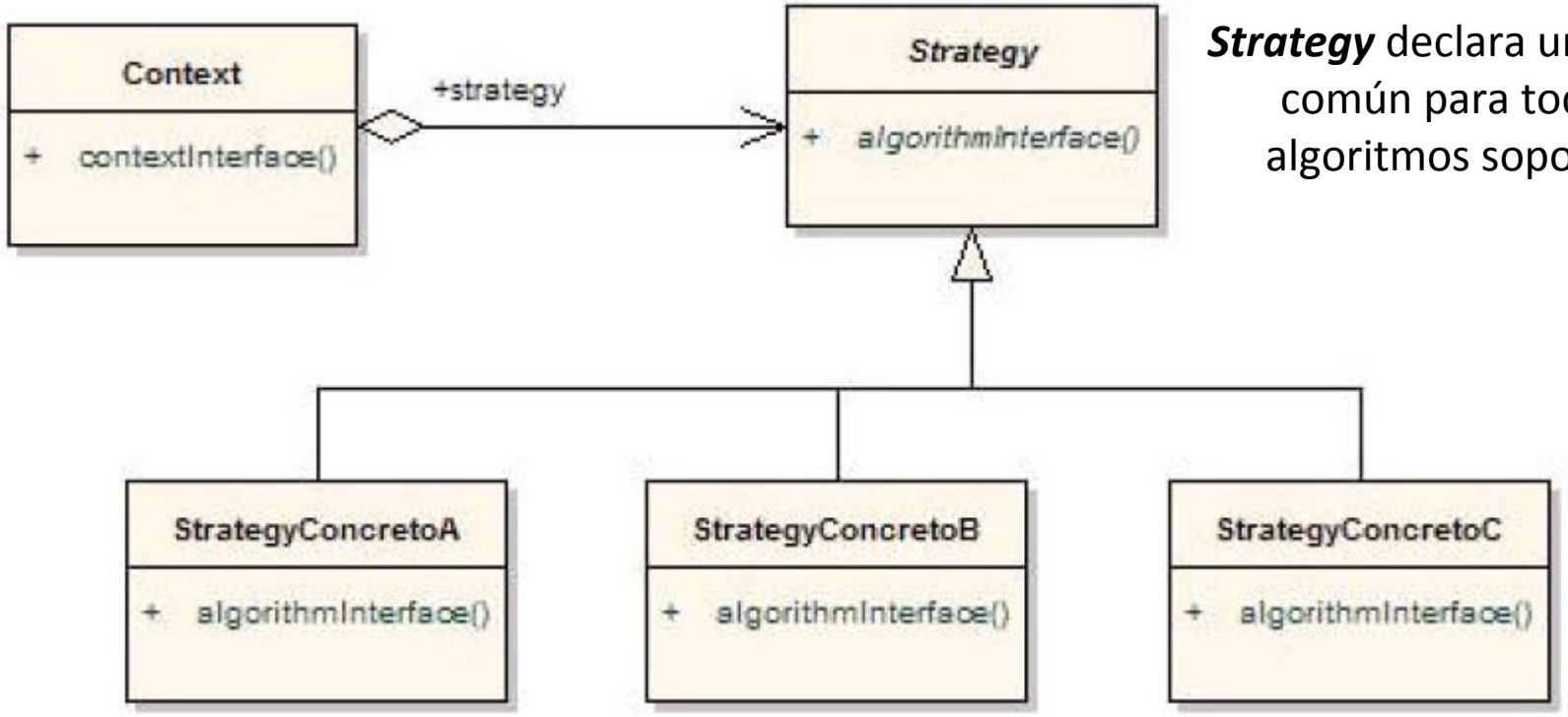
- Encapsula un algoritmo completo ignorando los detalles de su implementación, permitiendo intercambiarlo en tiempo de ejecución para permitir actuar a la clase cliente con un comportamiento distinto.

Participantes !

Patrón Strategy (Ejemplo de la Web)

Permite realizar un cambio de estrategia (el algoritmo) en tiempo de ejecución

El objeto **Context** recibe como parámetro (una interfaz o una clase abstracta) que se llama ***Strategy*** y de esta clase abstracta ***Strategy*** heredan los estados propiamente dicho (estados concretos que contienen el comportamiento que realiza el objeto acorde al "algoritmo" implementado)



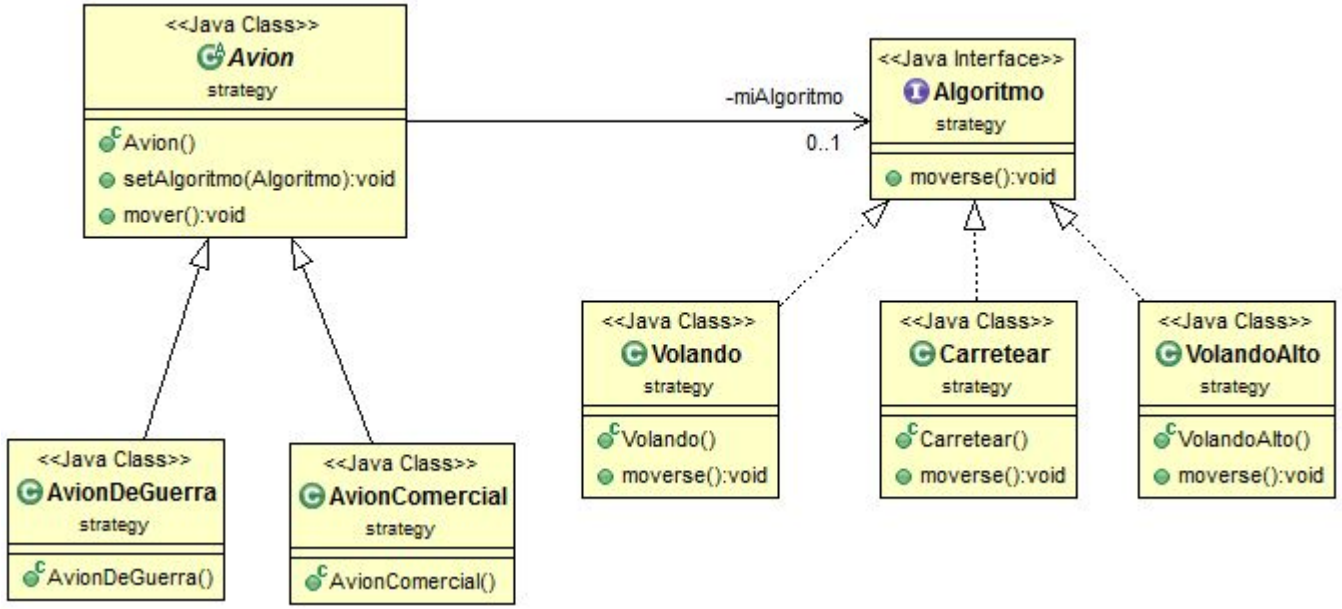
Strategy declara una interfaz común para todos los algoritmos soportados.

Patrón Strategy (Ejemplo de la Web)

Permite realizar un cambio de estrategia (el algoritmo) en tiempo de ejecución

AvionDeGuerra
AvionComercial
…son los contextos. Extienden de la clase **Avion**.

Cada contexto puede usar diferentes algoritmos en tiempo de ejecución.



La interfaz **Algoritmo** tiene un método **moverse** pero NO proporciona la implementación concreta de este método.

Volando
Carretear
VolandoAlto
Son los algoritmos (tienen la implementación concreta) implementan la interfaz **Algoritmo**

La interfaz **Algoritmo** debe implementarse en todos los algoritmos necesarios para mover los aviones.

Patrón Strategy (Ejemplo de la Web)

Permite realizar un cambio de estrategia (el algoritmo) en tiempo de ejecución

```
public class Carretear implements Algoritmo{

    public void moverse(){
        System.out.println("...andando por la pista");
    }

}
```

```
public class Volando implements Algoritmo {

    public void moverse(){
        System.out.println("...andando por el aire...");
    }
}
```

```
public class VolandoAlto implements Algoritmo {

    public void moverse(){
        System.out.println("volando a muy alta velocidad");
    }

}
```

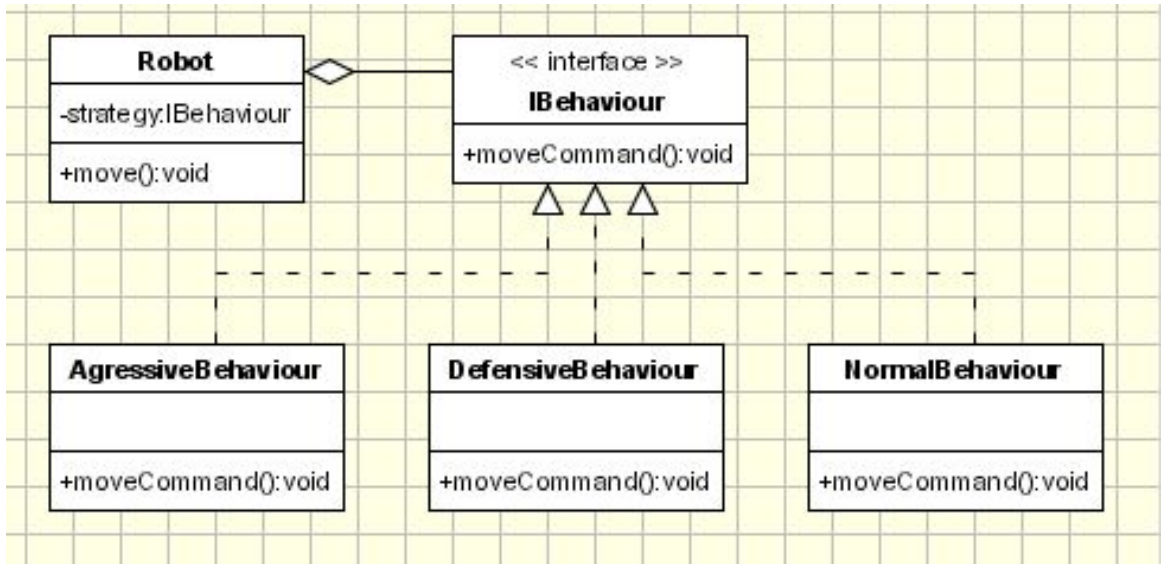
```
public abstract class Avion {
    private Algoritmo miAlgoritmo;

    public void setAlgoritmo(Algoritmo a){
        this.miAlgoritmo = a;
    }

    public void mover(){
        this.miAlgoritmo.moverse();
    }

}
```

```
public class StrategyTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AvionComercial avionComercial = new AvionComercial();
        AvionDeGuerra avionDeGuerra = new AvionDeGuerra();

        avionComercial.setAlgoritmo(new Carretear());
        avionComercial.mover();

        avionDeGuerra.setAlgoritmo(new Carretear());
        avionDeGuerra.mover();
        avionDeGuerra.setAlgoritmo(new VolandoAlto());
        avionDeGuerra.mover();


    }

}
```

Patrón de Comportamiento: **Strategy**

Patrón Strategy (otro ejemplo de la Web)

Permite realizar un cambio de estrategia (el algoritmo) en tiempo de ejecución



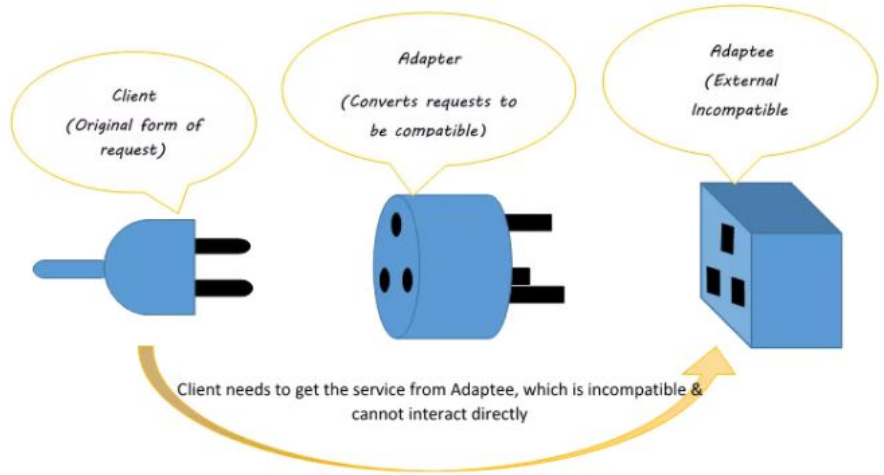
Fuente:
http://www.oodesign.com/strategy-pattern.html

Patrón de Comportamiento: **State**

Patrón Adapter

Busca una manera estandarizada de adaptar un objeto a otro.

Propósito		
De Creación	**Estructural**	De Comportamiento
Factory Method	Adapter	**Interpreter**
		Template Method
Abstract Factory	Adapter	**Chain of Responsability**
Builder	Bridge	**Command**
Prototype	Composite	**Iterator**
Singleton	Decorator	**Mediator**
	Facade	**Memento**
	Flyweight	**Observer**
	Proxy	**State**
		Strategy
		Visitor



Patrón Estructural: **Adapter**

Patrón Adapter

- El patrón estructural Adapter permite convertir la interfaz de una clase en otra que es la que esperan los clientes.

- Permite que trabajen juntas clases que de otro modo no podrían por tener interfaces incompatibles, dicho de otra manera, sirve para hacer que dos interfaces, en principio diferentes, puedan comunicarse.

Patrón Adapter

El patrón Adapter se puede aplicar cuando:

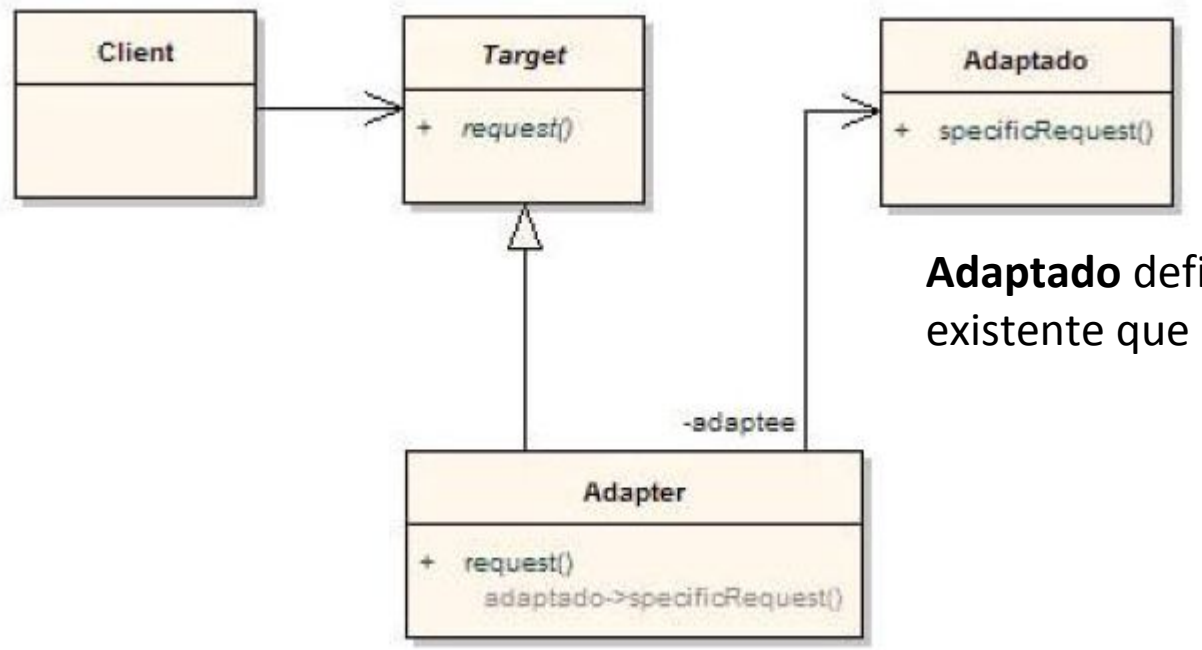
- Queremos usar una clase existente, y ésta no tiene la interfaz que necesitamos.
- Queremos crear una clase reutilizable que coopere con clases con las que no está relacionada. Por tanto, que no tendrán interfaces compatibles.
- Motiva el uso de este patrón cuando no es posible modificar la clase original (hay situaciones en las que no voy a poder cambiarla, por ejemplo librerías externas).

Participantes !

Patrón Adapter

Busca una manera estandarizada de adaptar un objeto a otro.

Target define la interfaz específica del dominio que Cliente usa.



Adaptado define una interface existente que necesita adaptarse.

Adapter adapta la interfaz **Adaptado** a la interfaz ***Target***

Para solucionarlo añadiremos un adaptador intermedio **Adapter**, que se encargará de realizar la conversión de una *interface* a otra (**Adaptado** a **Target**)

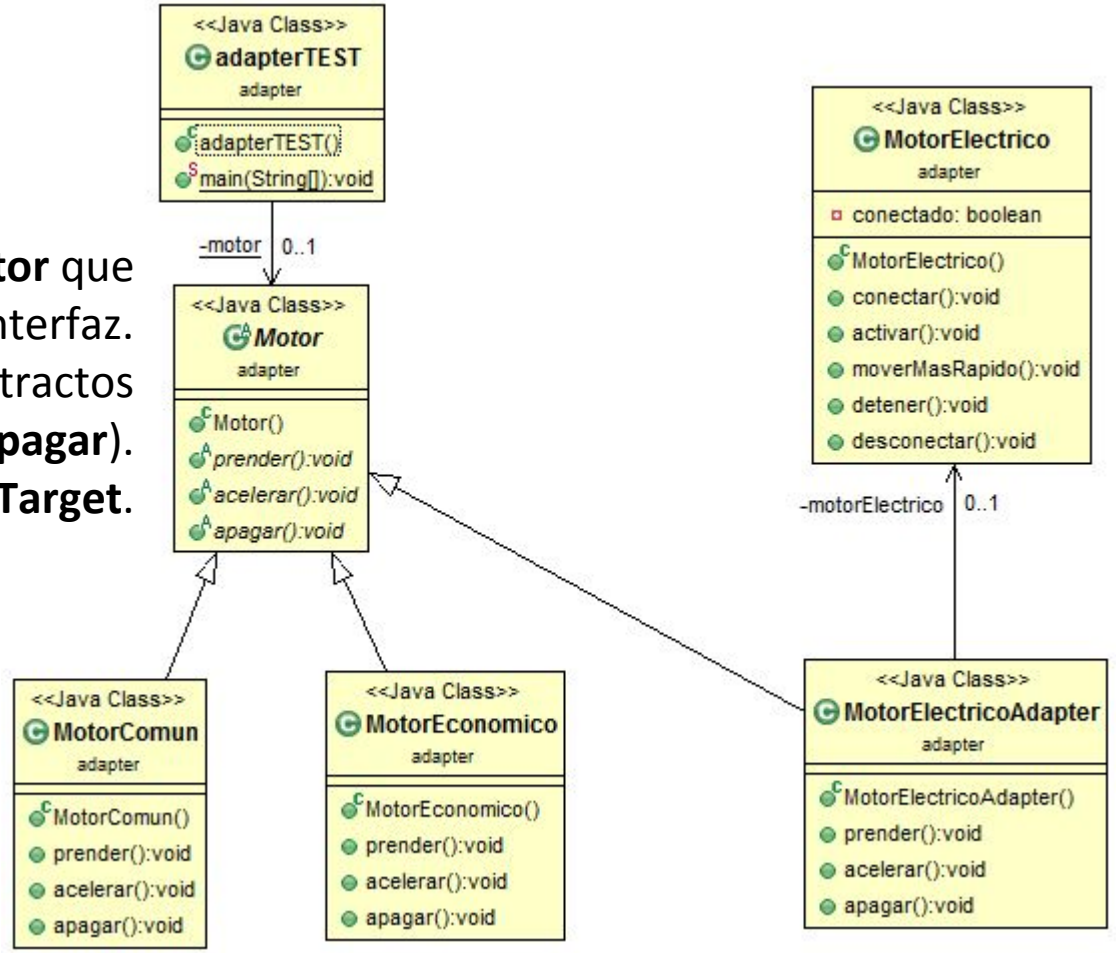
Patrón Estructural: **Adapter**

Patrón Adapter

Busca una manera estandarizada de adaptar un objeto a otro.

La Clase Abstracta **Motor** que bien podría ser una interfaz. Tiene 3 métodos abstractos (**prender**, **acelerar** y **apagar**). Motor es la clase **Target**.

Acá tenemos que poner los métodos que más tarde queremos "adaptar" y tienen que ser abstractos.



MotorElectrico tiene una interfaz distinta, tiene otros métodos con nombres distintos:
conectar()
activar()
moverMasRapido()
detener()
desconectar()

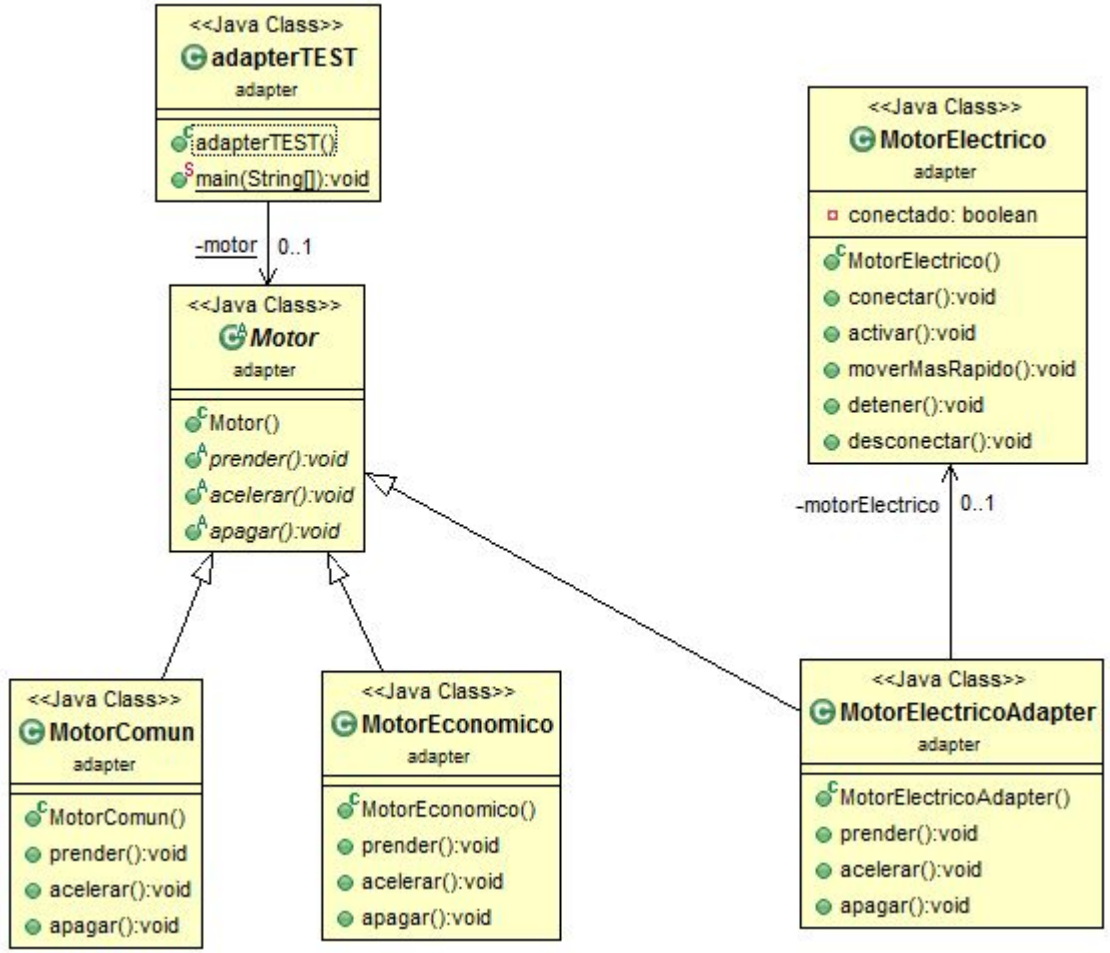
Entonces no pueden heredar de motor. ¿Cómo se resuelve? Con un **Adaptador**.

Motor es extendida por 3 clases hijas: **MotorComun**, **Motor económico** y **MotorElectricoAdapter.**
MotorComun y **MotorEconomico** tendrán el código de los métodos abstractos definidas en **Motor**.

Patrón Estructural: **Adapter**

Patrón Adapter

Busca una manera estandarizada de adaptar un objeto a otro.



MotorElectricoAdapter es la clase "puente" entre la clase que se va a adaptar **MotorElectrico** y el **Target Motor** (que NO es posible cambiar).

MotorElectrico tiene métodos que no son compatibles con el **Target** (**Motor**).

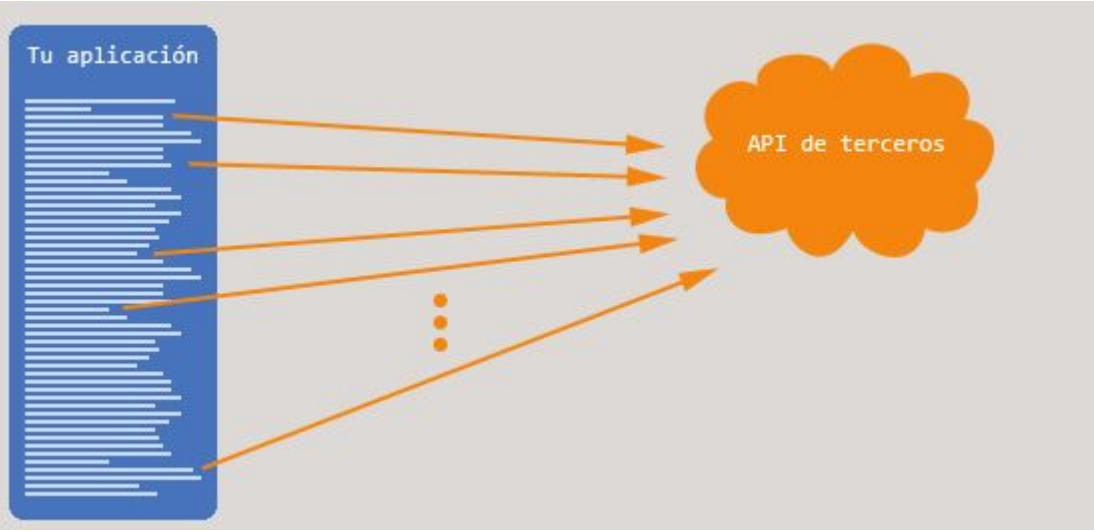
MotorElectrico nunca podría heredar de motor simplemente porque tiene métodos diferentes.

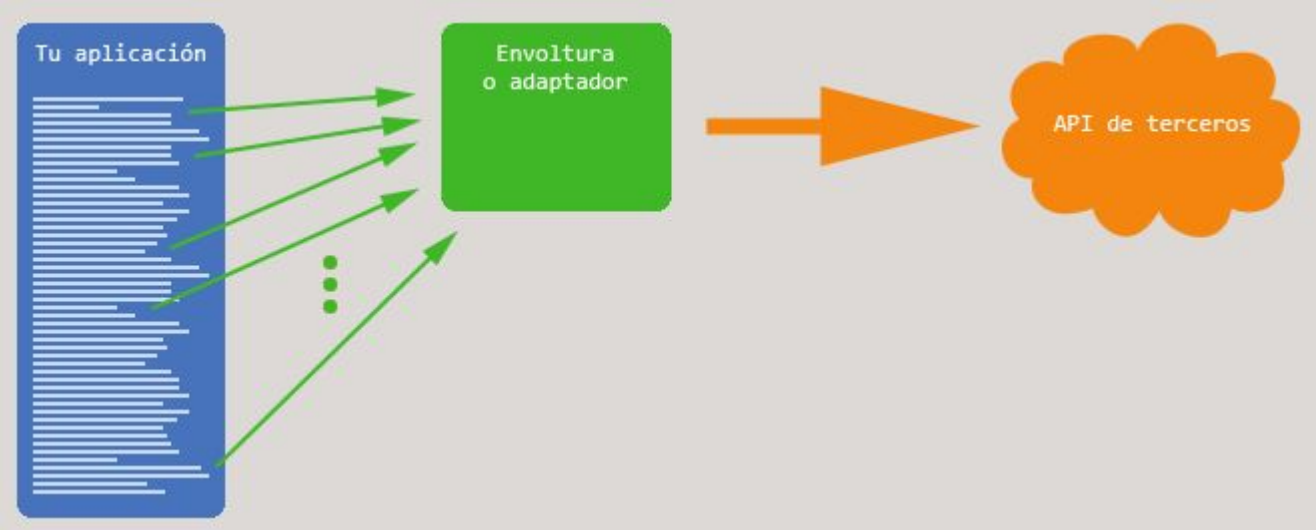
Entonces **MotorElectrico** (**Adaptee** o **Adaptado**) es adaptada a través de la clase **Adapter** (en este caso **MotorElectricoAdapter**) según lo que definimos en el **Target** (esos métodos abstractos).

Patrón Estructural: **Adapter**

Patrón Adapter

Busca una manera estandarizada de adaptar un objeto a otro.





Patrón Estructural: **Adapter**

Patrón Adapter (otro ejemplo)

Busca una manera estandarizada de adaptar un objeto a otro.