UNIVERSIDAD PRIVADA DE TACNA

INGENIERIA DE SISTEMAS

TEMA:

Patrones de Diseño Estructurales

CURSO:

BASE DE DATOS II

DOCENTE(ING):

Patrick Jose Cuadros Quiroga

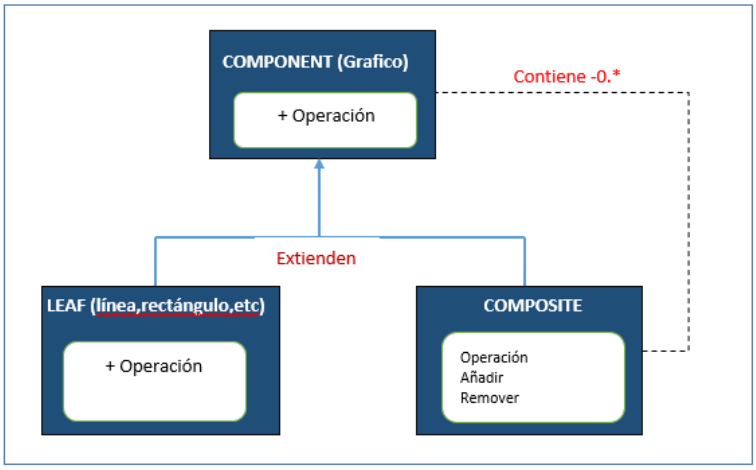
Integrantes:

Marko Antonio RIVAS RIOS	(2016055461)
Jorge Luis MAMANI MAQUERA	(2016055236)
Andree Ludwed VELASCO SUCAPUCA	(2016055286)
Yofer Nain CATARI CABRERA	(2017059289)
Adnner Sleyder ESPERILLA RUIZ	(2015050543)
Jesus ESCALANTE ALANOCA	(2015050641)

Índice

1. Patrones de Diseño Composite

El patrón de diseño Composite nos sirve para construir estructuras complejas partiendo de otras estructuras mucho más simples, dicho de otra manera, podemos crear estructuras compuestas las cuales están conformadas por otras estructuras más pequeñas. Para comprender mejor como funciona este patrón imaginemos una casa de ladrillos, las casas como tal no están hecha de una pieza, si observamos las paredes estas esta echas de pequeñas piezas llamadas ladrillos, entonces, el conjunto de estos ladrillos crea paredes, y un conjunto de paredes crean una casa. este ejemplo puede ser aplicado al patrón Composite, y no digo que vayamos a crear una casa con este patrón, sino más bien nos da una idea de cómo trabaja para poder utilizarlo con otros ejemplos.



Uso

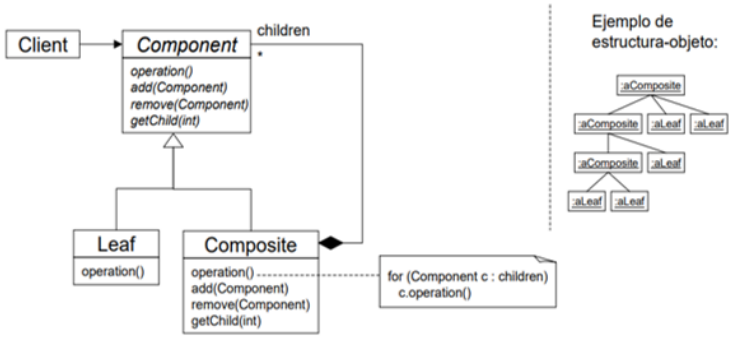
El patrón Composite sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera. Dependiendo de la implementación, pueden aplicarse procedimientos al total o una de las partes de la estructura compuesta (todo o parte) como si de un nodo final se tratara, aunque dicha parte esté compuesta a su vez de muchas otras.

Aplicación

Usar el patrón COMPOSITE cuando:

- Se quiere representar jerarquías de objetos todo-parte.
- Se quiere ser capaz de ignorar la diferencia entre objetos individuales y composiciones de objetos. Los clientes tratarán a todos los objetos de la estructura compuesta uniformemente.

Estructura



Participantes
El patrón Composite requiere mínimo de tres componentes para poder existir los cuales son Componente, Leaf o Rama y Composite.

a) **Component (Grafico)** Generalmente es una interface o clase abstracta la cual tiene las operaciones mínimas que serán utilizadas, este componente deberá ser extendido por los otros dos componentes Leaf y Composite. En nuestro ejemplo esto podría representar de forma abstracta un ladrillo o toda la casa (Mas adelante comprenderemos porque)

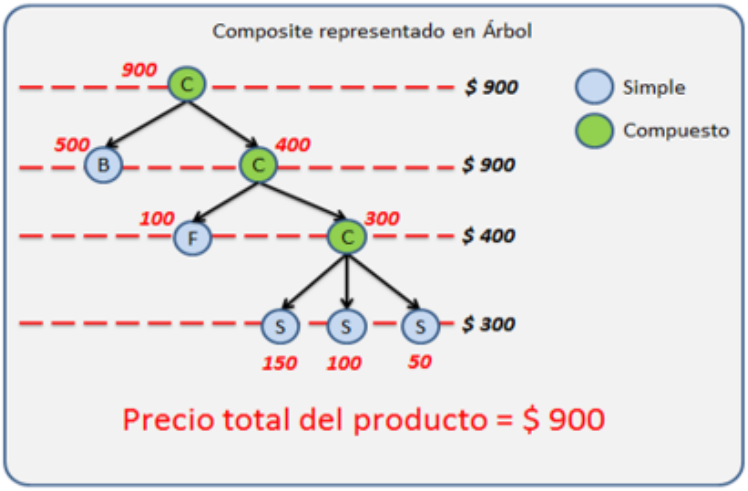
b) **Leaf o Rama (Línea, Rectángulo, Texto)** El leaf u hoja representa la parte más simple o pequeña de toda la estructura y este extiende o hereda de Component. En nuestro ejemplo, este representaría un ladrillo de nuestra casa.

c) **Composite (Dibujo)** Aquí es donde está la magia de este patrón, ya que el composite es una estructura conformada por otros Composite y Leaf, los Composite tiene los métodos add (añadir) y remove (remover) los cuales nos permiten agregar objetos de tipo Component, Sin embargo, el Componente es por lo general un Interface o Clase abstracta por lo que se puede agregar objetos de tipo Composite o Leaf. Desde el punto de vista del ejemplo de la casa el Composite podría representar un conjunto de ladrillos o la casa completa, Esto desde luego sería agregando varias Ladrillo(Leaf) al Composite para crear una Pared.

d) **Client** Es la entidad que hará uso del objeto compuesto.

EJEMPLO DE ESTUDIO DEL PATRON COMPOSITE Imaginemos un sistema de puntos de venta, en el cual se le pueden vender al cliente una serie de productos, estos productos pueden ser productos simples (Leaf) o paquetes (Composite). El sistema permitirá crear "Ordenes de Ventas", las cuales están compuestas por 1 o muchos productos.

En la imagen se muestra de forma gráfica cómo está compuesto un paquete. Los paquetes están creados a partir de un conjunto de productos simples y otros paquetes por lo que el precio de un paquete está calculado por el precio de sus hijos de forma recursiva. Muestra la estructura de una forma conceptual, sin embargo, la estructura es un poco más compleja, ya que está formado por una estructura de dato llamado "Arbol"



Esta imagen muestra un solo paquete, está formado de otros productos, simples y compuestos, un compuesto sería otro paquete, el cual tiene dentro más productos simples y como se vio en la figura anterior, el precio de un paquete es calculado por el precio de todos los hijos de forma recursiva.

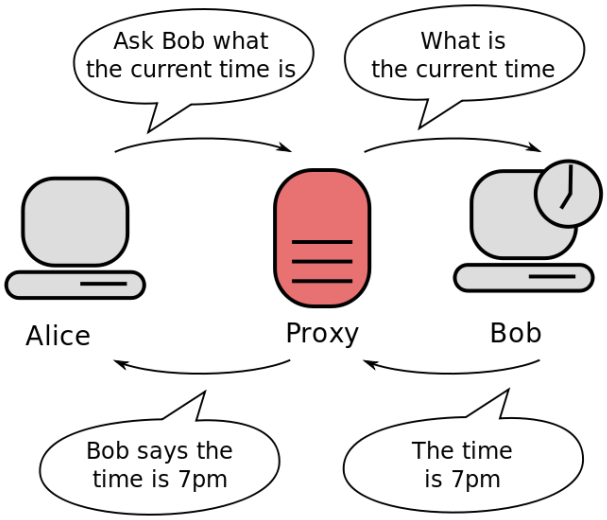
CONCLUSION

- Define jerarquías de clases hechas de objetos simples y compuestos.
- Si el código cliente espera un objeto simple, puede recibir también uno compuesto
- Puede hacer el diseño demasiado general. Es complicado restringir el tipo de componentes de un composite.
- Un paquete es producto compuesto de varios productos simples y otros paquetes.
- Simplifica el cliente. Los paquetes y productos simples deberán ser tratados de la misma forma, por lo que deberán tener un padre en común.
- El precio de un paquete es la suma de todos los productos simples que contenga.
- El sistema deberá mostrar el total de la Orden y los productos que contiene.
- Facilita la incorporación de nuevos tipos de componentes

2. Patrón de Diseño Proxy

1. ¿Qué es el Patrón Proxy?

El patrón Proxy proporciona un objeto intermediario entre el cliente y el objeto a utilizar, que permite configurar ciertas características (como el acceso) sin necesidad de modificar la clase original.



- Por ejemplo: Si tenemos muchos objetos imagen en un documento, se tardaría mucho tiempo en abrir el documento al cargar las imágenes de disco. Para evitarlo podemos sustituir los objetos imagen por objetos proxyImagen, con el mismo interfaz, pero que solamente cargan la imagen cuando se va a visualizar.

2. Tipos de Patrón Proxy

Existen varios tipos de Proxy que realizan distintos tipos de tareas: Proxy Remoto, Proxy Virtual, Proxy de Protección.

- **Proxy Remoto:** Se comporta como un representante local de un objeto, al realizar esto lo que hace es abstraer toda la "conversación" entre "dos" y de esta forma la comunicación entre el cliente y el objeto remoto es más fácil gastando menos recursos.
- **Proxy Virtual:** Lo que hace el proxy virtual es instanciar objetos cuyo costo computacional es muy elevado.
- **Proxy Protección:** Lo único que hace es establecer el control de acceso a un objeto dependiendo de los permisos o reglas de autorización.

3. Estructura del Patrón Proxy

3.1. Clasificación

- **Patrón estructural:** Ya que define la forma en cómo se organizan los objetos y las dependencias que tiene entre ellos.

3.2. Aplicaciones

- Útil cuando se desea retrasar la instanciación de un objeto hasta que sea necesario usarlo (optimiza operaciones costosas: invocar imagen).

– Proporciona un representante local de un objeto situado en otro espacio de direcciones (Proxy remoto o "Embajador").

– Uso en sistemas concurrentes, mediante cerrojo, controlando el acceso al objeto original.

– Puede utilizarse como un sustituto de un simple puntero, que lleva a cabo operaciones adicionales cuando se accede a un objeto (contar el número de referencias a un objeto real).

3.3. Consecuencias

– Un proxy puede ocultar el hecho de que un objeto reside en un espacio de direcciones diferente (proxy remoto).

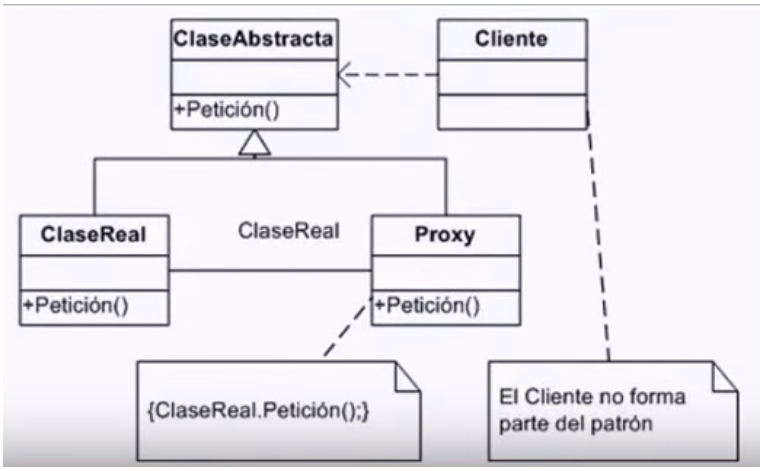
– Puede llevar a cabo optimizaciones tales como crear un objeto por encargo (invocar imagen).

– Permiten realizar tareas de mantenimiento adicionales cuando se accede a un objeto (Proxy de protección y de referencias inteligentes).

– Se introduce un nivel de indirección al acceder al objeto.

– Se consigue una administración transparente de los servicios del objeto real.

Representación UML



3.4. Participantes

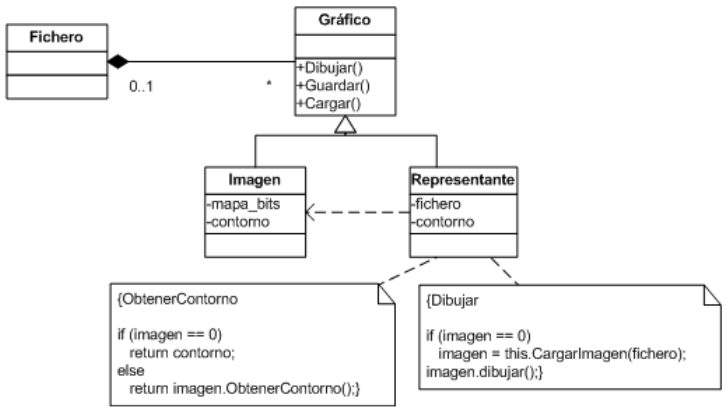
– **Sujeto:** Define la interfaz común para el RealSubject y el Proxy, de modo que pueda usarse un Proxy en cualquier sitio en el que se espere un RealSubject.

– **RealSubject:** Define el objeto real representado.

– **Proxy:**
Mantiene una referencia que permite al Proxy acceder al objeto real. Proporciona una interfaz idéntica a la del sujeto, de manera que un Proxy pueda ser sustituido por el sujeto real. Controla el acceso al sujeto real, y puede ser responsable de su creación y borrado.

4. Ejemplos UML

– **Diagrama UML**
Un ejemplo típico de aplicación del patrón proxy es el de un editor de documentos. El editor podrá incluir imágenes y dibujos complejos, y se plantea el problema de recuperar todos estos costosos objetos cada vez que se abre el documento. La aplicación del patrón proxy soluciona el problema definiendo un representante", que ocupe su lugar, hasta que sea necesario cargarlos.

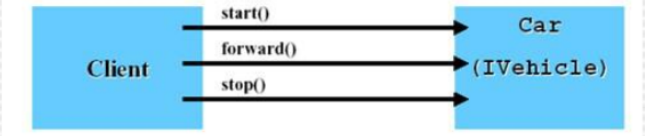


5. Ejemplos de Implementación sin Proxy y/o con Proxy

– **Sin Proxy**

```
public interface Ivehiculo {

	void start();
	void forward();
	void stop();
}

public class Coche implements Ivehiculo{

	private String nombre;

	public Coche(String nombre){
		this.nombre=nombre;
	}
	public void start() {
		System.out.println("coche"+this.nombre+"arranca");
	}
	public void stop() {
		System.out.println("coche"+this.nombre+"frena");
	}
	public void forward() {
		System.out.println("coche"+this.nombre+"acelera");
	}
}
```

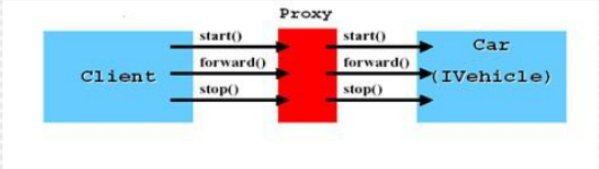
```
public class Cliente{
    public static void main(String[]args){
            IVehiculo v = new Coche("Sin Proxy");
            v.start();
            v.forward();
            v.stop();
    }
}
```



– **Con Proxy**

```
public class VehiculoProxy implements IVehiculo{
    private IVehiculo v;

    public VehiculoProxy(IVehiculo v){
            this.v = v;
    }

    public void start(){
            System.out.println("Vehiculo Proxy: parado");
            v.start();
            System.out.println("Vehiculo Proxy: en marcha");
    }
    public void forward(){
             System.out.println("Vehiculo Proxy: en aceleracion");
            v.forward();

    }
    public void stop(){
            System.out.println("Vehiculo Proxy: frenando");
            v.forward();
    }
```

```
public class ClienteConProxy{
    public static void main(String[]args){
            IVehiculo c = new Coche("Con Proxy");
            IVehiculo v=new VehiculoProxy(c);
            v.start();
            v.stop();
            v.forward();
    }
}
```



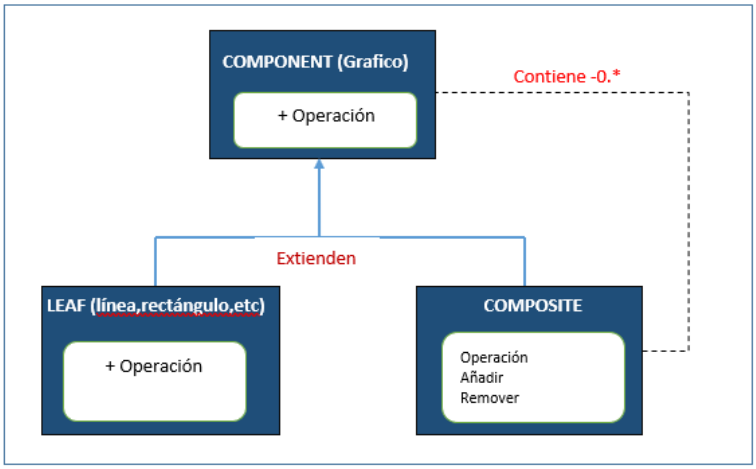
6. Conclusiones

– **No Proxy:** El cliente interactúa directamente con los métodos definidos en la interface

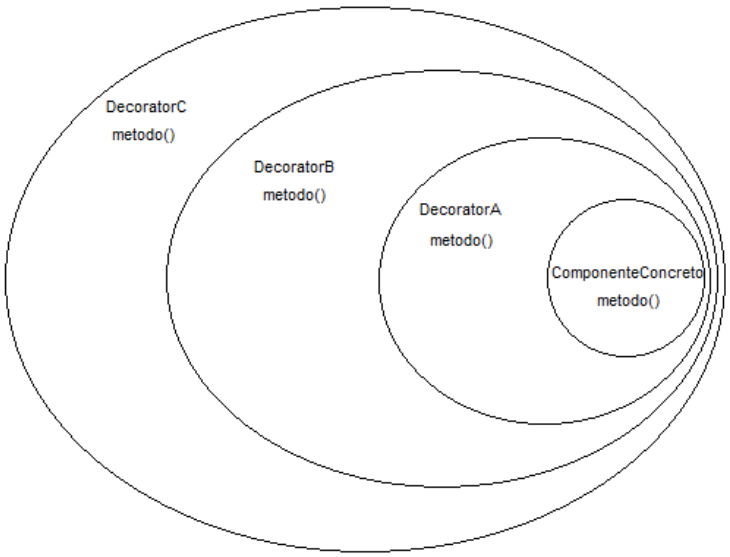
– **Proxy:** Se encuentra entre la interfaz y la implementación e intercepta las llamadas a los métodos. La intención del Proxy es controlar el acceso al objeto deseado, además de mejorar la funcionalidad del mismo

3. Patrones de Diseño Decorator

"Añadir responsabilidades a un objeto de forma dinámica. Los decoradores proporcionan una alternativa flexible a la herencia para extender funcionalidad."



El siguiente de los patrones estructurales que veremos sera el patrón Decorator o decorador. Su filosofía consiste en añadir responsabilidades de forma dinámica con el principal objetivo de evitar la conocida como "explosión de clases", es decir, la generación de un número elevado de subclases a partir de una superclase común. Como podemos observar en el gráfico superior, la clase Decorator hereda de la misma clase que el componente que se quiere decorar. Así, cada decorador es capaz de encapsular una instancia de cualquier otro objeto que herede del componente común, bien un componente concreto u otro decorador. Este comportamiento recuerda al que vimos previamente en el patrón Adapter, con la diferencia de que la clase Decorator, a diferencia de la clase Adapter, no transforma una interfaz, sino que añade cierta funcionalidad. La encapsulación puede ser iterativa, de modo que un componente concreto puede ser encapsulado por un decorador, que a su vez puede ser encapsulado por otro decorador... y así sucesivamente, añadiendo nueva funcionalidad en cada uno de los pasos. Resumiendo: el patrón Decoratorsustituye la herencia por un proceso iterativo de composición.



El objeto con el que el objeto cliente interactuará será aquel que se encuentre en la capa más externa (en este caso, DecoratorC), que se encargará de acceder a los objetos contenidos e invocar su funcionalidad, que será devuelta a las capas exteriores. Para comenzar, por tanto, debemos tener claros los siguientes conceptos sobre este patrón:

- Un decorador hereda de la misma clase que los objetos que tendrá que decorar.
- Es posible utilizar más de un decorador para encapsular un mismo objeto.
- El objeto decorador añade su propia funcionalidad, bien antes, bien después, de delegar el resto del trabajo en el objeto que está decorando.
- Los objetos pueden decorarse en cualquier momento, por lo que es posible decorar objetos de forma dinámica en tiempo de ejecución.

La razón por la que la clase Decorator hereda de la misma clase que el objeto que tendrá que decorar no es la de añadir funcionalidad, sino la de asegurarse de que ambos comparten el mismo tipo y puedan intercambiarse: un decorador podrá sustituir a un objeto decorado, basándonos en el principio SOLID del Principio de sustitución de Liskov.

Declarando las clases funcionales

Como viene siendo habitual, ilustraremos nuestro patrón haciendo uso de vehículos. En este caso, utilizaremos una case abstracta, llamada Vehiculo, del que heredarán las clases funcionales a las que llamaremos “Berlina” y “Monovolumen”, y los decoradores, que se limitarán a añadir funcionalidad a estas clases funcionales. Los decoradores que diseñaremos serán “Diesel”, “Gasolina”, “Inyeccion”, “CommonRail” y “Turbo”.

Estos decoradores se caracterizarán por:

- Disponer de una referencia a un vehículo que será inyectada en el constructor.
- Modificar el funcionamiento original de la clase que decoran, sobrecargando los métodos y llamando a los métodos de las clases encapsuladas para modificar su información o funcionamiento.

Comencemos codificando nuestra clase abstracta Vehiculo de la cual heredarán el resto de clases.

```
1
2   public abstract class Vehiculo
3   {
4       // Atributo común a todos los objetos que heredarán de esta clase
5       protected string descripcion = "Vehículo genérico";
6
7       // Método no abstracto que devolverá el contenido de la descripción
8       // Se declara como virtual para que pueda sustituirse en las clases
    derivadas
9       public virtual string Descripcion()
10      {
11          return descripcion;
12      }
13
14      // Métodos abstractos
15      public abstract int VelocidadMaxima();
16      public abstract double Consumo();
    }
```

Hecho esto, añadiremos nuestras clases funcionales: Monovolumen y Berlina.

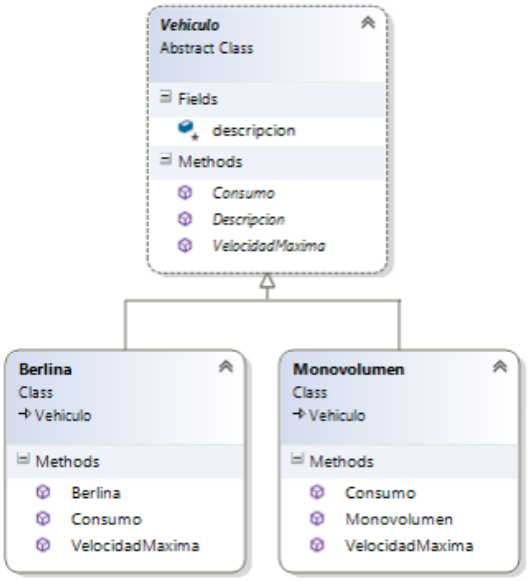
Monovolumen

```
1
2   public class Monovolumen : Vehiculo
3   {
4       public Monovolumen()
5       {
6           descripcion = "Monovolumen";
7       }
8
9       // Funcionalidad básica
10      public override int VelocidadMaxima()
11      {
12          return 160;
13      }
14
15      // Funcionalidad básica
16      public override double Consumo()
17      {
18          return 7.5;
19      }
  }
```

Berlina

```
1
2   public class Berlina : Vehiculo
3   {
4       public Berlina()
5       {
6           descripcion = "Berlina";
7       }
8
9       public override int VelocidadMaxima()
10      {
11          return 180;
12      }
13
14      public override double Consumo()
15      {
16          return 6.2;
17      }
  }
```

Como vemos, ambas clases heredan un atributo común, “descripcion”, y la funcionalidad VelocidadMaxima y Consumo. Asumiremos que un monovolumen posee una velocidad máxima de 160km/h y un consumo de 7,5 litros/100km, mientras que una berlina podrá alcanzar los 180km/h con un consumo de 6,2 litros/100km. Esta funcionalidad será modificada por nuestras clases decoradoras, que podrán aumentar o disminuir estas características. Con esto habríamos codificado nuestra rama “funcional”. Aún no hay rastro del patrón Decorator, ya que únicamente hemos hecho uso de la herencia de la manera habitual (de hecho, de una forma un tanto escueta).



Creando las clases Decorator

Es hora, por tanto, de añadir una nueva "rama" a nuestro árbol, añadiendo los decoradores. Comenzaremos por crear una nueva clase abstracta que herede de Vehiculo (para que pueda ocupar su lugar) y de la cual heredarán todos los decoradores.

```
1  public abstract class Decorator : Vehiculo
2  {
3      // Declaramos el método como abstracto para que todos los decoradores
   lo
4      // reimplementen.
5      public override abstract string Descripcion();
6  }
```

A continuación añadiremos los decoradores, que incluirán una referencia a un vehículo y que se construirán mediante la inyección de éste. Por tanto, las características de estos decoradores, además de heredar de Decorator, serán las siguientes:

- Contienen una referencia a un Vehiculo, que se insertará en el constructor.
- Modifican el funcionamiento de las clases que encapsulan, accediendo a sus atributos y métodos y adaptándolos a la nueva funcionalidad deseada.

Comenzaremos añadiendo un decorador llamado "Gasolina". La gasolina, al ser más explosiva, proporciona mayor velocidad punta, pero al ser menos energética que el gasoil, también conlleva tener un consumo más elevado. Esta clase tendrá, por tanto, el siguiente aspecto:

```
1
2   public class Gasolina : Decorator
3   {
4       // Instancia de la clase vehiculo
5       private Vehiculo vehiculo;
6
7       // Constructor que recibe el vehículo que encapsulará el decorator
8       public Gasolina(Vehiculo vehiculo)
9       {
10          this.vehiculo = vehiculo;
11      }
12
13      // Los métodos utilizan la información del objeto encapsulado y le
14      // incorporan su propia funcionalidad.
15      public override string Descripcion()
16      {
17          return vehiculo.Descripcion() + " Gasolina";
18      }
19
20      // Un vehículo gasolina proporciona más potencia, por lo que
21  "decora" el
22      // vehiculo añadiendo mayor velocidad máxima
23      public override int VelocidadMaxima()
24      {
25          return vehiculo.VelocidadMaxima() + 60;
26      }
27
28      // La gasolina es menos energética que el diesel, por lo que el
29  consumo
30      // de combustible es mayor. Decoraremos el vehículo añadiéndole un
31  consumo
32      // de 1.2 litros adicionales a los 100 km.
33      public override double Consumo()
34      {
35          return vehiculo.Consumo() + 1.2;
36      }
37  }
38
```

Podemos utilizar un razonamiento análogo para un motor diesel, que tendrá el funcionamiento opuesto.

```
1
2
3   public class Diesel : Decorator
4   {
5       // Instancia de la clase vehiculo
6       private Vehiculo vehiculo;
7
8       // Constructor que recibe el vehículo que encapsulará el decorator
9       public Diesel(Vehiculo vehiculo)
10      {
11          this.vehiculo = vehiculo;
12      }
13
14      // Los métodos utilizan la información del objeto encapsulado y le
15      // incorporan su propia funcionalidad.
16      public override string Descripcion()
17      {
18          return vehiculo.Descripcion() + " Diesel";
19      }
20
21      public override int VelocidadMaxima()
22      {
23          return vehiculo.VelocidadMaxima() + 20;
24      }
25
26      public override double Consumo()
27      {
28          return vehiculo.Consumo() - 0.8;
29      }
30  }
31
32
```

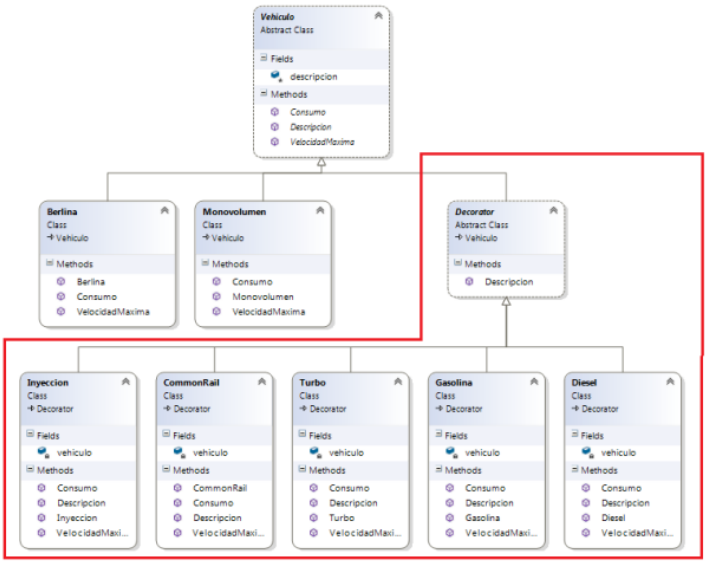
¡Ojo! Según nuestro diseño, un vehículo podría ser a la vez diesel y gasolina, ya que ambos heredan de la clase Vehiculo. Las reglas de negocio no deberían permitir que esto fuera así, por lo que deberíamos utilizar reglas adicionales para evitar que ambos decoradores estuviesen presentes en un mismo objeto. No obstante, ignoraremos este hecho en este ejemplo. A continuación añadiremos nuevos decoradores a nuestro vehículo. Por ejemplo, el turbo.

```
1
2
3   public class Turbo : Decorator
4   {
5       private Vehiculo vehiculo;
6
7       public Turbo(Vehiculo vehiculo)
8       {
9           this.vehiculo = vehiculo;
10      }
11
12      public override string Descripcion()
13      {
14          return vehiculo.Descripcion() + " Turbo";
15      }
16
17      public override int VelocidadMaxima()
18      {
19          return vehiculo.VelocidadMaxima() + 30;
20      }
21
22      public override double Consumo()
23      {
24          return vehiculo.Consumo() + 0.4;
25      }
26  }
```

Otra posible modificación al vehículo original podría ser la inyección de combustible, que no afectará a la velocidad pero mejorará notablemente el consumo de combustible:

```
1
2
3   public class CommonRail : Decorator
4   {
5       private Vehiculo vehiculo;
6
7       public CommonRail(Vehiculo vehiculo)
8       {
9           this.vehiculo = vehiculo;
10      }
11
12      public override string Descripcion()
13      {
14          return vehiculo.Descripcion() + " Common Rail";
15      }
16
17      public override int VelocidadMaxima()
18      {
19          return vehiculo.VelocidadMaxima() - 15;
20      }
21
22      public override double Consumo()
23      {
24          return vehiculo.Consumo() - 0.4;
25      }
26  }
```

Tras nuestra última adquisición, nuestra jerarquía de clases tendrá el siguiente aspecto:



Ahora es posible componer, en tiempo de ejecución, un objeto que combine algunas o todas estas características sin necesidad de codificar una clase por cada posible combinación.
Utilizando el patrón Decorator Es hora de hacer uso del patrón que acabamos de explicar. Comenzaremos creando un vehículo monovolumen y otro vehículo de tipo berlina, mostrando por pantalla sus características:

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           Vehiculo monovolumen = new Monovolumen();
6           Vehiculo berlina = new Berlina();
7
8           MostrarCaracteristicas(monovolumen);
9           MostrarCaracteristicas(berlina);
10      }
11
12      private static void MostrarCaracteristicas(Vehiculo v)
```

```
13      {
14          Console.WriteLine(string.Format("{0}\n\t- Velocidad punta de {1} km/h \n
    1/100km\n",
15                                          v.Descripcion(),
16                                          v.VelocidadMaxima(),
17                                          v.Consumo()));
18      }
19  }
```

Nuestro programa nos mostrará el siguiente resultado:

```
Monovolumen
        - Velocidad punta de 160 km/h
        Consumo medio de 7,5 l/100km

Berlina
        - Velocidad punta de 180 km/h
        Consumo medio de 6,2 l/100km
```

Como vemos, se nos ofrece una versión "básica" de nuestros objetos, que aún no han sido decorados. Probemos a decorar nuestro monovolumen añadiéndole un motor gasolina:

```
1
2   Vehiculo monovolumen = new Monovolumen();
3
4   // Decoramos el monovolumen añadiéndole un motor gasolina a través
5   // del decorador "Gasolina"
6   monovolumen = new Gasolina(monovolumen);
7
8   Vehiculo berlina = new Berlina();
9
10  MostrarCaracteristicas(monovolumen);
    MostrarCaracteristicas(berlina);
```

Esto modificará el comportamiento de nuestro monovolumen, que presentará las siguientes características:

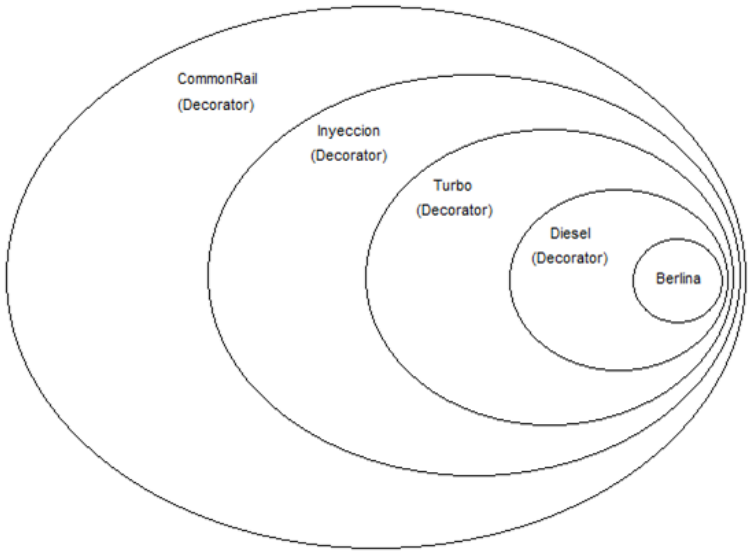
Se ha modificado, por tanto, su descripción, velocidad punta y consumo medio. Hagamos lo propio con el vehículo de tipo Berlina, al que convertiremos en un vehículo diesel turbo inyección common-rail:

```
1
2   Vehiculo berlina = new Berlina();
3
4   berlina = new Diesel(berlina);
5   MostrarCaracteristicas(berlina);
6
7   berlina = new Turbo(berlina);
8   MostrarCaracteristicas(berlina);
9
10  berlina = new Inyeccion(berlina);
11  MostrarCaracteristicas(berlina);
12
13  berlina = new CommonRail(berlina);
    MostrarCaracteristicas(berlina);
```

Como podemos observar, la propia instancia de Vehiculo es pasada como parámetro al constructor, que devuelve la misma instancia decorada con la nueva funcionalidad.

Así, con una única referencia, hemos conseguido modificar el comportamiento de la instancia en tiempo de ejecución sustituyendo la capacidad de especialización de la herencia por un proceso horizontal de composición.

¿Cuándo utilizar este patrón? Ejemplos reales? No debemos hacer caso omiso de que la referencia "berlina" ocupará ahora en memoria el quíntuple de lo que originalmente ocupaba, ya que se trata en realidad de una referencia a CommonRail que contiene un objeto Inyeccion que contiene un objeto Turbo que contiene un objeto Diesel que a su vez contiene la instancia Berlina original. Por tanto, conviene estudiar bien el contexto en el que se utilizará este patrón, ya que el ahorro que obtendremos en diseño lo pagaremos en memoria.



Un ejemplo claro de este patrón en la API de .NET es la familia de clases Stream. La clase Streames una clase abstracta que expone la funcionalidad básica que será implementada por las clases concretas y decoradas por las clases que componen el patrón.

Algunas de las clases concretas que podemos encontrar en esta familia son las siguientes:

FileStream: representa un flujo (stream) que se encargará de realizar operaciones E/S sobre un fichero físico.

MemoryStream: representa un flujo que realizará operaciones E/S en memoria. Se usa como una proyección temporal en memoria de otro flujo.

BufferedStream: representa una sección de un flujo que realizará operaciones E/S en memoria. La diferencia con el anterior es que el MemoryStream representa un flujo completo (por ejemplo, una proyección en memoria de un FileStream), mientras que BufferedStream se usa en conjunción con otros Streams para realizar operaciones de E/S que posteriormente serán leídas o volcadas desde/hacia el flujo original.

Estas clases serían nuestra Berlina y Monovolumen. Pero ¿y los Decorator? Las clases que actúan como decoradores son fácilmente identificables porque reciben un Stream como parámetro a la hora de crear una nueva instancia, a la vez que extienden la funcionalidad del objeto de la clase original.

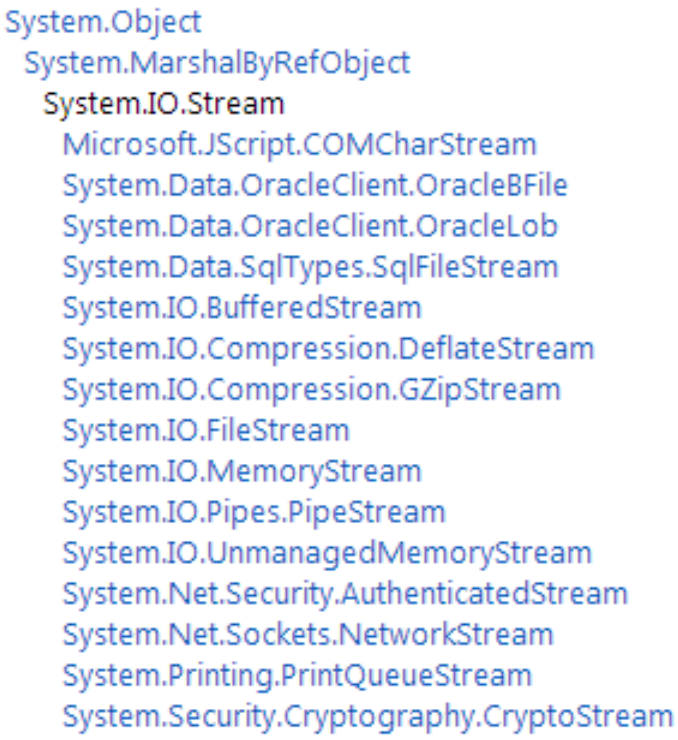
Algunos de los decoradores que podemos encontrar, y que son aplicables a cualquiera de las tres clases anteriores son:

CryptoStream: define una secuencia que vincula los flujos de datos a las transformaciones criptográficas.

AuthenticatedStream: proporciona métodos para pasar las credenciales a través de una secuencia y solicitar o realizar la autenticación para las aplicaciones de cliente-servidor.

GZipStream: proporciona los métodos y propiedades que permiten comprimir y descomprimir secuencias.

Cada uno de estos Decorator reciben en su constructor un Stream, añadiéndole nuevas funcionalidades y permitiendo que sigan actuando como Stream. Debido al polimorfismo, un CryptoStream que decore un MemoryStream seguirá pudiendo sustituir a cualquier objeto que se pase como parámetro como una referencia a Stream.



Podemos ver un ejemplo similar en Java, donde la clase InputStream actuaría como la clase base abstracta, clases como FileInputStream, StringBufferInputStream y ByteArrayInputStream actuarían como clases concretas, FilterInputStream actuaría como la clase abstracta de la que heredan todos los decoradores (clase que no existe en la familia Stream de .NET) y clases como BufferedInputStream, DataInputStream o LineNumberInputStream actuarían como decoradores, recibiendo un objeto de la clase InputStream como parámetro en su constructor.

4. Patrones de Diseño Facade

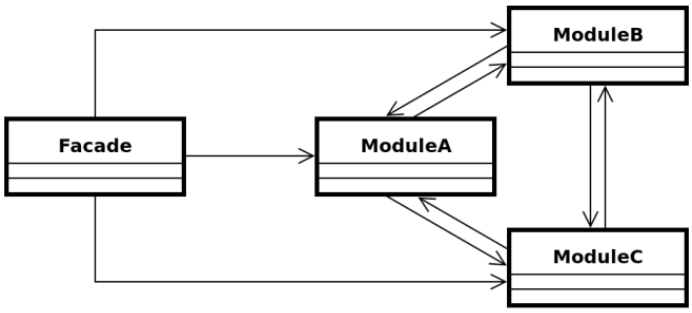
patrones de diseño se consideran una de las herramientas más valiosas para producir diseños de calidad y una técnica de propósito general para mejorar un diseño es identificar todas las realizaciones de patrones y aplicar reglas conocidas para mejorarlos.

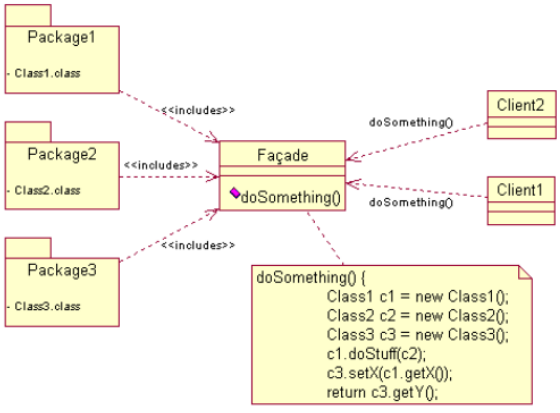
- Patrones de Diseño
  es un tipo de patrón de diseñoestructural. Viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre estos.

- Consideraciones para su aplicación
  Se aplicará el patrón fachada cuando se necesite proporcionar una interfaz simple para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel. Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable (esto es, reduciendo dependencias entre los subsistemas y los clientes).

- ¿Qué es una fachada o facade en inglés?
  Es un patrón de diseño que nos permite simplificar el interface de comunicación entre dos objetos.

- Estructura





- Participantes
  Fachada (Facade): conoce qué clases del subsistema son responsables de una determinada petición, y delega esas peticiones de los clientes a los objetos apropiados del subsistema.

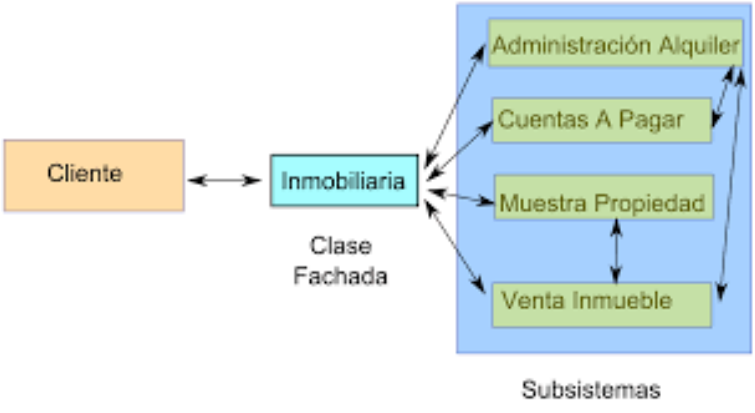
- Colaboraciones
  Los clientes que se comunican con el subsistema enviando peticiones al objeto Fachada, el cual las reenvía a los objetos apropiados del subsistema.
  Los objetos del subsistema realizan el trabajo final, y la fachada hace algo de trabajo para pasar de su interfaz a las del subsistema.
  Los clientes que usan la fachada no tienen que acceder directamente a los objetos del subsistema.

- Ventajas e inconvenientes
  La principal ventaja del patrón fachada consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.

  Busca simplificar el sistema, desde el punto de vista del cliente, proporcionando una interfaz unificada para un conjunto de subsistemas, definiendo una interfaz de nivel más alto. Esto hace que el sistema sea más fácil de usar.

  Este patrón busca reducir al mínimo la comunicación y dependencias entre subsistemas. Para ello, utilizaremos una fachada, simplificando la complejidad al cliente. El cliente debería acceder a un subsistema a través del Facade. De esta manera, se estructura un entorno de programación más sencillo, al menos desde el punto de vista del cliente (por ello se llama "fachada").

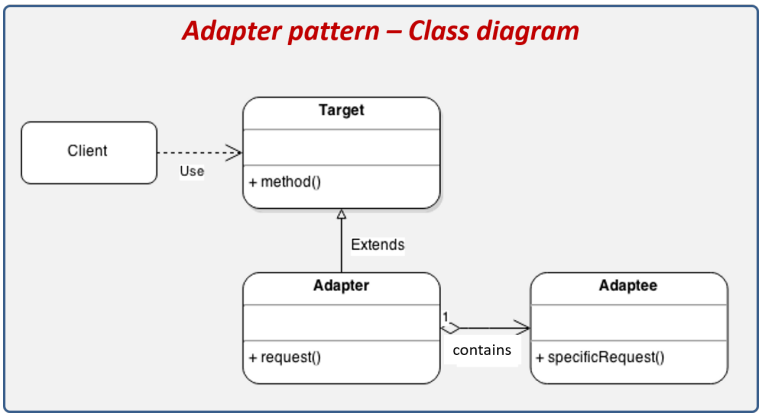
- ¿Se debe utilizar cuando?
  Se quiera proporcionar una interfaz sencilla para un subsistema complejo. Se quiera desacoplar un subsistema de sus clientes y de otros subsistemas, haciéndolo más independiente y portable. Se quiera dividir los sistemas en niveles: las fachadas serían el punto de entrada a cada nivel. Facade puede ser utilizado a nivel aplicación.



  Los clientes se comunican con el subsistema a través de la facade, que reenvía las peticiones a los objetos del subsistema apropiados y puede realizar también algún trabajo de traducción. Los clientes que usan la facade no necesitan acceder directamente a los objetos del sistema.

5. Patrones de Diseño Adapter

El patrón de diseño Adapter es utilizado cuando tenemos interfaces de software incompatibles, las cuales a pesar de su incompatibilidad tiene una funcionalidad similar. Este patrón es implementado cuando se desea homogeneizar la forma de trabajar con estas interfaces incompatibles, para lo cual se crea una clase intermedia que funciona como un adaptador. Esta clase adaptador proporcionará los métodos para interactuar con la interface incompatible.



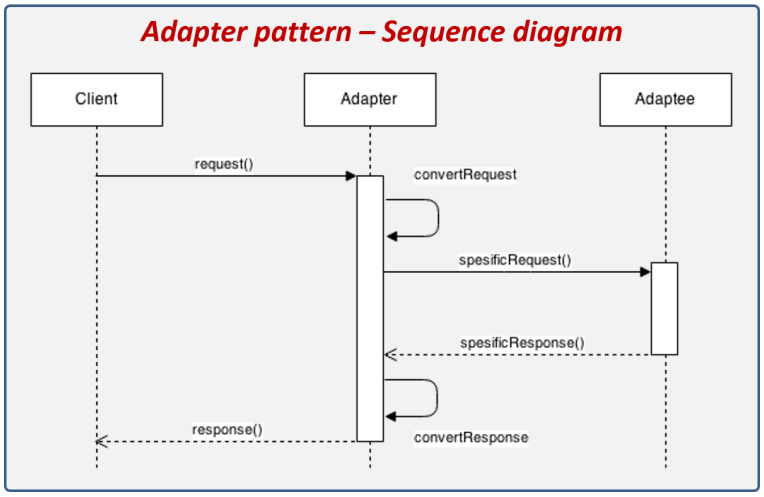
Los componentes que conforman el patrón son los siguientes:

a) Client: Actor que interactua con el Adapter.

b) Target: Interface que nos permitirá homogenizar la forma de trabajar con las interfaces incompatibles, esta interface es utilizada para crear los Adapter.

c) Adapter: Representa la implementación del Target, el cual tiene la responsabilidad de mediar entre el Client y el Adaptee. Oculta la forma de comunicarse con el Adaptee.

d) Adaptee: Representa la clase con interface incompatible.



a) El Client invoca al Adapter con parámetros genéricos.

b) El Adapter convierte los parámetros genéricos en parámetros específicos del Adaptee.

c) El Adapter invoca al Adaptee.

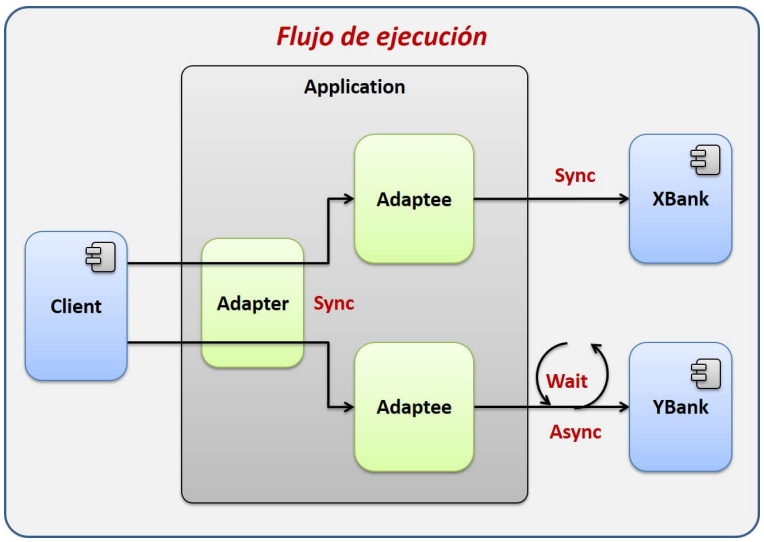
d) El Adaptee responde.

e) El Adapter convierte la respuesta del Adaptee a una respuesta genérica para el Client.

f) El Adapter responde al Client con una respuesta genérica.

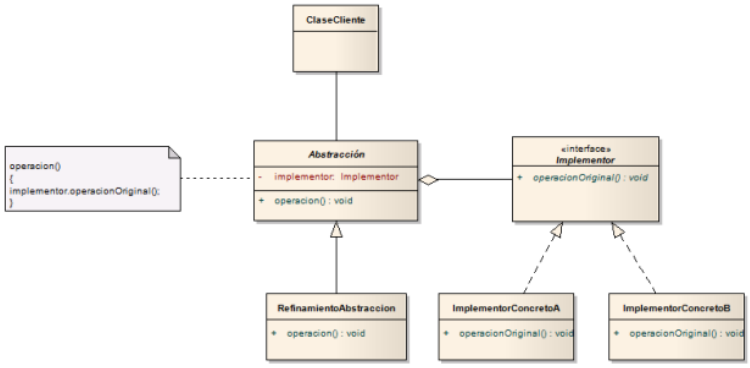
EJEMPLO DEL MUNDO REAL

Mediante la implementación del patrón de diseño Adapter crearemos un adaptador que nos permite interactuar de forma homogénea entre dos API bancarías, las cuales nos permite aprobar créditos personales, sin embargo, las dos API proporcionadas por los bancos cuenta con interfaces diferentes y aunque su funcionamiento es prácticamente igual, las interfaces expuestas son diferentes, lo que implica tener dos implementaciones diferentes para procesar los préstamos con cada banco. Mediante este patrón crearemos un adaptador que permitirá ocultar la complejidad de cada implementación del API, exponiendo una única interface compatible con las dos API proporcionadas, además que dejáramos el camino preparado por si el día de mañana llegara una nueva API bancaría.



6. Patrón de Diseño Bridge

Es normalmente uno de los patrones que más cuesta entender, especialmente si nos ceñimos únicamente a su descripción. La idea tras este patrón, sin embargo, es sencilla: dado que cualquier cambio que se realice sobre una abstracción afectará a todas las clases que la implementan, Bridge propone añadir un nuevo nivel de abstracción entre ambos elementos que permitan que puedan desarrollarse cada uno por su lado.

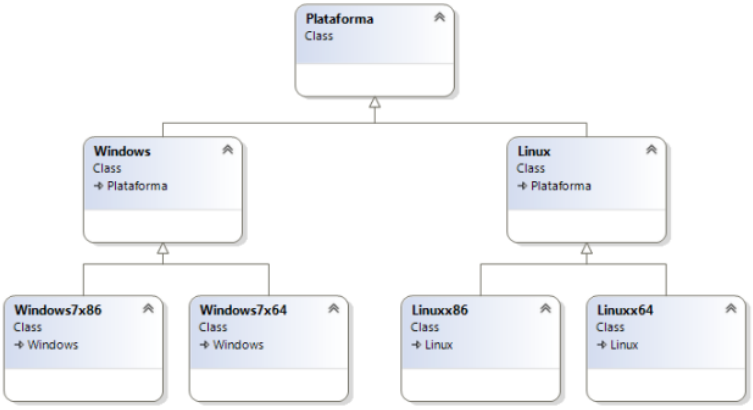


1. ¿Por qué "Bridge"?

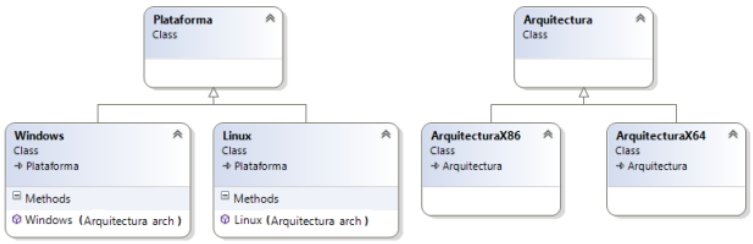
Patrón tras patrón, vemos que los mismos conceptos se repiten una y otra vez. Minimizar el acoplamiento, hacer que las clases dependan de abstracciones en lugar de depender de implementaciones, preferir el uso de composición antes que el uso de herencia... El patrón Bridge no es una excepción. Sin embargo, hasta el momento todos los patrones que hemos visto tenían una relación entre su nombre y su funcionalidad. Un Adapter adapta. Una factoría fabrica objetos. Un Builder construye. Pero... ¿Bridge? ¿Qué tiene que ver un puente con todo esto?

Una de las razones por las que opino que este patrón es complicado de entender a la primera es, precisamente, que no existe una relación clara entre su nombre y su descripción. ¿Llamar puente a desligar una interfaz de la implementación? ¿Por qué? La razón no está tanto en este proceso sino en el camino que existe entre la clase que refina la abstracción y las implementaciones de la interfaz.

Hablando en plata, y a modo de resumen, realizaremos la transformación del siguiente árbol de herencia:



En una composición como la siguiente:



2. Similitudes

La estructura de este patrón se parece mucho a la del patrón Adapter, ya que nuestra clase Abstracción hace las veces de "adaptador" entre nuestra clase cliente y la interfaz Implementor. Sin embargo, nos movemos por la sinuosa senda de la ingeniería, por lo que afirmar que Adapter y Bridge realizan lo mismo simplemente porque su estructura sea muy parecida es quedarnos en la superficie del problema que tratamos de resolver. La estructura de este patrón se parece mucho a la del patrón Adapter, ya que nuestra clase Abstracción hace las veces de "adaptador" entre nuestra clase cliente y la interfaz Implementor. Sin embargo, nos movemos por la sinuosa senda de la ingeniería, por lo que afirmar que Adapter y Bridge realizan lo mismo simplemente porque su estructura sea muy parecida es quedarnos en la superficie del problema que tratamos de resolver.

– **Un ejemplo de patrón Bridge**

Veamos la aplicación de nuestro patrón Bridge con un ejemplo en código C sharp. El ejemplo, como seguro que habréis adivinado, estará basado en vehículos. Nuestra abstracción simbolizará el vehículo en sí, mientras que la parte que se tenderá "al otro lado del puente" será el motor del mismo. Los tipos de vehículo podrán así evolucionar con independencia de los motores que éstos posean. Comenzaremos codificando la interfaz Implementor, que en nuestro ejemplo estará representada por el motor. O más específicamente, por la interfaz IMotor.

MOTOR

```
1  // Implementor
2  public interface IMotor
3  {
4      void InyectarCombustible(double cantidad);
5      void ConsumirCombustible();
6  }
```

Como vemos, nada complicado: nuestro Implementor expone dos métodos, InyectarCombustibley ConsumirCombustible, que deberán ser codificados en las clases que implementen la interfaz. Y dicho y hecho, añadiremos un par de clases cuyo papel en el patrón se corresponderá con ImplementorConcretoA e ImplementorConcretoB, y modelarán dos tipos de motores: diesel y gasolina.

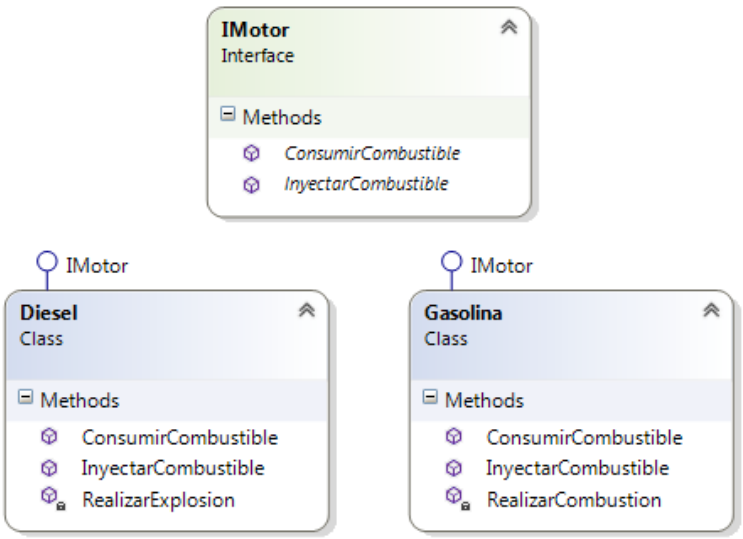
DIESEL

```
// ImplementorConcretoA
public class Diesel : IMotor
{
    #region IMotor Members

    public void InyectarCombustible(double cantidad)
    {
        Console.WriteLine("Inyectando " + cantidad + " ml. de Gasoil");
    }

    public void ConsumirCombustible()
    {
        RealizarExplosion();
    }

    #endregion

    private void RealizarExplosion()
    {
        Console.WriteLine("Realizada la explosión del Gasoil");
    }
}
```

GASOLINA

```
// ImplementorConcretoB
public class Gasolina : IMotor
{
    #region IMotor Members

    public void InyectarCombustible(double cantidad)
    {
        Console.WriteLine("Inyectando " + cantidad + " ml. de Gasolina");
    }

    public void ConsumirCombustible()
    {
        RealizarCombustion();
    }

    #endregion

    private void RealizarCombustion()
    {
        Console.WriteLine("Realizada la combustión de la Gasolina");
    }
}
```

Con estas tres clases ya habríamos desarrollado el subárbol izquierdo del diagrama UML que mostramos al comienzo del artículo: la interfaz Implementor junto a sus implementaciones:



La siguiente parte será encapsular la interfaz dentro de nuestra abstracción Vehiculo, que dispondrá de una referencia a IMotor y de un método que hará uso de los métodos de nuestra interfaz, encapsulando su funcionalidad tal y como hacíamos en el patrón Adapter:

VEHICULO

```
// Abstracción
public abstract class Vehiculo
{
    private IMotor motor;

    public Vehiculo(IMotor motor)
    {
        this.motor = motor;
    }

    // Encapsulamos la funcionalidad de la interfaz IMotor
    public void Acelerar(double combustible)
    {
        motor.InyectarCombustible(combustible);
        motor.ConsumirCombustible();
    }

    public void Frenar()
    {
        Console.WriteLine("El vehículo está frenando.");
    }

    // Método abstracto
    public abstract void MostrarCaracteristicas();
}
```

Como venimos observando a lo largo de los últimos patrones, el objeto que implementará el motor es inyectado en el constructor, siguiendo el quinto de los principios SOLID.
Por último, codificaremos la evolución de nuestra abstracción, que se corresponderá con RefinamientoAbstracciónA y RefinamientoAbstracciónB, representadas por dos tipos de vehículos: Berlina y Monovolumen.

VERLINA

```
// RefinamientoAbstraccionA
public class Berlina : Vehiculo
{
    // Atributo propio
    private int capacidadMaletero;

    // La implementacion de los vehículos se desarrolla de forma independiente
    public Berlina(IMotor motor, int capacidadMaletero) : base(motor)
    {
        this.capacidadMaletero = capacidadMaletero;
    }

    // Implementación del método abstracto
    public override void MostrarCaracteristicas()
    {
        Console.WriteLine("Vehiculo de tipo Berlina con un maletero con una capacidad
            capacidadMaletero + " litros.");
    }
}
```

3. ¿Cuándo utilizar este patrón?
Un ejemplo típico de un patrón Bridge lo puede conformar cualquier familia de drivers de un dispositivo, tal y como vimos en el primer ejemplo.
Otro ejemplo típico suele ser el de las APIs de dibujo. Los elementos genéricos, tales como formas y figuras serían las abstracciones (por ejemplo, Forma sería el elemento Abstraction del que derivarían abstracciones refinadas como Circulo o Cuadrado), mientras que la parte "dependiente" del sistema sería la API concreta que se encargaría de dibujar en pantalla las formas genéricas definidas en la abstracción. Este funcionamiento puede observarse en los paquetes de java java.awt y java.awt.peer. (en Button y List, por ejemplo).
Las situaciones óptimas en los que se debe utilizar este patrón serán, por tanto:

- Cuando se desea evitar un enlace permanente entre la abstracción y (toda o parte de) su implementación.
- Cuando los cambios en la implementación de una abstracción no debe afectar a las clases que hace uso de ella.
- Cuando se desea compartir una implementación entre múltiples objetos.

7. Webgrafía

- https://www.oscarblancarteblog.com/2014/10/07/patron-de-diseno-composite/
- http://arantxa.ii.uam.es/ eguerra/docencia/0708/05 %20Composite.pdf
- https://programacion.net/articulo/patrones_de_diseno_ix_patrones_estructurales_composite_1011
- http://design-patterns-with-uml.blogspot.com/2013/02/facade-pattern.html
- https://www.genbeta.com/desarrollo/diseno-con-patrones-y-fachadas
- http://blog.koalite.com/2016/12/los-patrones-de-diseno-hoy-patrones-estructurales/
- https://slideplayer.es/slide/3097642/
- https://slideplayer.es/slide/5568855/