

UNIVERSIDAD PRIVADA DE TACNA



INGENIERIA DE SISTEMAS

TEMA:

Patrones de Diseño Estructurales

CURSO:

BASE DE DATOS II

DOCENTE(ING):

Patrick Jose Cuadros Quiroga

Integrantes:

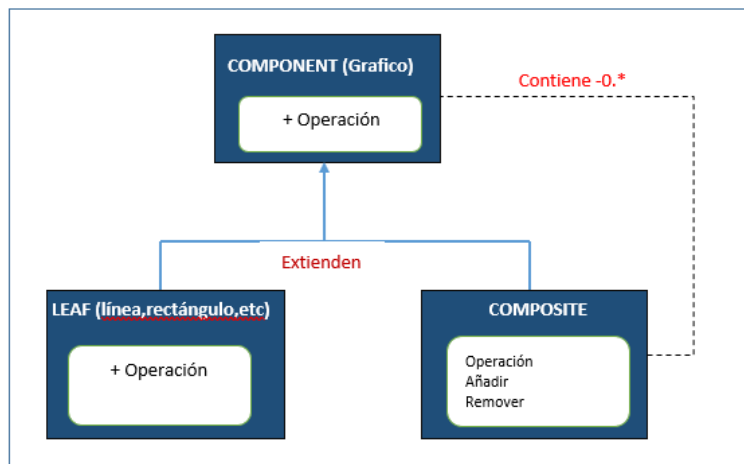
Marko Antonio RIVAS RIOS	(2016055461)
Jorge Luis MAMANI MAQUERA	(2016055236)
Andree Ludwed VELASCO SUCAPUCA	(2016055286)
Yofer Nain CATARI CABRERA	(2017059289)
Adnner Sleyder ESPERILLA RUIZ	(2015050543)
Jesus ESCALANTE ALANOCA	(2015050641)

Índice

1. Patrones de Diseño Composite	1
2. Patrón de Diseño Proxy	5
3. Patrón de Diseño ...	9
4. Patrones de Diseño Facade	10
5. Patrones de Diseño Adapter	12
6. Patrón de Diseño Bridge	14
7. Webgrafía	19

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

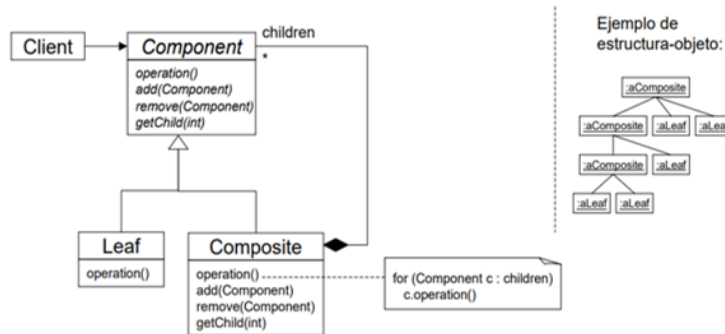
El patrón Composite sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol. Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos 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

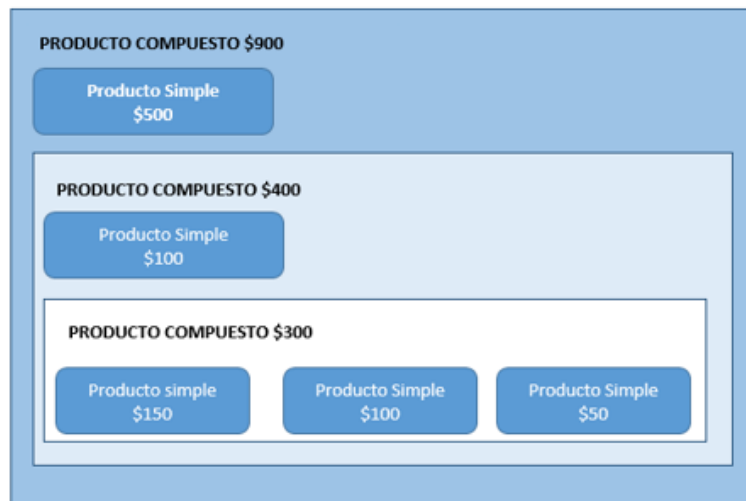


Participantes

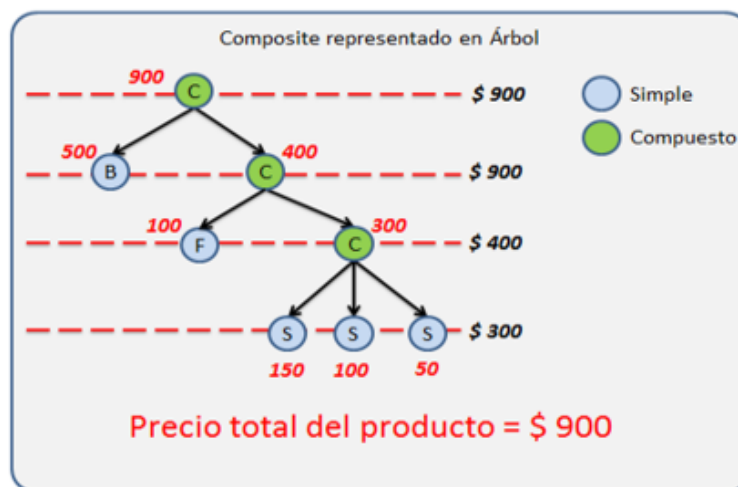
El patrón Composite requiere mínimo de tres componentes para poder existir los cuales son Componente, Leaf o Rama y Composite.

- 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)
- 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.
- 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.
- 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 “Árbol”



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.

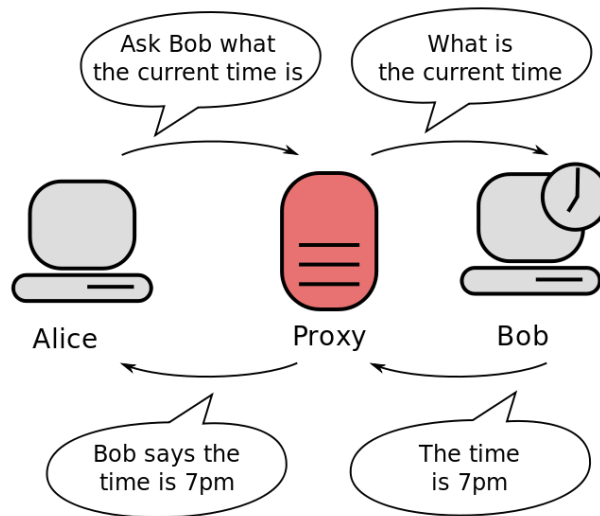
CONCLUSION

- 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

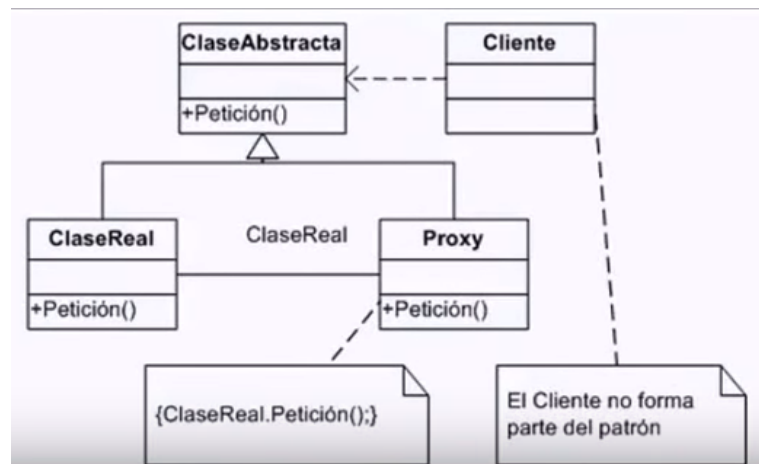
- Útil cuando se desea retrasar la instanciación de un objeto hasta que sea necesario usarlo (optimiza operaciones costosas: invocar imagen).

- 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.

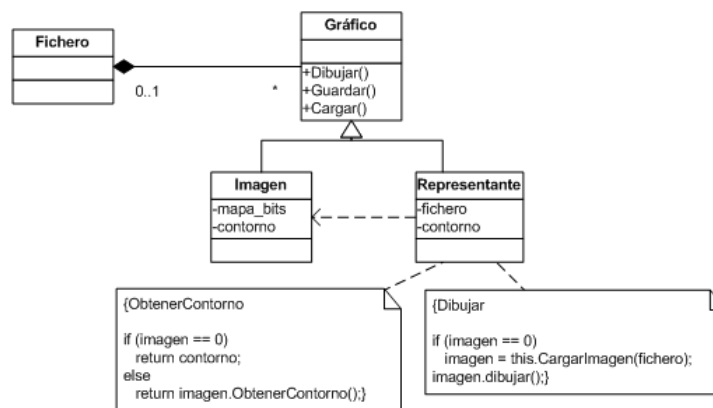
– **Proxy:**

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 {

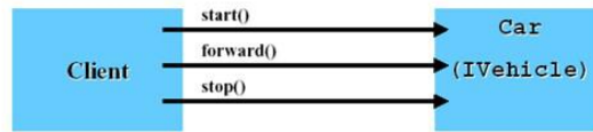
    void start();
    void forward();
    void stop();
}

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

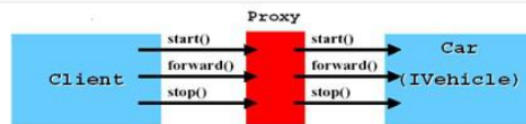
```
public class VehiculoProxy implements Ivehiculo{
    private Ivehiculo v;

    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

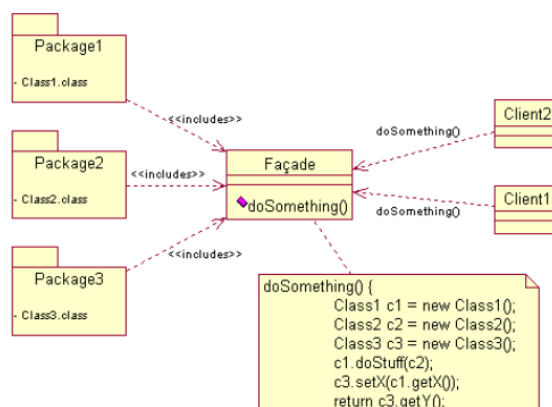
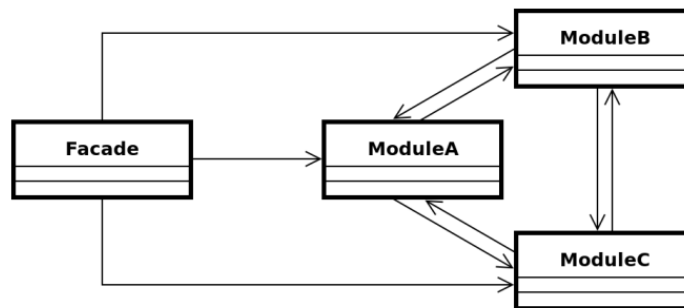
3. Patrón de Diseño ...

1. ¿Qué es el Patrón ...?

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 diseño estructural. 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.

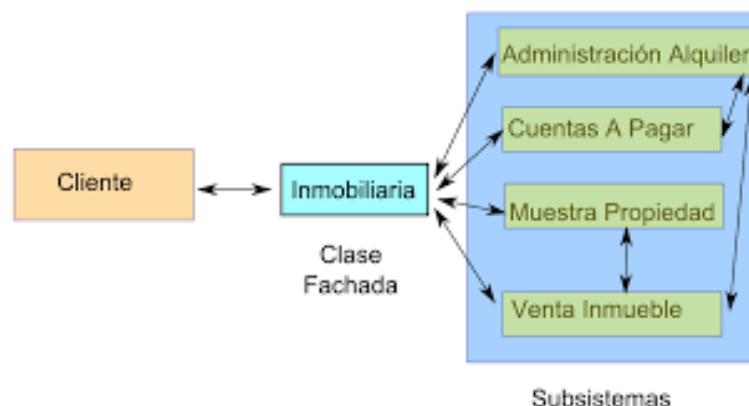
Busca simplificar el sistema, desde el punto de vista del cliente, proporcionando una interfaz unificada para un conjunto de subsistemas, definiendo una interfaz de nivel más alto. Esto hace que el sistema sea más fácil de usar.

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?

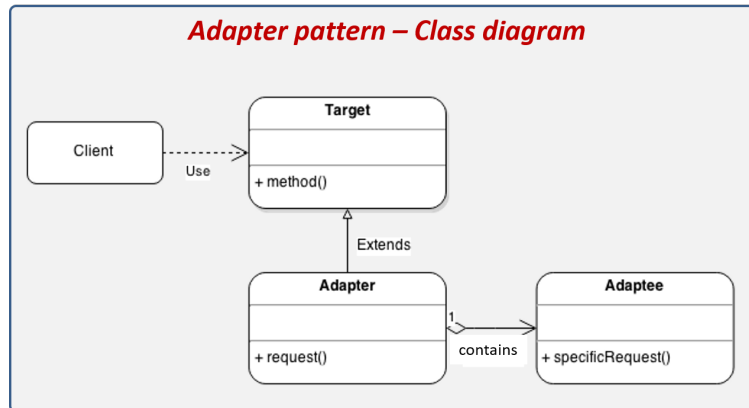
Se quiera proporcionar una interfaz sencilla para un subsistema complejo. Se quiera desacoplar un subsistema de sus clientes y de otros subsistemas, haciéndolo más independiente y portable. Se quiera dividir los sistemas en niveles: las fachadas serían el punto de entrada a cada nivel. Facade puede ser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

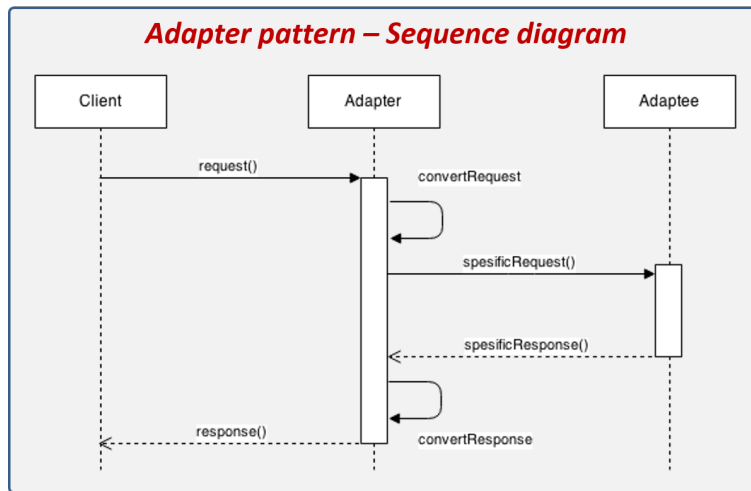
5. Patrones de Diseño Adapter

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:

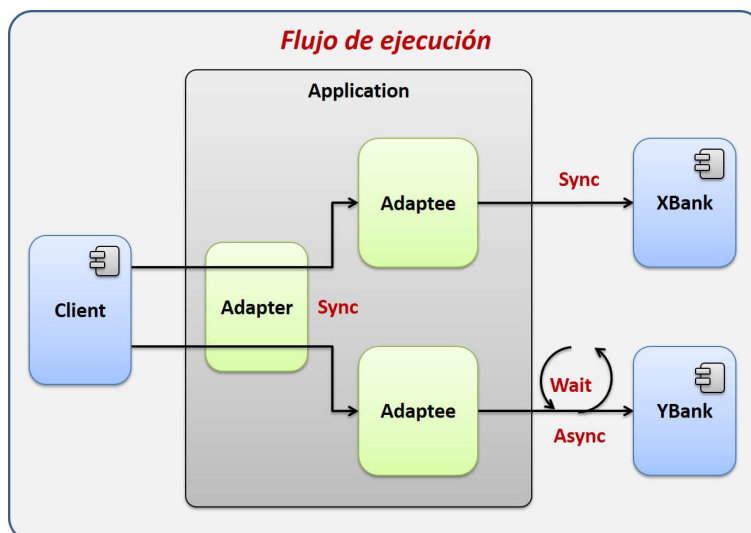
- a) Client: Actor que interactua con el Adapter.
- b) Target: Interface que nos permitirá homogeneizar la forma de trabajar con las interfaces incompatibles, esta interface es utilizada para crear los Adapter.
- c) Adapter: Representa la implementación del Target, el cual tiene la responsabilidad de 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.

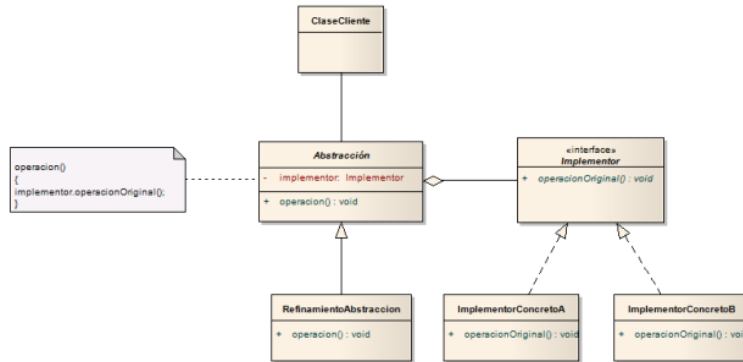
EJEMPLO DEL MUNDO REAL

Mediante la implementación del patrón de diseño Adapter crearemos un adaptador que nos permite interactuar de forma homogénea entre dos API bancarias, 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 bancaria.



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.

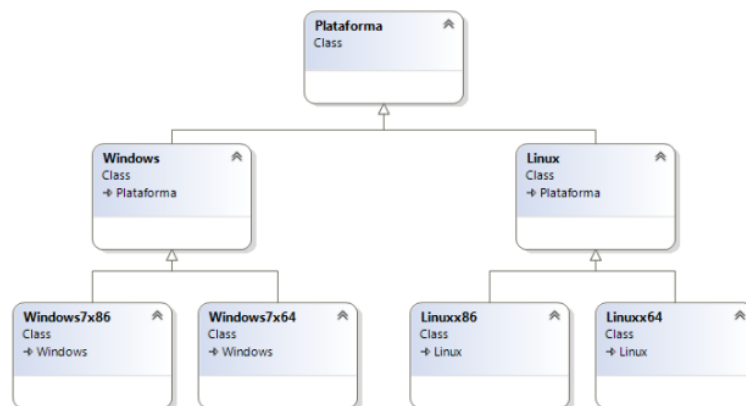


1. ¿Por qué “Bridge”?

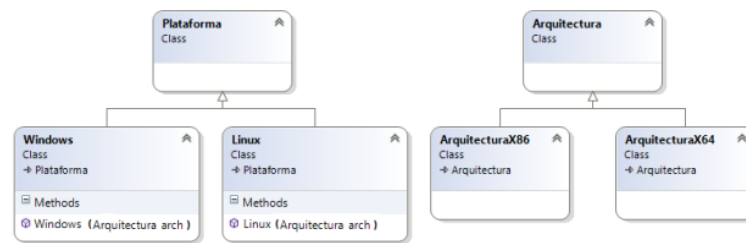
Patrón tras patrón, vemos que los mismos conceptos se repiten una y otra vez. Minimizar el acoplamiento, hacer que las clases dependan de abstracciones en lugar de depender de implementaciones, preferir el uso de composición antes que el uso de herencia. . . El patrón Bridge no es una excepción. Sin 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, InyectarCombustible y 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

```

1 // ImplementorConcretoA
2 public class Diesel : IMotor
3 {
4     #region IMotor Members
5
6     public void InyectarCombustible(double cantidad)
7     {
8         Console.WriteLine("Inyectando " + cantidad + " ml. de Gasoil");
9     }
10
11     public void ConsumirCombustible()
12     {
13         RealizarExplosion();
14     }
15
16     #endregion
17
18     private void RealizarExplosion()
19     {
20         Console.WriteLine("Realizada la explosión del Gasoil");
21     }
22 }

```

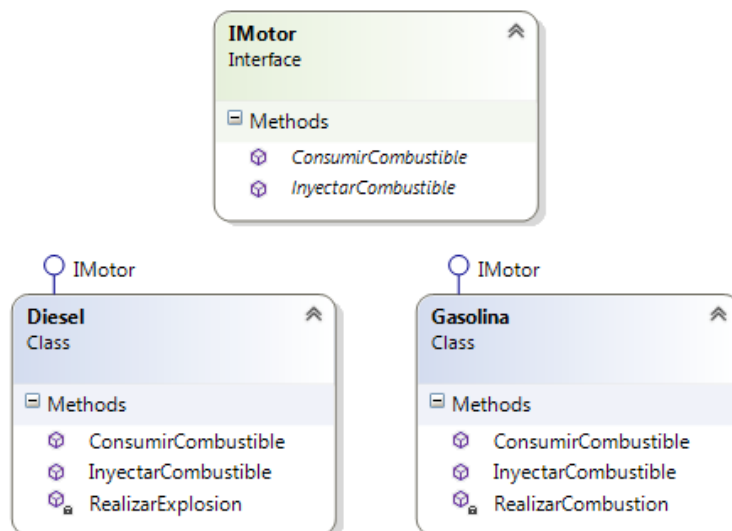
GASOLINA

```

1 // ImplementorConcretoB
2 public class Gasolina : IMotor
3 {
4     #region IMotor Members
5
6     public void InyectarCombustible(double cantidad)
7     {
8         Console.WriteLine("Inyectando " + cantidad + " ml. de Gasolina");
9     }
10
11     public void ConsumirCombustible()
12     {
13         RealizarCombustion();
14     }
15
16     #endregion
17
18     private void RealizarCombustion()
19     {
20         Console.WriteLine("Realizada la combustión de la Gasolina");
21     }
22 }

```

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

```
1 // Abstracción
2 public abstract class Vehiculo
3 {
4     private IMotor motor;
5
6     public Vehiculo(IMotor motor)
7     {
8         this.motor = motor;
9     }
10
11     // Encapsulamos la funcionalidad de la interfaz IMotor
12     public void Acelerar(double combustible)
13     {
14         motor.InjectarCombustible(combustible);
15         motor.ConsumirCombustible();
16     }
17
18     public void Frenar()
19     {
20         Console.WriteLine("El vehículo está frenando.");
21     }
22
23     // Método abstracto
24     public abstract void MostrarCaracteristicas();
25 }
```

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.

BERLINA

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

    // La implementación 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("Vehículo 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/disenio-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/>