Adapter

migranitodejava.blogspot.com/2011/06/adapter.html

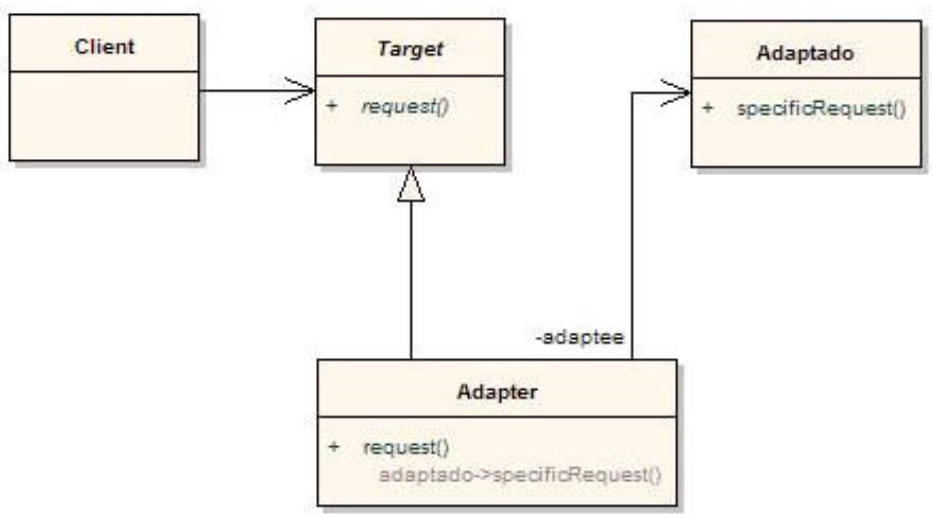
Busca una manera estandarizada de adaptar un objeto a otro. Se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.
Es conocido como Wrapper (al patrón Decorator también se lo llama Wrapper, con lo cual es nombre Wrapper muchas veces se presta a confusión).
Una clase Adapter implementa un interfaz que conoce a sus clientes y proporciona acceso a una instancia de una clase que no conoce a sus clientes, es decir convierte la interfaz de una clase en una interfaz que el cliente espera. Un objeto Adapter proporciona la funcionalidad prometida por un interfaz sin tener que conocer que clase es utilizada para implementar ese interfaz. Permite trabajar juntas a dos clases con interfaces incompatibles.

Este patrón se debe utilizar cuando:

- Se quiere utilizar una clase que llame a un método a través de una interface, pero se busca utilizarlo con una clase que no implementa ese interface.
- Se busca determinar dinámicamente que métodos de otros objetos llama un objeto.
- No se quiere que el objeto llamado tenga conocimientos de la otra clase de objetos.

Este patrón convierte la interfaz de una clase en otra interfaz que el cliente espera. Esto permite a las clases trabajar juntas, lo que de otra manera no podrían hacerlo debido a sus interfaces incompatibles.
Por lo general, esta situación se da porque no es posible modificar la clase original, ya sea porque no se tiene el código fuente de la clase o porque la clase es una clase de propósito general, y es inapropiado para ella implementar un interface par un propósito específico. En resumen, este patrón debe ser aplicado cuando debo transformar una estructura a otra, pero sin tocar la original, ya sea porque no puedo o no quiero cambiarla.

Diagrama UML



Target: define la interfaz específica del dominio que Cliente usa.
Cliente: colabora con la conformación de objetos para la interfaz Target.
Adaptado: define una interfaz existente que necesita adaptarse
Adapter: adapta la interfaz de Adaptee a la interfaz Target
El Cliente llama a las operaciones sobre una instancia Adapter. De hecho, el adaptador llama a las operaciones de Adaptee que llevan a cabo el pedido.

Ejemplo

Vamos a plantear el siguiente escenario: nuestro código tiene una clase Persona (la llamamos PersonaVieja) que se utiliza a lo largo de todo el código y hemos importado un API que también necesita trabajar con una clase Persona (la llamamos PersonaNueva), que si bien son bastante similares tienen ciertas diferencias:
Nosotros trabajamos con los atributos nombre, apellido y fecha de nacimiento.
Sin embargo, la PersonaNueva tiene un solo atributo nombre (que es el nombre y apellido de la persona en cuestión) y la edad actual, en vez de la fecha de nacimiento.

Para esta situación lo ideal es utilizar el Adapter. Para ello primero crearemos las 2 clases de Persona y sus correspondientes interfaces.

```
public interface IPersonaNueva {

    public String getNombre() ;
    public void setNombre(String nombre) ;
    public int getEdad() ;
    public void setEdad(int edad);

}
```

```
public interface IPersonaVieja {

    public String getNombre();
    public void setNombre(String nombre);
    public String getApellido();
    public void setApellido(String apellido);
    public Date getFechaDeNacimiento();
    public void setFechaDeNacimiento(Date fechaDeNacimiento);

}
```

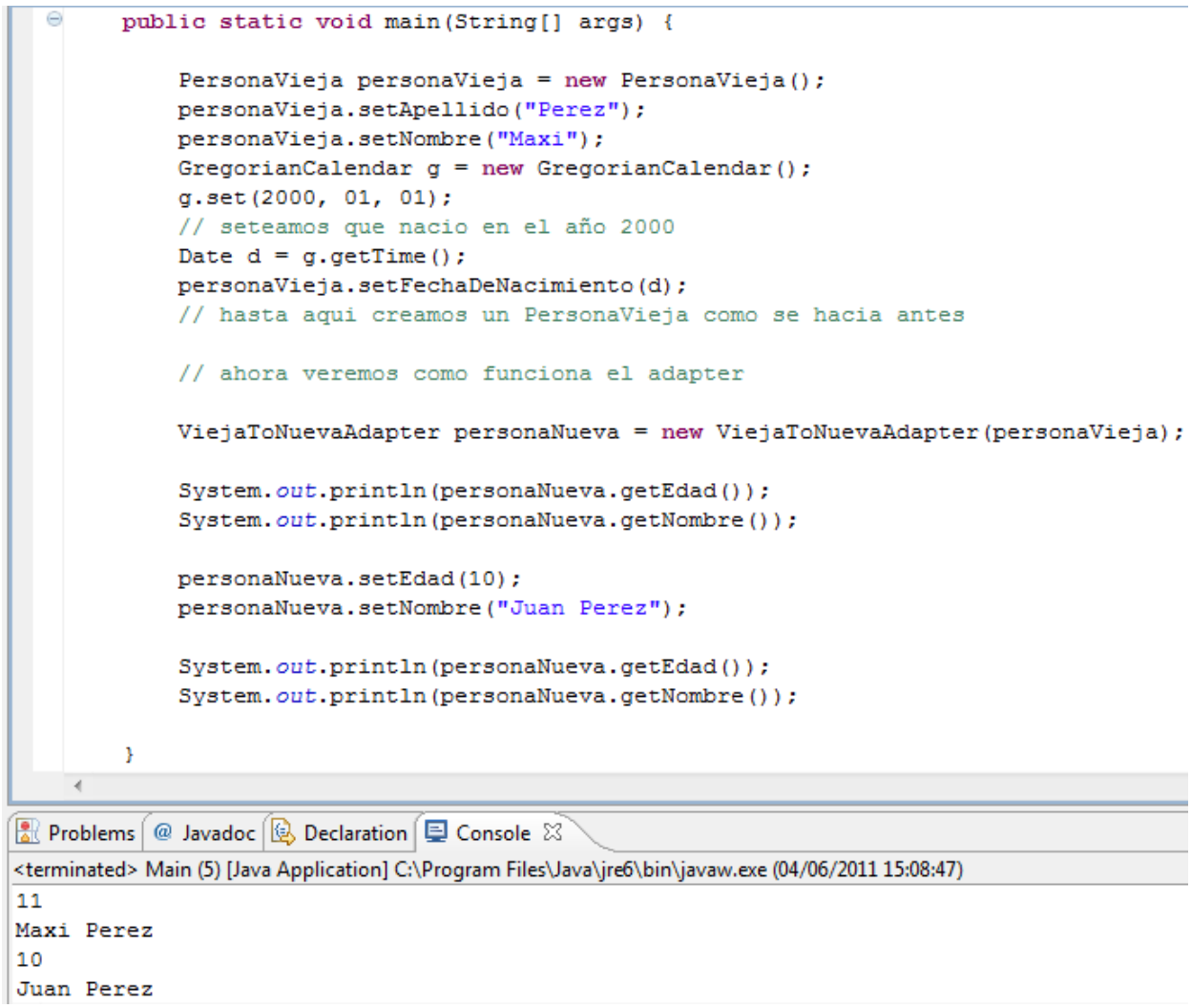
```
public class PersonaNueva implements IPersonaNueva {
    private String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
```

```
public class PersonaVieja implements IPersonaVieja {
    private String nombre;
    private String apellido;
    private Date fechaDeNacimiento;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
```

Ahora crearemos al Adapter.

```
public class ViejaToNuevaAdapter implements IPersonaNueva {
    private IPersonaVieja vieja;

    public ViejaToNuevaAdapter(IPersonaVieja vieja) {
        this.vieja = vieja;
    }

    public int getEdad() {
        GregorianCalendar c = new GregorianCalendar();
        GregorianCalendar c2 = new GregorianCalendar();
        c2.setTime(vieja.getFechaDeNacimiento());
        return c.get(1) - c2.get(1);
    }

    public String getNombre() {
        return vieja.getNombre() + " " + vieja.getApellido();
    }

    public void setEdad(int edad) {
        GregorianCalendar c = new GregorianCalendar();
        int anioActual = c.get(1);
        c.set(1, anioActual - edad);
        vieja.setFechaDeNacimiento(c.getTime());
    }

    public void setNombre(String nombreCompleto) {
        String[] name = nombreCompleto.split(" ");
        String firstName = name[0];
        String lastName = name[1];
        vieja.setNombre(firstName);
        vieja.setApellido(lastName);
    }

}
```

Y se utiliza de esta manera:



```
    public static void main(String[] args) {

        PersonaVieja personaVieja = new PersonaVieja();
        personaVieja.setApellido("Perez");
        personaVieja.setNombre("Maxi");
        GregorianCalendar g = new GregorianCalendar();
        g.set(2000, 01, 01);
        // seteamos que nacio en el año 2000
        Date d = g.getTime();
        personaVieja.setFechaDeNacimiento(d);
        // hasta aqui creamos un PersonaVieja como se hacia antes

        // ahora veremos como funciona el adapter

        ViejaToNuevaAdapter personaNueva = new ViejaToNuevaAdapter(personaVieja);

        System.out.println(personaNueva.getEdad());
        System.out.println(personaNueva.getNombre());

        personaNueva.setEdad(10);
        personaNueva.setNombre("Juan Perez");

        System.out.println(personaNueva.getEdad());
        System.out.println(personaNueva.getNombre());

    }
```

Problems | Javadoc | Declaration | Console

<terminated> Main (5) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (04/06/2011 15:08:47)

```
11
Maxi Perez
10
Juan Perez
```

Consecuencias

- El cliente y las clases Adaptee permanecen independientes unas de las otras.
- Puede hacer que un programa sea menos entendible.
- Permite que un único Adapter trabaje con muchos Adaptees, es decir, el Adapter por sí mismo y las subclases (si es que la tiene). El Adapter también puede agregar funcionalidad a todos los Adaptees de una sola vez.

Temas a tener en cuenta.
Si bien el Adapter tiene una implementación relativamente sencilla, se puede llevar a cabo con varias técnicas:
1) Creando una nueva clase que será el Adaptador, que extienda del componente existente e implemente la interfaz obligatoria. De este modo tenemos la funcionalidad que queríamos y cumplimos la condición de implementar la interfaz.
2) Pasar una referencia a los objetos cliente como parámetro a los costructores de los objetos adapter o a uno de sus métodos. Esto permite al objeto adapter ser utilizado con cualquier instancia o posiblemente muchas instancias de la clase Adaptee. En este caso particular, el Adapter tiene una implementación casi idéntica al patrón Decorator.
3) Hacer la clase Adapter una clase interna de la clase Adaptee. Esto asume que tenemos acceso al código de dicha clase y que es permitido la modificación de la misma.
4) Utilizar sólo interfaces para la comunicación entre los objetos.

Las opciones más utilizadas son la 1 y la 4.