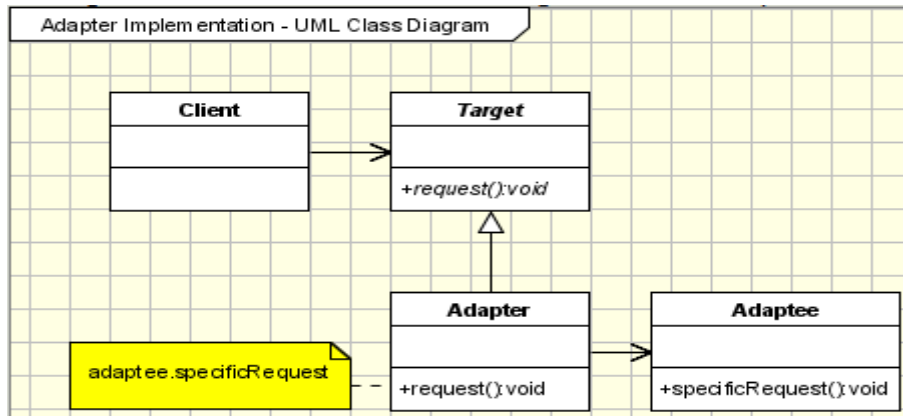


Adapter

Diagrama de clases e interfaces



Intención

- Convertir la interfaz de una clase en otra que esperan las clases cliente. El patrón Adaptador permite la colaboración de clases que de otra manera no sería posible debido a la incompatibilidad de sus interfaces.
- Envolver una clase existente con una nueva interfaz.

Motivación

El patrón estructural Adaptador (Adapter), también conocido como Envoltorio (wrapper) se puede utilizar en muchos contextos en los que se necesita utilizar una clase cuya interfaz pública no se adapta a nuestras necesidades. Un caso habitual es cuando se trata de integrar librerías o APIs de terceros (sobre las que no tenemos control) con nuestras librerías y nuestras clases cliente.

Tanto el nombre de adaptador como el de envoltorio tienen bastante sentido. Pensemos en la siguiente cuestión. Nos vamos de viaje a una isla caribeña. Por supuesto nos llevaremos el teléfono móvil, así como el cargador. ¿Podremos cargarlo conectándolo a los enchufes de ese país?

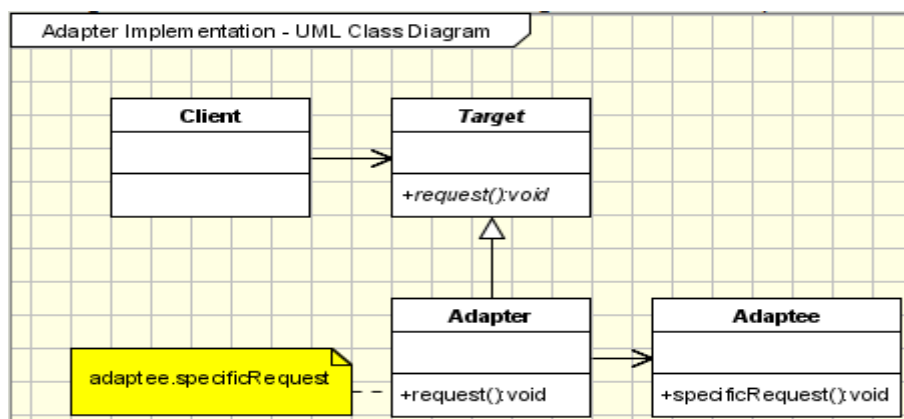


La respuesta es que probablemente no podremos. Los enchufes allí y en otros países sólo aceptan clavijas de conectores planos y los de nuestro cargador son redondos.

Para resolver el problema de nuestro móvil, está claro que necesitaremos un adaptador. Esta pieza es la metáfora del patrón adaptador que vamos a desarrollar en este documento.

Implementación

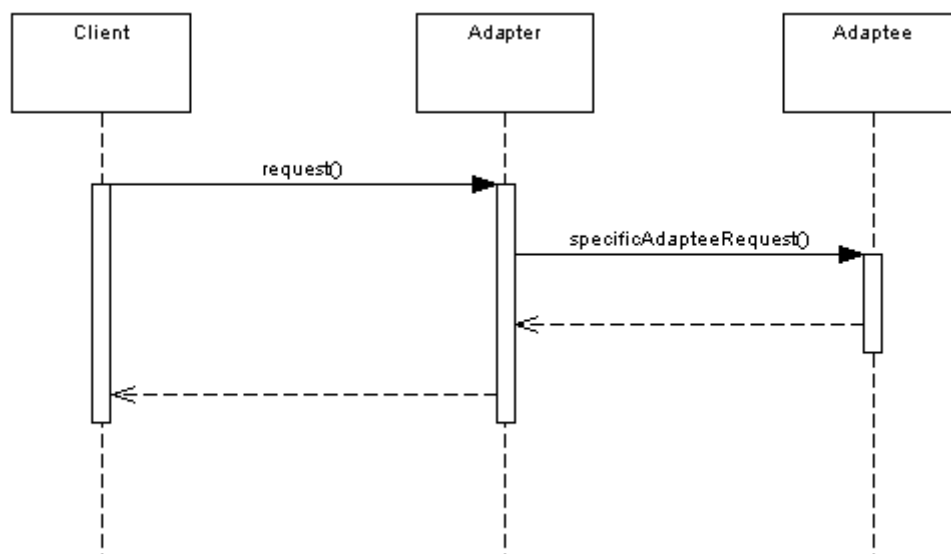
El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



Las clases participantes en el patrón son las siguientes:

- **Target (Objetivo):** Se trata de la interfaz específica del dominio que utilizan las clases cliente.
- **Client (Cliente):** Necesita utilizar objetos que se ajustan a la interfaz Target.
- **Adaptee (Adaptada):** Es la clase que necesita ser adaptada de alguna forma.
- **Adapter (Adaptador):** Adapta la interfaz Adaptee a la interfaz Target.

El diagrama de secuencias del patrón Adaptador es muy sencillo:



Aplicabilidad y Ejemplos

Un componente considerado ajeno a una arquitectura determinada (un contexto), ofrece una funcionalidad interesante que se necesita utilizar desde el contexto. El problema es que la "visión del mundo" del componente no es compatible con la filosofía y la arquitectura del sistema actualmente en desarrollo. La solución es el patrón Adaptador, ya que crea una abstracción que hace corresponder un componente ajeno (o antiguo) con el nuevo sistema. Los clientes llaman a los métodos del Adaptador, el cual redirige estas llamadas al componente ajeno (adaptado). Esta estrategia puede ser implementada mediante Herencia o con Delegación.

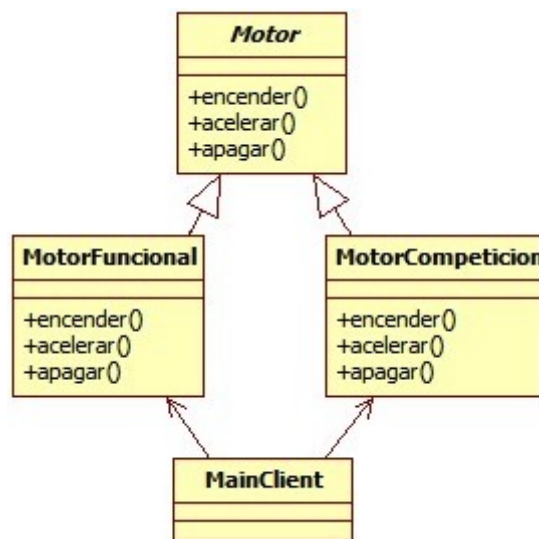
El patrón Adaptador funciona como un envoltorio o modificador de una clase existente, ofreciendo una visión diferente o corregida de esa clase.

Por tanto, podemos utilizar este patrón cuando tenemos una clase (Client) que invoca los métodos definidos en una interfaz (Target) y por otro lado, tenemos una clase (Adaptee) que no implementa la interfaz pero que dispone de unas operaciones que necesitan ser utilizadas por la primera clase (Client) a través de la interfaz. No podemos modificar nada del código existente (supongamos que sólo tenemos los binarios de Adaptee (Adaptada)). Solución: podemos crear una clase Adapter (Adaptador) que implemente la interfaz y que haga de puente entre la clase Client y la clase Adaptee (Adaptada).

Ejemplo 1 - Simulador de motores

Supongamos que trabajamos en una empresa de software en la que hemos desarrollado un simulador de motores de diferente cilindrada, todos basados en gasolina. Nuestro software consiste de una API muy potente y de diferentes programas (clases cliente) que la emplean.

El siguiente diagrama muestra las clases que intervienen en nuestra aplicación (tened en cuenta que no se utiliza el patrón Adaptador, ya que no se necesita):



Comencemos por ver nuestro software (que no implementa el patrón Adaptador, porque no lo requiere).

Nuestro API tiene una clase abstracta llamada Motor, cuyo código queda como sigue:

Motor.java

```
package estructurales.adapter;

public abstract class Motor {
    abstract public void encender();
    abstract public void acelerar();
    abstract public void apagar();
    // ...mas metodos, con codigo para heredar muy interesante
}
```

Tenemos varias clases que heredan de la clase abstracta anterior. No obstante, para el ejemplo sólo veremos dos.

La clase MotorFuncional, caracterizada por dar un buen rendimiento y consumir poco combustible.

MotorFuncional.java

```
package estructurales.adapter;

public class MotorFuncional extends Motor {

    public MotorFuncional(){
        System.out.println("Creando una instancia de motor funcional");
    }

    @Override
    public void encender() {
        System.out.println("motor funcional encendido");
    }

    @Override
    public void acelerar() {
        System.out.println("motor funcional acelerado");
    }

    @Override
    public void apagar() {
        System.out.println("motor funcional apagado");
    }

}
```

La clase MotorCompeticion, caracterizada por proporcionar un excelente rendimiento bajo las condiciones más exigentes.

MotorCompeticion.java

```
package estructurales.adapter;

public class MotorCompeticion extends Motor {

    public MotorCompeticion(){
        System.out.println("Creando una instancia de motor competicion");
    }

    @Override
    public void encender() {
        System.out.println("motor competicion encendido");
    }

    @Override
    public void acelerar() {
        System.out.println("motor competicion acelerado");
    }

    @Override
    public void apagar() {
        System.out.println("motor competicion apagado");
    }

}
```

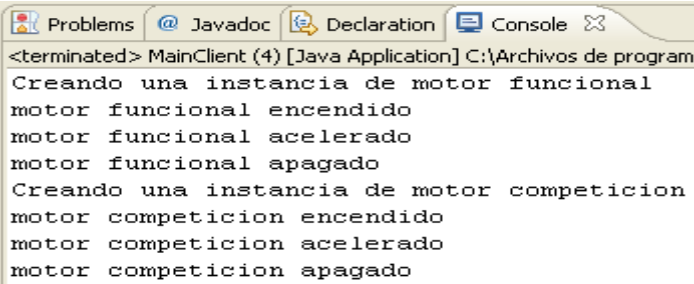
```
}
```

Veamos ahora un programa de simulación que utiliza nuestro API.

MainClient.java

```
package estructurales.adapter;  
  
public class MainClient {  
    public static void main(String[] args) {  
        Motor motor = new MotorFuncional();  
        motor.encender();  
        motor.acelerar();  
        motor.apagar();  
  
        motor = new MotorCompeticion();  
        motor.encender();  
        motor.acelerar();  
        motor.apagar();  
    }  
}
```

Salida:



The screenshot shows a Java IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the MainClient program. The output is as follows:

```
<terminated> MainClient (4) [Java Application] C:\Archivos de program  
Creando una instancia de motor funcional  
motor funcional encendido  
motor funcional acelerado  
motor funcional apagado  
Creando una instancia de motor competicion  
motor competicion encendido  
motor competicion acelerado  
motor competicion apagado
```

Por otro lado, nos informan de que hemos establecido un acuerdo de colaboración con una empresa que también se dedica al desarrollo de simuladores de motores pero que está especializada en simuladores de motores eléctricos. Nuestro jefe nos pregunta si es posible que nuestros programas utilicen el API de motores eléctricos. Aquí es cuando entra el juego el patrón adaptador.

El API de motores eléctricos tiene una interfaz parecida a la nuestra, pero los nombres de los métodos son diferentes. Por ejemplo, nosotros tenemos un método llamado `acelerar()` y ellos uno parecido, denominado `aumentarVelocidad()`. Nosotros tenemos uno llamado `encender()` y ellos otro parecido denominado `activar()`. Además, hay dos restricciones con los motores eléctricos:

- Para poder activar un motor eléctrico (ponerlo a funcionar a ralentí) o acelerarlo, es necesario que previamente se haya invocado al método conectar().
- Para poder detener un motor eléctrico, previamente se tiene que haber llamado al método desconectar().

La empresa de los motores eléctricos nos hace llegar la interfaz de su API.

MotorElectrico.java

```
package estructurales.adapter;

public interface MotorElectrico {
    public void conectar();
    public void activar();
    public void aumentarVelocidad();
    public void detener();
    public void desconectar();
}
```

También nos ha enviado el código fuente de una clase llamada MotorElectricoImpl que implementa la interfaz MotorElectrico.

MotorElectricoImpl.java

```
public class MotorElectricoImpl implements MotorElectrico {

    private boolean conectado = false;

    public MotorElectricoImpl() {
        System.out.println("Creando instancia de motor electrico");
    }

    @Override
    public void conectar() {
        System.out.println("motor electrico conectado");
        this.conectado = true;
    }

    @Override
    public void activar() {
        if (!conectado) {
            System.out
                .println("No se puede activar porque no esta
conectado el motor electrico");
        } else {
            System.out.println("motor electrico activado");
        }
    }
}
```

```

@Override
public void aumentarVelocidad() {
    if (!conectado) {
        System.out.println("No se puede acelerar el motor
electrico porque no esta conectado");
    } else {
        System.out.println("motor electrico acelerado");
    }
}

@Override
public void detener() {
    if (!conectado) {
        System.out.println("No se puede detener motor electrico
porque no esta conectado");
    } else {
        System.out.println("motor electrico detenido");
    }
}

@Override
public void desconectar() {
    System.out.println("motor electrico desconectado");
    conectado = false;
}
}

```

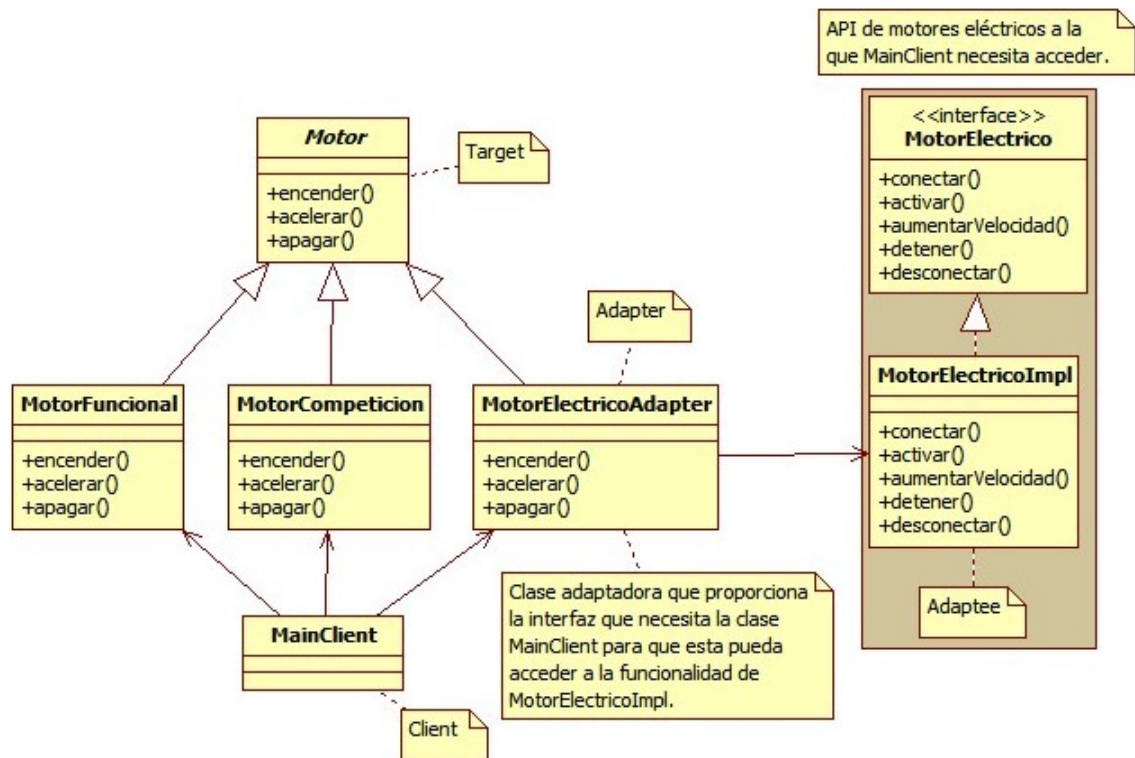
Como vemos, este motor realiza operaciones muy parecidas a las de nuestros motores, pero de distinta manera y con llamadas diferentes. ¿Cómo hacemos para integrar MotorElectricoImpl al resto de nuestro sistema?

Respuesta: necesitamos un adaptador. El adaptador envolverá al objeto “extraño” para hacerlo familiar a nuestras clases clientes.

Nota:

Hemos de tener en mente que no es necesario disponer del código fuente de la clase MotorElectricoImpl para utilizar el patrón Adaptador, ya que con el código binario tendríamos suficiente. En otras palabras, no tenemos porqué conocer los detalles de implementación, tan sólo la interfaz que da acceso a esa funcionalidad. Al fin y al cabo, somos expertos en motores de gasolina y no en motores eléctricos.

El siguiente diagrama de clases muestra el resultado de aplicar el patrón Adaptador a nuestro ejemplo:



La estrategia seguida en este ejemplo para el patrón Adaptador es la Delegación (La clase Adaptador define un atributo del tipo Adaptada). Existe otra variante donde se utiliza la Herencia (veremos más sobre esto al final del documento).

El código para el adaptador queda como sigue:

MotorElectricoAdapter.java

```

package estructurales.adapter;

public class MotorElectricoAdapter extends Motor {

    MotorElectrico motorE;

    public MotorElectricoAdapter() {
        motorE = new MotorElectricoImpl();
        System.out
            .println("Creando una instancia del adaptador del motor
                electrico");
    }

    @Override
    public void encender() {
        motorE.conectar();
        motorE.activar();
    }
}
  
```

```

@Override
public void acelerar() {
    motorE.aumentarVelocidad();
}

@Override
public void apagar() {
    motorE.detener();
    motorE.desconectar();
}
}

```

Como vemos, el adaptador no sólo se encarga de corregir los nombres de los métodos, sino también de controlar aspectos como la conexión y desconexión del motor en la operación adecuada, es decir, de las variaciones que se tengan que hacer para que la API de motores eléctricos sea transparente para el código cliente. Lo más importante es que ahora podemos utilizar esta implementación de Motor en nuestro sistema.

Por ejemplo podemos hacer cosas como esta:

MainClient.java (versión extendida)

```

package estructurales.adapter;

public class MainClient {

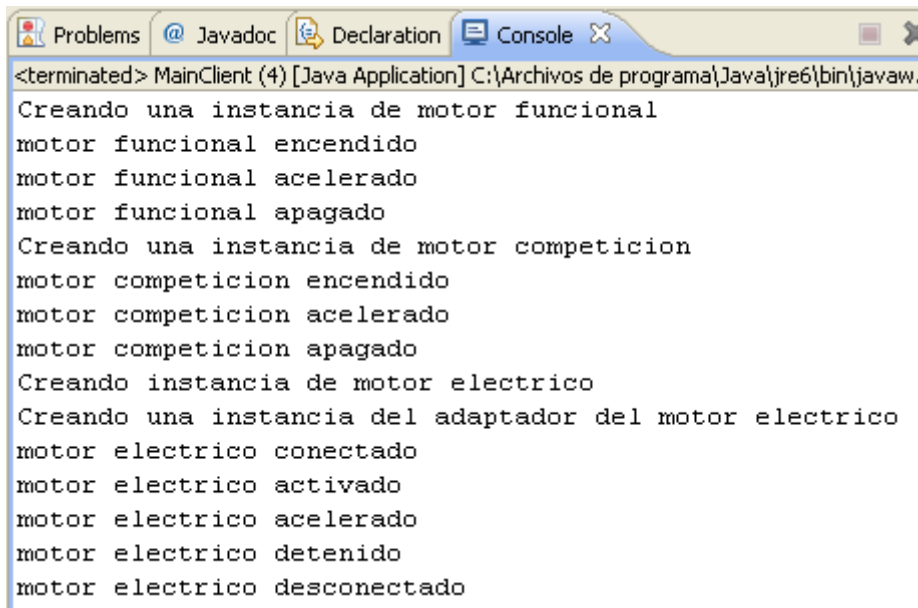
    public static void main(String[] args) {
        Motor motor = new MotorFuncional();
        motor.encender();
        motor.acelerar();
        motor.apagar();

        motor = new MotorCompeticion();
        motor.encender();
        motor.acelerar();
        motor.apagar();

        motor = new MotorElectricoAdapter();
        motor.encender();
        motor.acelerar();
        motor.apagar();
    }
}

```

Salida:



```
<terminated> MainClient (4) [Java Application] C:\Archivos de programa\Java\jre6\bin\javaw.  
Creando una instancia de motor funcional  
motor funcional encendido  
motor funcional acelerado  
motor funcional apagado  
Creando una instancia de motor competicion  
motor competicion encendido  
motor competicion acelerado  
motor competicion apagado  
Creando instancia de motor electrico  
Creando una instancia del adaptador del motor electrico  
motor electrico conectado  
motor electrico activado  
motor electrico acelerado  
motor electrico detenido  
motor electrico desconectado
```

Problemas específicos e implementación

Adaptadores de Objetos - Patrón Adaptador que usa la técnica de Delegación

También conocido como *Adaptador de Objetos*, es la forma más habitual en la que se presenta el patrón Adaptador (es la utilizada en el ejemplo visto de los motores). Se utiliza la técnica de la Composición (Delegación), por lo que la clase Adapter tiene visibilidad de atributo sobre la clase Adaptee, esto, Adaptee es un atributo de Adapter en el que delega las llamadas efectuadas por Client.

Esto es muy diferente a lo que hacen los adaptadores que utilizan la Herencia, también conocidos como *Adaptadores de Clase*, que extienden a Adaptee para obtener por Herencia su implementación.

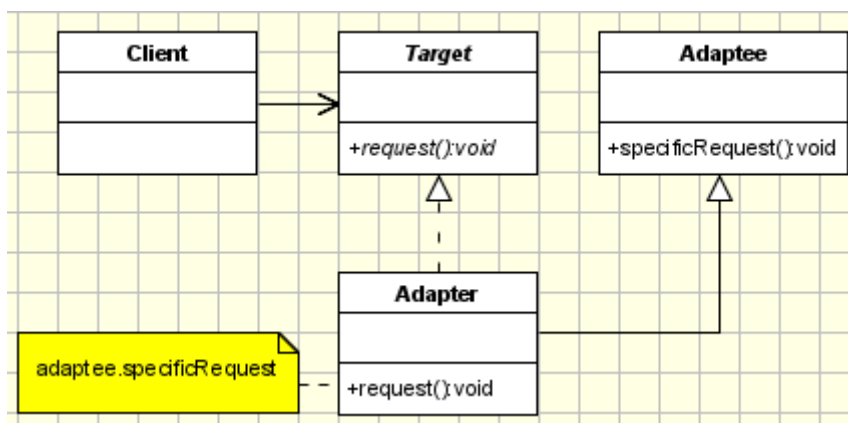
Los *Adaptadores de Objetos* presentan algunas ventajas sobre los *Adaptadores de Clase*. La principal ventaja es que el Adapter adapta tanto al Adaptee como a las subclases que este tenga. No obstante, con una restricción: todas las subclases que no añadan nuevos métodos a la interfaz pública de Adaptee. Esta limitación la impone el propio mecanismo de la Delegación (no puede conocer las existencia de otros métodos que no formen parte de la interfaz de Adaptee). Por tanto, un nuevo método en alguna subclase de Adaptee obligaría a modificar el Adapter o a subclasificarlo para contemplar ese nuevo método.

La principal desventaja de los *Adaptadores de Objetos* es la inherente al uso de la Delegación (en contraposición a la Herencia): es necesario escribir el código que delega las peticiones en el Adaptee, lo cual a veces puede ser trivial y a veces puede ser complejo o farragoso.

Adaptadores de Clase - Patrón Adaptador que usa el mecanismo de Herencia

Los *Adaptadores de Clase* utilizan la Herencia en lugar de la Delegación. Esto significa que el Adapter, en lugar de delegar las llamadas en el Adaptee lo que hace es subclasificarlo, es decir, convertirse en su subclase. El Adapter también debe heredar del Target, pero como en Java no se admite la Herencia múltiple, lo que se hace es implementar la interfaz Target (en este caso Target debe ser una interfaz necesariamente, no puede ser una clase abstracta).

En el siguiente diagrama podemos ver claramente la idea de esta variante del patrón en la que el Adapter, en lugar de contener un atributo del tipo Adaptee lo que hace es extenderlo, pasando a formar parte así de su jerarquía de herencia. Por otro lado, el adapter implementa al Target, ya que es la manera de proporcionarle al Client la interfaz que necesita.



Esta variante presenta el inconveniente que si la clase Adaptee, una vez adaptada por el Adapter, es subclasificada por una clase de dominio, el adaptador no sabrá adaptar esa nueva subclase, por lo que se tendría que crear un adaptador específico para ella. Esto conlleva el peligro de tener que crear y mantener una estructura de clases paralelas.

Como ventaja respecto a la variante normal del patrón, se tiene que debido a la Herencia, el Adapter hereda toda la implementación del Adaptee, lo que supone un ahorro de código respecto al mecanismo de Delegación.

¿Cómo saber la responsabilidad (el trabajo) que le corresponde al Adapter?

Esta pregunta tiene una respuesta muy simple: el Adapter debe hacer lo necesario con el fin de adaptar al Adaptee. Suponiendo la variante normal del patrón, si Target y Adaptee son similares, entonces el Adapter tan sólo tiene que delegar en el Adaptee las peticiones procedentes del Client. En cambio, si Target y Adaptee no son similares, entonces el Adapter podría tener que transformar las estructuras de datos entre estos e implementar el código necesario para realizar correctamente las peticiones al Adaptee.

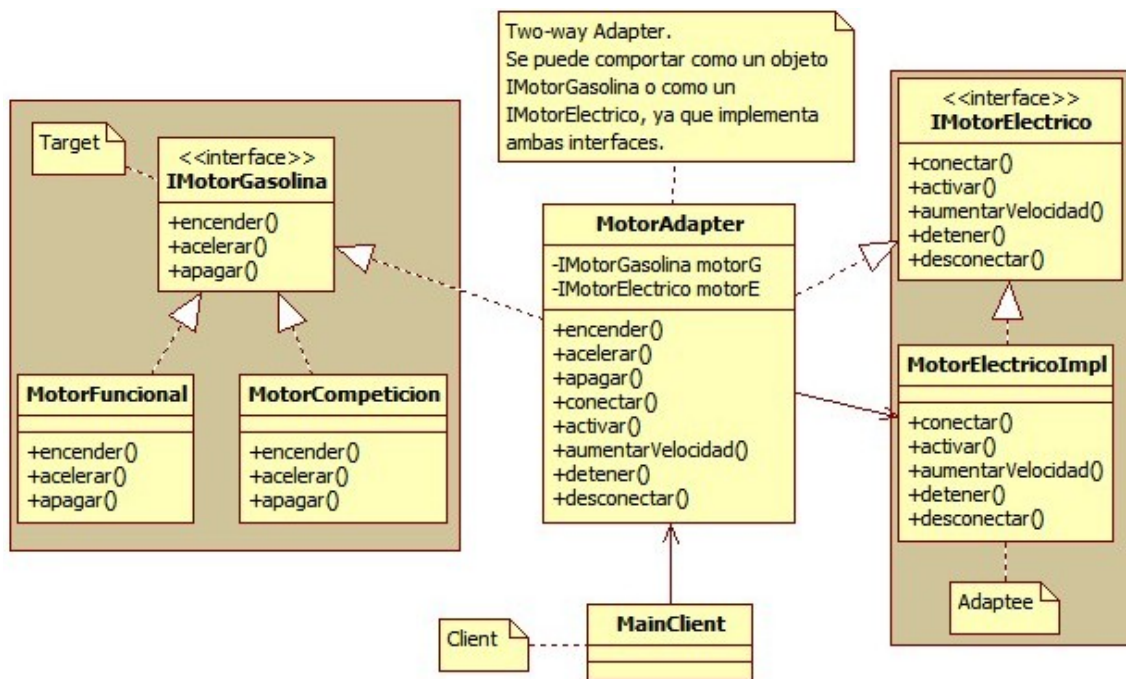
Adaptadores de dos vías (Two-way adapters)

Supongamos que un sistema dispone de una interfaz antigua y también de una interfaz nueva. Asumamos que las clases cliente necesitan acceder tanto a la interfaz antigua como a la nueva. ¿Nos puede ayudar aquí el patrón Adaptador?

Sí, mediante una variante conocida como *Adaptador de dos vías*. Estos adaptadores implementan las interfaces de las dos caras del problema con el objetivo de integrarlas. Es decir, un adaptador de este tipo implementa la interfaz Target y también la interfaz Adaptee.

El *Adaptador de dos vías* nos permite mediante una sola clase adaptadora acceder a dos objetos diferentes.

La siguiente figura muestra el diagrama obtenido para el ejemplo del simulador de motores.



Hay que aclarar algunos cambios respecto a la versión anterior:

- Se ha suprimido la anterior clase abstracta Motor a favor de la interfaz IMotorGasolina. Realmente, la clase abstracta Motor podría perfectamente haber sido una interfaz desde el principio, pues no incluía un solo método no abstracto, por lo que parece más adecuado sustituirla por una interfaz, además de vernos mejor una interfaz para esta variante del patrón.
- MotorFuncional y MotorCompeticion ahora implementan la interfaz IMotorGasolina, dado que ahora no existe la clase abstracta Motor.
- Se ha renombrado la interfaz MotorElectrico a IMotorElectrico. De esta manera resalta el hecho de que tenemos dos interfaces, cada una para una familia de clases.
- La clase adaptadora MotorElectricoAdapter ahora no existe. En su lugar tenemos a MotorAdapter: el adaptador de dos vías.

Veamos el código. Comenzamos por la jerarquía de los motores de gasolina:

IMotorGasolina.java

```

package estructurales.adapter.dosvias;

public interface IMotorGasolina {
    abstract public void encender();
}

```

```

        abstract public void acelerar();
        abstract public void apagar();
    }

```

MotorFuncional.java

```

package estructurales.adapter.dosvias;

public class MotorFuncional implements IMotorGasolina {

    public MotorFuncional(){
        System.out.println("Creando una instancia de motor
funcional");
    }

    @Override
    public void encender() {
        System.out.println("motor funcional encendido");
    }

    @Override
    public void acelerar() {
        System.out.println("motor funcional acelerado");
    }

    @Override
    public void apagar() {
        System.out.println("motor funcional apagado");
    }

}

```

MotorCompeticion.java

```

package estructurales.adapter.dosvias;

public class MotorCompeticion implements IMotorGasolina {

    public MotorCompeticion(){
        System.out.println("Creando una instancia de motor
competicion");
    }

    @Override
    public void encender() {
        System.out.println("motor competicion encendido");
    }

    @Override
    public void acelerar() {
        System.out.println("motor competicion acelerado");
    }

    @Override
    public void apagar() {
        System.out.println("motor competicion apagado");
    }

}

```

```
}
```

Veamos ahora la jerarquía de los motores eléctricos:

IMotorElectrico.java

```
package estructurales.adapter.dosvias;
```

```
public interface IMotorElectrico {  
    public void conectar();  
    public void activar();  
    public void aumentarVelocidad();  
    public void detener();  
    public void desconectar();  
}
```

MotorElectricoImpl.java

```
package estructurales.adapter.dosvias;
```

```
public class MotorElectricoImpl implements IMotorElectrico {  
  
    private boolean conectado = false;  
  
    public MotorElectricoImpl() {  
        System.out.println("Creando instancia de motor electrico");  
    }  
  
    @Override  
    public void conectar() {  
        System.out.println("motor electrico conectado");  
        this.conectado = true;  
    }  
  
    @Override  
    public void activar() {  
        if (!conectado) {  
            System.out  
                .println("No se puede activar porque no esta  
conectado el motor electrico");  
        } else {  
            System.out.println("motor electrico activado");  
        }  
    }  
  
    @Override  
    public void aumentarVelocidad() {  
        if (!conectado) {  
            System.out.println("No se puede acelerar el motor  
electrico porque no esta conectado");  
        } else {  
            System.out.println("motor electrico acelerado");  
        }  
    }  
}
```



```

@Override
public void detener() {
    if (!conectado) {
        System.out.println("No se puede detener motor electrico
porque no esta conectado");
    } else {
        System.out.println("motor electrico detenido");
    }
}

@Override
public void desconectar() {
    System.out.println("motor electrico desconectado");
    conectado = false;
}
}

```

MotorAdapter.java

La clase tiene dos constructores, ambos privados y con un solo argumento. Un constructor tiene como parámetro un objeto IMotorGasolina y el otro un IMotorElectrico. Por otro lado, la clase dispone de dos atributos, uno del tipo IMotorGasolina y otro del tipo IMotorElectrico. Cada uno de estos atributos se inicializará cuando se ejecute el constructor cuyo parámetro coincide con su tipo.

Dado que no hay ningún constructor público, se ha creado un método de factoría estático llamado crear() para que las clases cliente puedan crear un objeto adaptador de dos vías. Así, las clases cliente deben pasar un String que representa el tipo de motor con el que quieren trabajar: "gasolinaF" (funcional), "gasolinaC" (Competicion) y "electrico". Cuando el método de factoría es invocado, éste llama al constructor privado que corresponda y el método de factoría estático retorna una instancia de MotorAdapter.

Notad que de esta manera, una clase cliente tiene tres opciones para recoger el objeto MotorAdapter devuelto por el método de factoría estático:

- Utilizar una referencia del tipo IMotorGasolina. Esto es lo adecuado cuando el cliente quiere trabajar con este tipo de motores.
- Utilizar una referencia del tipo IMotorElectrico. Esto es lo adecuado cuando el cliente quiere trabajar con este tipo de motores.

- Utilizar una referencia del tipo MotorAdapter (no recomendado). Esto puede ser problemático, ya que a pesar de que la referencia nos dejará acceder a todos los métodos de MotorAdapter, la verdad es que sólo funcionarán los del objeto que realmente nos haya devuelto el método de factoría estático, lo cual depende del String que se le haya pasado: "gasolinaF" (funcional), "gasolinaC" (Competicion) y "electrico".

MotorAdapter.java

```
package estructurales.adapter.dosvias;

public class MotorAdapter implements IMotorGasolina, IMotorElectrico {

    private IMotorGasolina motorG;
    private IMotorElectrico motorE;

    private MotorAdapter(IMotorGasolina motorG_) {
        motorG = motorG_;
        System.out.println("Creando una instancia del adaptador
para el motor de gasolina");
    }

    private MotorAdapter(IMotorElectrico motorE_) {
        motorE = motorE_;
        System.out.println("Creando una instancia del adaptador
para el motor electrico");
    }

    public static MotorAdapter crear(String tipo) {
        if (tipo.equals("gasolinaF")) {
            return new MotorAdapter(new MotorFuncional());
        } else if (tipo.equals("gasolinaC")) {
            return new MotorAdapter(new MotorCompeticion());
        } else if (tipo.equals("electrico")) {
            return new MotorAdapter(new MotorElectricoImpl());
        } else {
            throw new RuntimeException("El tipo de motor
especificado no existe");
        }
    }

    /*
     * Metodos de MotorGasolina
     */

    @Override
    public void encender() {
        motorG.encender();
    }

    @Override
    public void acelerar() {
        motorG.acelerar();
    }
}
```

```

    }

    @Override
    public void apagar() {
        motorG.apagar();
    }

    /**
     * Metodos de motor electrico
     */
    @Override
    public void conectar() {
        motorE.conectar();
    }

    @Override
    public void activar() {
        motorE.activar();
    }

    @Override
    public void aumentarVelocidad() {
        motorE.aumentarVelocidad();
    }

    @Override
    public void detener() {
        motorE.detener();
    }

    @Override
    public void desconectar() {
        motorE.desconectar();
    }
}

```

MainClient.java

```

package estructurales.adapter.dosvias.client;

import estructurales.adapter.dosvias.IMotorElectrico;
import estructurales.adapter.dosvias.IMotorGasolina;
import estructurales.adapter.dosvias.MotorAdapter;

public class MainClient {

    public static void main(String[] args) {

        IMotorGasolina motorFuncional =
        MotorAdapter.crear("gasolinaF");
        motorFuncional.encender();
        motorFuncional.acelerar();
        motorFuncional.apagar();

        IMotorGasolina motorCompeticion =
        MotorAdapter.crear("gasolinaC");
    }
}

```

```

        motorCompeticion.encender();
        motorCompeticion.acelerar();
        motorCompeticion.apagar();

        IMotorElectrico motorElectrico =
MotorAdapter.crear("electrico");
        motorElectrico.conectar();
        motorElectrico.activar();
        motorElectrico.aumentarVelocidad();
        motorElectrico.detener();
        motorElectrico.desconectar();

        /*
        * Peligroso, no se recomienda utilizar una
        * referencia del tipo MotorAdapter: la llamada
        * a conectar() fallará porque hemos creado un
        * motor de gasolina y conectar() es un metodo
        * de los motores electricos.

        MotorAdapter todo = MotorAdapter.crear("gasolinaF");
        todo.encender();
        todo.conectar();*/
    }
}

```

Salida:

```

<terminated> MainClient (8) [Java Application] D:\java\jdk\jdk1.6.0_21\bin\javaw.exe (28/10/2011 21:
Creando una instancia de motor funcional
Creando una instancia del adaptador para el motor de gasolina
motor funcional encendido
motor funcional acelerado
motor funcional apagado
Creando una instancia de motor competicion
Creando una instancia del adaptador para el motor de gasolina
motor competicion encendido
motor competicion acelerado
motor competicion apagado
Creando instancia de motor electrico
Creando una instancia del adaptador para el motor electrico
motor electrico conectado
motor electrico activado
motor electrico acelerado
motor electrico detenido
motor electrico desconectado

```

Patrones relacionados

Bridge tiene una estructura similar a un objeto adaptador, sin embargo Bridge tiene un propósito diferente: mediar para separar una interfaz de su implementación, con la finalidad de que ambas partes evolucionen fácilmente y de forma autónoma. En cambio, un Adapter es un medio para cambiar la interfaz de un objeto existente.

Decorator es un patrón que extiende la funcionalidad de un objeto sin alterar su interfaz, por lo que un Decorator es más transparente de cara a una aplicación que un Adapter. Una consecuencia importante de esto, es que un Decorator admite sucesivas composiciones, lo cual es imposible para un Adapter.

El patrón Proxy define un representante o sustituto para otro objeto, al cual no modifica su interfaz.