Unidad 4: Patrones Estructurales

Introducción y orientaciones para el estudio

En el desarrollo de este módulo abordaremos:

- Concepto de patrones de estructurales.
- ¿Cómo reconocer su necesidad?
- Adapter, Bridge.
- Composite, Decorator.
- Proxy, Facade.

Objetivos

Pretendemos que al finalizar de estudiar esta unidad el alumno logre:

- Comprender qué es y cómo se aplica un patrón estructural.
- Definir el ámbito de aplicación.
- Poder utilizar los patrones definidos en la unidad.

Aclaraciones previas al estudio

En este módulo, el alumno encontrará:

- Contenidos
- Conceptualizaciones centrales

Usted debe tener presente que los contenidos presentados en el módulo no ahondan profundamente en el tema, sino que pretenden ser un recurso motivador para que, a través de la lectura del material, la bibliografía sugerida y el desarrollo de las actividades propuestas alcance los objetivos planteados.

Cada módulo constituye una guía cuya finalidad es facilitar su aprendizaje.

Concepto de patrones estructurales.

Son los patrones de diseño software que solucionan problemas de composición (agregación) de clases y objetos.

¿Cuándo usarlos?

Adapter (Adaptador). (Tiempo estimado: 2 hs)

Permite convertir o adaptar la interfaz (o contrato, o la suma de los métodos públicos) de un objeto / clase en otra esperada por el objeto cliente .

El problema / motivación

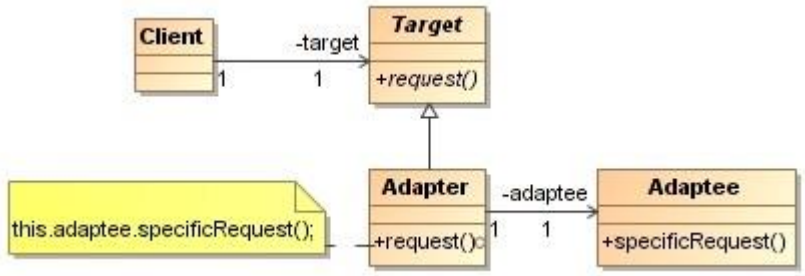
Cuando se tiene dos objetos / clases que quieren dialogar juntos pero sus interfaces no son compatibles por alguna razón.

Solución

Existirá una clase adaptadora y un clase adaptada. La adaptadora envuelve (por eso también se lo llama “wrapper”= envoltorio) o contiene a la clase adaptada, recibe las llamadas del cliente, las adapta y las retransmite a su objeto adaptado e interpreta, devolviendo también los resultados al cliente.

Variantes

- Objeto Adaptador (usa la asociación entre el cliente y el objeto adaptado)
- Clase Adaptadora (se usa solo en lenguajes con herencia múltiple)



Participantes

- Target define la interfaz específica del dominio que Client usa.
- Client colabora con la conformación de objetos para la interfaz Target.
- Adaptee define una interfaz existente que necesita adaptarse.
- Adapter adapta la interfaz de Adaptee a la interfaz Target.

Colaboraciones

Client llama a las operaciones sobre una instancia Adapter. De hecho, el adaptador llama a las operaciones de Adaptee que llevan a cabo el pedido.

Consecuencias

Los adaptadores de clase y objetos tienen varios pros y contras.

- Un adaptador de clase:
    - adapta Adaptee a Target encargando a una clase Adaptee concreta. Como consecuencia, una clase adaptadora no funcionará cuando se desea adaptar una clase y todas sus subclases.

    - permite a los Adapter sobreescribir algo de comportamiento de Adaptee, ya que Adapter es una subclase de Adaptee.
- Un adaptador de objeto:
    - 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.
    - hace difícil sobrescribir el comportamiento de Adaptee. Esto requerirá derivar Adaptee y hacer que Adapter se refiera a la subclase en lugar que al Adaptee por sí mismo.

Aquí hay otras cuestiones a considerar cuando se utiliza el patrón Adapter:

- **¿Cuánta adaptación hace el Adapter?** Adapter varía en la cantidad de trabajo que hace para adaptar Adaptee a la interfaz Target. Hay un espectro de trabajo posible, desde una simple conversión (por ejemplo, cambiando los nombres de las operaciones) hasta soportando un conjunto de operaciones enteramente diferentes. La cantidad de trabajo que Adapter hace depende de cuanto de similar tienen la interfaz Target con Adaptee.

- **Adaptadores enganchables**. Una clase es más reutilizable cuando se deja a un lado la suposición de que otras clases deben utilizarla. Convirtiendo la adaptación de una interfaz en una clase, se elimina la suposición de que otras clases ven la misma interfaz. Dicho de otra manera, la adaptación de la interfaz permite incorporar a la clase en sistemas existentes que pueden esperar diferentes interfaces de la misma.

Implementación

```
package Structural_patterns;

public class AdapterWrapperPattern {

	public static void main(String args[]){
		Guitar eGuitar = new ElectricGuitar();
		eGuitar.onGuitar();
		eGuitar.offGuitar();
		Guitar eAGuitar = new ElectricAcousticGuitar();
		eAGuitar.onGuitar();
		eAGuitar.offGuitar();
	}


	public abstract class Guitar{
		abstract public void onGuitar();
		abstract public void offGuitar();
```

```
	}

	public class ElectricGuitar extends Guitar{

		public void onGuitar() {
			System.out.println("Playing Guitar");
		}

		public void offGuitar() {
			System.out.println("I'm tired to play the guitar");
		}
	}

	/**
	 * Class to Adapter/Wrapper
	 */
	public class AcousticGuitar{

		public void play(){
			System.out.println("Playing Guitar");
		}
		public void leaveGuitar(){
			System.out.println("I'm tired to play the guitar");
		}
	}

	/**
	 * we Adapter/Wrapper AcousticGuitar into
	   * ElectricAcousticGuitar to adapt into the GuitarModel
	 */
	public class ElectricAcousticGuitar extends Guitar{
		AcousticGuitar acoustic = new AcousticGuitar();

		public void onGuitar() {
			acoustic.play();
		}

		public void offGuitar() {
			acoustic.leaveGuitar();
		}
	}
}
```

Facade (Fachada) (Tiempo estimado: 2 hs)

Provee una única interfase a un set de interfaces en un subsistema. Define una interfase se alto nivel que permite que un subsistema sea fácil de ser usado.

El problema / motivación

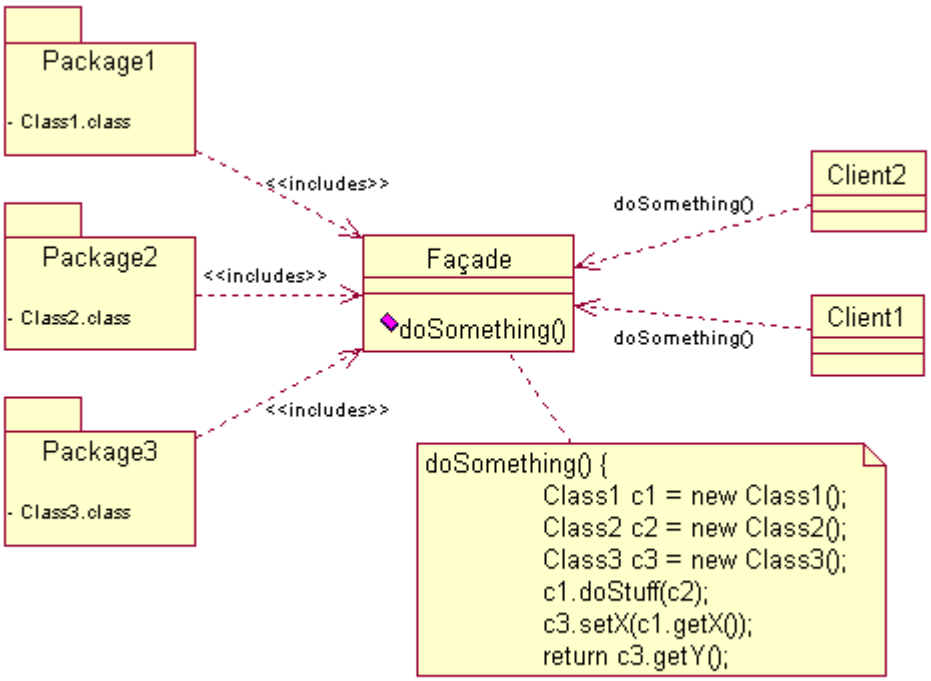
Es sabido que estructurar un sistema en subsistemas reduce la complejidad.
El logro de un diseño global exitoso es minimizar la comunicación y dependencias entre los distintos subsistemas (siguiendo las normas del bajo acoplamiento). El desafío es implementarlo de una manera elegante.

Solución

- La solución es anteponer como único punto de acceso a la funcionalidad de un módulo o subsistema una clase que permita canalizar los requerimientos desacoplando las dependencias de las clases clientes el exterior hacia dentro de los componentes del subsistema.
- La clase fachada estará provista de los elementos y relaciones necesarias para acceder y dialogar con las clases internas del subsistema de manera de poder proveer y cumplir con los requerimientos.

En resumen:

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).



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.

- **Subclases** (ModuleA, ModuleB, ModuleC...): implementan la funcionalidad del subsistema. Realizan el trabajo solicitado por la fachada. No conocen la existencia de la fachada.

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.

Consecuencias

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.

Como **inconveniente**, si se considera el caso de que varios clientes necesiten acceder a subconjuntos diferentes de la funcionalidad que provee el sistema, podrían acabar usando sólo una pequeña parte de la fachada, por lo que sería conveniente utilizar varias fachadas más específicas en lugar de una única global.

Usos Conocidos (Problemas / Soluciones)
Problema: Un cliente necesita acceder a parte de la funcionalidad de un sistema más complejo.
Solución: Definir una interfaz que permita acceder solamente a esa funcionalidad.

Problema: Existen grupos de tareas muy frecuentes para las que se puede crear código más sencillo y legible.
Solución: Definir funcionalidad que agrupe estas tareas en funciones o métodos sencillos y claros.

Problema: Una biblioteca es difícilmente legible.
Solución: Crear un intermediario más legible.

Problema: Dependencia entre el código del cliente y la parte interna de una biblioteca.
Solución: Crear un intermediario y realizar llamadas a la biblioteca sólo o, sobre todo, a través de él.

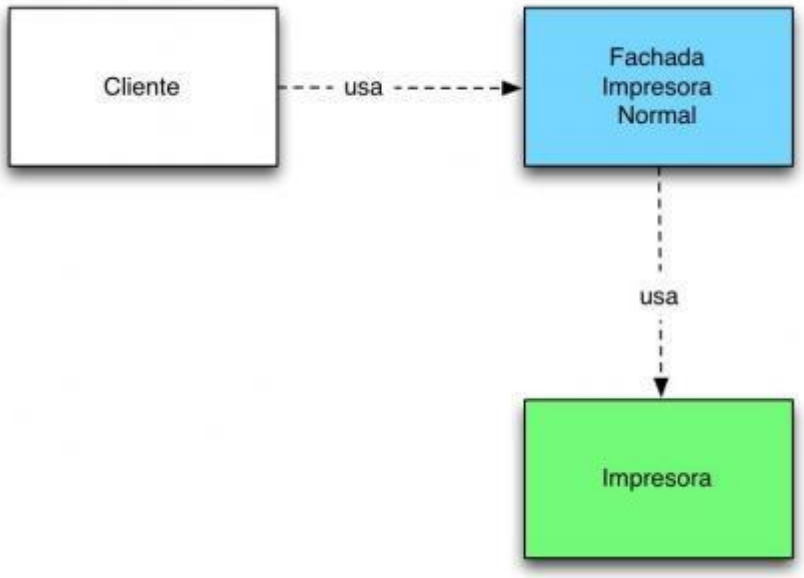
Problema: Necesidad de acceder a un conjunto de APIs que pueden además tener un diseño no muy bueno.
Solución: Crear una API intermedia, bien diseñada, que permita acceder a la funcionalidad de las demás.

Problema: Muchas clases cliente quieren usar varias clases servidoras, y deben saber cuál es exactamente la que le proporciona cada servicio. El sistema se volvería muy complejo, porque habría que relacionar todas las clases cliente con todas y cada una de las clases servidoras.
Solución: Crear una o varias clases Facade, que implementen todos los servicios, de modo que o todos los clientes utilicen esa única clase, o que cada grupo de clientes use la fachada que mejor se ajuste a sus necesidades.

Implementación

Diseño inicial

Diseño aplicando el patrón Facade




```
public class Impresora {
            private String tipoDocumento;
            private String hoja;
            private boolean color;
            private String texto;

            public String getTipoDocumento() {
                        return tipoDocumento;
             }

             public void setTipoDocumento(String tipoDocumento) {
                        this.tipoDocumento = tipoDocumento;
            }

            public void setHoja(String hoja) {
                        this.hoja = hoja;
            }

            public String getHoja() {
                        return hoja;
```

```
        }

        public void setColor(boolean color) {
                this.color = color;
        }

        public boolean getColor() {
                return color;
        }

        public void setTexto(String texto) {
                this.texto = texto;
        }

        public String getTexto() {
                return texto;
        }

        public void imprimir() {
                impresora.imprimirDocumento();
        }
 }
```

Se trata de una clase sencilla que imprime documentos en uno u otro formato. El código de la clase cliente nos ayudará a entender mejor su funcionamiento.

```
public class PrincipalCliente {
        public static void main(String[] args) {
                Impresora i = new Impresora();
                i.setHoja("a4");
                i.setColor(true);
                i.setTipoDocumento("pdf");
                i.setTexto("texto 1");
                i.imprimirDocumento();
                Impresora i2 = new Impresora();
                i2.setHoja("a4");
                i2.setColor(true);
                i2.setTipoDocumento("pdf");
                i2.setTexto("texto 2");
                i2.imprimirDocumento();
                Impresora i3 = new Impresora();
                i3.setHoja("a3");
                i3.setColor(false);

                i3.setTipoDocumento("excel");
                i3.setTexto("texto 3");
                i3.imprimirDocumento();
        }
```

```
}
```

Como podemos ver la clase cliente se encarga de invocar a la impresora, y configurarla para después imprimir varios documentos .Ahora bien prácticamente todos los documentos que escribimos tienen la misma estructura (formato A4, Color , PDF). Estamos continuamente repitiendo código. Vamos a construir una nueva clase FachadaImpresoraNormal que simplifique la impresión de documentos que sean los más habituales

```
public class FachadaImpresoraNormal {
            Impresora impresora;
            public FachadaImpresoraNormal(String texto) {
                        super();
                        impresora= new Impresora();
                        impresora.setColor(true);
                        impresora.setHoja("A4");
                        impresora.setTipoDocumento("PDF");
                        impresora.setTexto(texto);
             }

            public void imprimir() {
                        impresora.imprimirDocumento();
             }
 }
```

De esta forma el cliente quedará mucho más sencillo :

```
public class PrincipalCliente2 {
            public static void main(String[] args) {
                        FachadaImpresoraNormal        fachada1=        new
FachadaImpresoraNormal("texto1");
                        fachada1.imprimir();
                        FachadaImpresoraNormal        fachada2=        new
FachadaImpresoraNormal("texto2");
                        fachada2.imprimir();
                        Impresora i3 = new Impresora();
                        i3.setHoja("a4");
                        i3.setColor(true);
                        i3.setTipoDocumento("excel");
                        i3.setTexto("texto 3");
                        i3.imprimirDocumento();
             }

 }
```

Composite (Objeto Compuesto) (Tiempo estimado: 3 hs)

El diseño del patrón busca organizar objetos dentro de un árbol de estructuras jerárquicas de tipo todo-parte.
De esta forma le permite a aquellos objetos cliente tratar a cualquier objeto de esta jerarquía en forma individual (sean simples o compuestos) de la misma forma o de forma uniforme.

El problema / motivación

Se presentan escenarios en los cuales tenemos objetos y contenedores de estos objetos en los cuales se pueden requerir la invocación de los mismos métodos o la ejecución de las mismas responsabilidades.

Si bien, no parece haber dificultad, el problema que se presenta, es justamente el carácter de contención de unos para los otros que obligan a una implementación diferente teniendo que distinguir unos de otros.

Solución

La solución está en el diseño de una estructura que permite ser usada en forma simple o compuesta en el caso de la contención de unos con otros (es esto le llamamos la composición) de la misma manera, usando un mecanismo recursivo para que los clientes no tengan que saber de qué están compuestos.

Implementación

Situación a resolver:

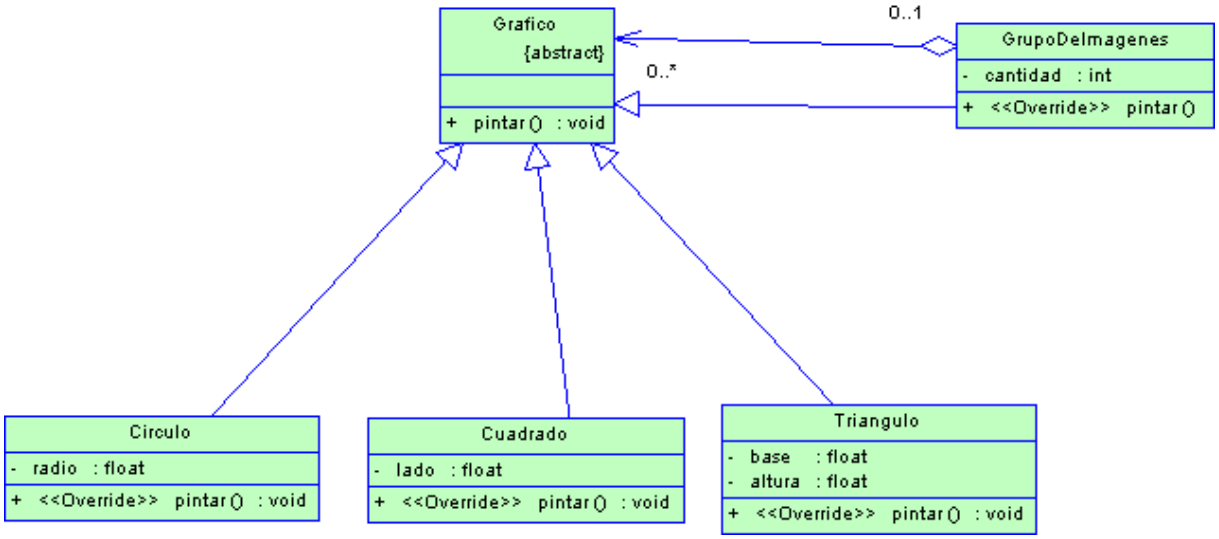
Imaginemos que necesitamos crear una serie de clases para guardar información acerca de una serie de figuras que serán círculos, cuadrados y triángulos. Además necesitamos poder tratar también grupos de imágenes porque nuestro programa permite seleccionar varias de estas figuras a la vez para moverlas por la pantalla.

En principio tenemos las clases Círculo, Cuadrado y Triángulo, que heredarán de una clase padre que podríamos llamar Figura e implementarán todas la operaciones pintar(). En cuanto a los grupos de Figuras podríamos caer en la tentación de crear una clase particular separada de las anteriores llamada GrupoDeImágenes, también con un método pintar().

Problema: Esta idea de separar en clases privadas componentes (figuras) y contenedores (grupos) tiene el problema de que, para cada uno de las dos clases, el método pintar() tendrá una implementación diferente, aumentando la complejidad del sistema.
El patrón Composite da una solución elegante a este problema, de la que además resulta en una implementación más sencilla.

A la clase Figura la llamaríamos Gráfico y de ella extenderían tanto Círculo, Cuadrado y Triángulo, como GrupoDeImágenes. Además, esta última tendría una relación todo-parte de multiplicidad * con Gráfico: un GrupoDeImágenes contendría varios Gráficos, ya fuesen éstos Cuadrados, Triángulos, u otras clases GrupoDeImágenes.

Así, es posible definir a un grupo de imágenes recursivamente. Por ejemplo, un objeto cuya clase es GrupoDeImágenes podría contener un Cuadrado, un Triángulo y otro GrupoDeImágenes, este grupo de imágenes podría contener un Círculo y un Cuadrado. Posteriormente, a este último grupo se le podría añadir otro GrupoDeImágenes, generando una estructura de composición recursiva en árbol, por medio de muy poca codificación y un diagrama sencillo y claro



```
import java.util.*;
public abstract class Componente
{
	protected String nombre;
	public Componente (String nombre)
	{
		this.nombre = nombre;
	}
	abstract public void agregar(Componente c);
	abstract public void eliminar(Componente c);
	abstract public void mostrar(int profundidad);
}
class Compuesto extends Componente
{
	private ArrayList<Componente> hijo = new ArrayList<Componente>();
	public Compuesto (String name)
	{
		super(name);
	}
	@Override
	public void agregar(Componente componente)
	{
		hijo.add(componente);
	}
	@Override
	public void eliminar(Componente componente)
	{
		hijo.remove(componente);
	}
	@Override
	public void mostrar(int profundidad)
	{
		System.out.println(nombre + " nivel: " + profundidad);
		for (int i = 0; i < hijo.size(); i++)
```

```
                    hijo.get(i).mostrar(profundidad + 1);
        }
}
class Hoja extends Componente
{
        public Hoja (String nombre)
        {
                super(nombre);
        }
        public void agregar(Componente c)
        {
                System.out.println("no se puede agregar la hoja");
        }
        public void eliminar(Componente c)
        {
                System.out.println("no se puede quitar la hoja");
        }
        public void mostrar(int depth)
        {
                System.out.println('-' + "" + nombre);
        }
}
public class Client
{
        public static void main(String[] args)
        {
                Compuesto raiz = new Compuesto("root");
                raiz.agregar(new Hoja("hoja A"));
                raiz.agregar(new Hoja("hoja B"));
                Compuesto comp = new Compuesto("compuesto X");
                comp.agregar(new Hoja("hoja XA"));
                comp.agregar(new Hoja("hoja XB"));
                raiz.agregar(comp);
                raiz.agregar(new Hoja("hoja C"));
```

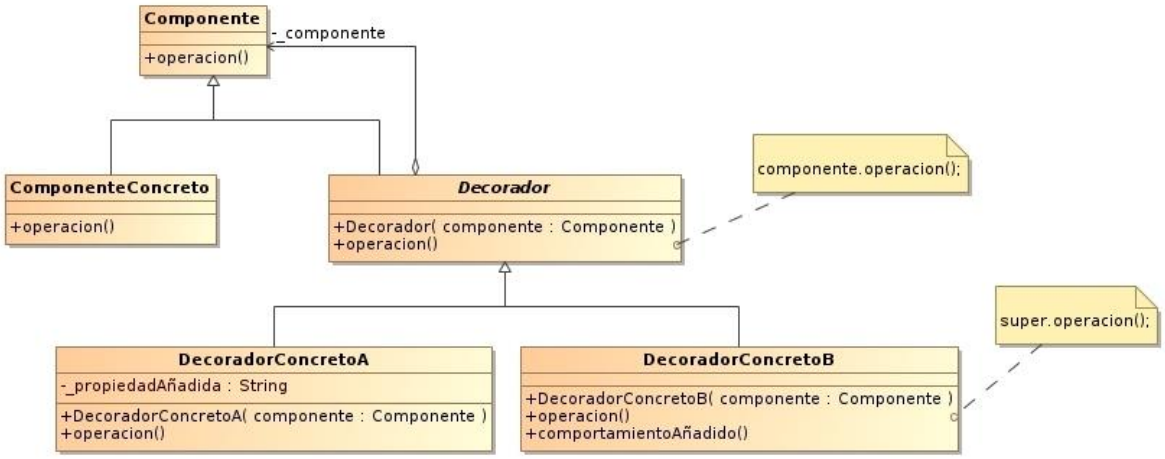
```
            Hoja l = new Hoja("hoja D");
            raiz.agregar(l);
            raiz.eliminar(l);
            raiz.mostrar(1);
        }
}
```

Decorator (Decorador) (Tiempo estimado: 3 hs)

Diseñado inicialmente para agregar responsabilidades a un objeto dado (en forma dinámica). Los decoradores proveen una alternativa flexible a “subclasear”en el proceso de extender funcionalidad.

El problema / Motivación

- Algunas veces queremos agregar responsabilidades individualmente a objetos, y no a una clase entera de la forma convencional a través de la generalización.
- Una implementación convencional (a través de una generalización) no podría controlar cómo o cuándo obtener el objeto decorado (con la responsabilidad deseada agregada).
- Añadir responsabilidades a objetos individuales de forma dinámica y transparente
- Responsabilidades de un objeto pueden ser retiradas
- Cuando la extensión mediante la herencia no es viable
- Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
- Existe la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida



.

Participantes

- **Componente**
  Define la interfaz para los objetos que pueden tener responsabilidades añadidas.

- **Componente Concreto**
  Define un objeto al cual se le pueden agregar responsabilidades adicionales.

- **Decorador**
  Mantiene una referencia al componente asociado. Implementa la interfaz de la superclase Componente delegando en el componente asociado.

- **Decorador Concreto**
  Añade responsabilidades al componente.

Colaboraciones

- El decorador redirige las peticiones al componente asociado.
- Opcionalmente puede realizar tareas adicionales antes y después de redirigir la petición.

Consecuencias

- Más flexible que la herencia. Al utilizar este patrón, se pueden añadir y eliminar responsabilidades en tiempo de ejecución. Además, evita la utilización de la herencia con muchas clases y también, en algunos casos, la herencia múltiple.
- Evita la aparición de clases con muchas responsabilidades en las clases superiores de la jerarquía. Este patrón nos permite ir incorporando de manera incremental responsabilidades.
- Genera gran cantidad de objetos pequeños. El uso de decoradores da como resultado sistemas formados por muchos objetos pequeños y parecidos.
- Puede haber problemas con la identidad de los objetos. Un decorador se comporta como un envoltorio transparente. Pero desde el punto de vista de la identidad de objetos, estos no son idénticos, por lo tanto no deberíamos apoyarnos en la identidad cuando estamos usando decoradores.

Implementación

El patrón Decorator soluciona este problema de una manera mucho más sencilla y extensible.

Se crea a partir de Ventana la subclase abstracta VentanaDecorator y, heredando de ella, BordeDecorator y BotonDeAyudaDecorator. VentanaDecorator encapsula el comportamiento de Ventana y utiliza composición recursiva para que sea posible añadir tantas "capas" de Decorators como se desee. Podemos crear tantos Decorators como queramos heredando de VentanaDecorator.

```
public abstract class Componente{
    abstract public void operacion();
 }

 public class ComponenteConcreto extends Componente{
    public void operacion(){
        System.out.println("ComponenteConcreto.operacion()");
    }
 }

 public abstract class Decorador extends Componente{
    private Componente componente;

    public Decorador(Componente componente){
```

```
        this.componente = componente;
    }
    public void operacion(){
        componente.operacion();
    }
}

public class DecoradorConcretoA extends Decorador{
    private String propiedadAñadida;

    public DecoradorConcretoA(Componente componente){
        super(componente);
    }
    public void operacion(){
        super.operacion();
        this.propiedadAñadida = "Nueva propiedad";
        System.out.println("DecoradorConcretoA.operacion()");
    }
}

public class DecoradorConcretoB extends Decorador{
    public DecoradorConcretoB(Componente componente){
        super(componente);
    }
    public void operacion(){
        super.operacion();
        comportamientoAñadido();
        System.out.println("DecoradorConcretoB.operacion()");
    }
    public void comportamientoAñadido(){
        System.out.println("Comportamiento B añadido");
    }
}

public class Cliente{
    public static void main(String[] args){
        ComponenteConcreto c = new ComponenteConcreto();
        DecoradorConcretoA d1 = new DecoradorConcretoA(c);
        DecoradorConcretoB d2 = new DecoradorConcretoB(d1);
        d2.operacion();//output:       "ComponenteConcreto.operacion()\n
DecoradorConcretoA.operacion()\n      Comportamiento      B      añadido\n
DecoradorConcretoB.operacion()"
    }
}
```

Proxy (Tiempo estimado: 3 hs)

El patrón Proxy es un patrón estructural que tiene como propósito proporcionar un subrogado o intermediario de un objeto para controlar su acceso.

El problema / Motivación

Para explicar la motivación del uso de este patrón veamos un escenario donde su aplicación sería la solución más adecuada al problema planteado. Consideremos un editor que puede incluir objetos gráficos dentro de un documento. Se requiere que la apertura de un documento sea rápida, mientras que la creación de algunos objetos (imágenes de gran tamaño) es costosa. En este caso no es necesario crear todos los objetos con imágenes nada más abrir el documento porque no todos los objetos son visibles. Interesa por tanto retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario (por ejemplo, no abrir las imágenes de un documento hasta que no son visibles). La solución que se plantea para ello es la de cargar las imágenes bajo demanda. Pero, ¿cómo cargar las imágenes bajo demanda sin complicar el resto del editor? La respuesta es utilizar un objeto proxy. Dicho objeto se comporta como una imagen normal y es el responsable de cargar la imagen bajo demanda.
El patrón proxy se usa cuando se necesita una referencia a un objeto más flexible o sofisticada que un puntero. Dependiendo de la función que se desea realizar con dicha referencia podemos distinguir diferentes tipos de proxies:

- proxy remoto: representante local de un objeto remoto.
- proxy virtual: crea objetos costosos bajo demanda (como la clase ImagenProxy en el ejemplo de motivación).
- proxy de protección: controla el acceso al objeto original.
- proxy de referencia inteligente: sustituto de un puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto (ej. contar número de referencias al objeto real, cargar un objeto persistente bajo demanda en memoria, control de concurrencia de acceso tal como bloquear el objeto para impedir acceso concurrente, …).

Participantes

La clase Proxy : mantiene una referencia al objeto real (en el siguiente ejemplo se le denomina _sujetoReal) y proporciona una interfaz idéntica al sujeto (la clase Sujeto). Además controla el acceso a dicho objeto real y puede ser el responsable de su creación y borrado. *También tiene otras responsabilidades que dependen del tipo de proxy:*

proxy remoto: responsable de codificar una petición y sus argumentos, y de enviarla al objeto remoto.
proxy virtual: puede hacer caché de información del objeto real para diferir en lo posible el acceso a este.
proxy de protección: comprueba que el cliente tiene los permisos necesarios para realizar la petición.

La clase Sujeto: define una interfaz común para el proxy (Proxy) y el objeto real (de la clase SujetoReal), de tal modo que se puedan usar de manera indistinta.

La clase SujetoReal: clase del objeto real que el proxy representa.

```
/**
 * Clase Cliente:  el cliente del sujeto tan solo conoce que maneja un objeto de la
 * clase Sujeto. Por tanto, funciona indistintamente con el SujetoReal
 * como con su Proxy.
 */

public class Cliente {

  /**
   * El constructor guarda la referencia al sujeto.
   */

  public Cliente(Sujeto sujeto) {
        _sujeto = sujeto;
  }
 /**
    * Lo único que realiza el cliente en su método ejecutar es
    * llamar 2 veces al metodo1, luego 1 vez al metodo2,
    * y de nuevo al metodo1. _sujeto, ejecutará el método dependiendo
    * si fue creado como proxy o sujetoReal.
    */

  public void ejecutar() {
        _sujeto.metodo1();
        _sujeto.metodo1();
        _sujeto.metodo2();
        _sujeto.metodo1();
  }

 /**
    *  La clase Cliente tiene el atributo _sujeto que le permite tener una referencia al sujeto al que
    *  el cliente envía la petición de ejecutar un determinado método.
    */

     private Sujeto _sujeto;

 }

/**
    * La clase Sujeto dentro del patrón Proxy es la interfaz del sujeto
    * real de cara al exterior (Cliente). Es una clase abstracta cuyos
```

```
   * métodos serán implementados tanto por el sujeto real como por el
proxy.
   */

   public abstract class Sujeto {

  /**
    * El constructor guarda el nombre del sujeto.
    */
     public Sujeto(String nombre) {
          _nombre = nombre;
    }

  /**
   * Método que devuelve el nombre del sujeto.
   */
     public String toString() {
           return _nombre;
    }

  /**
   * Métodos definidos de forma abstracta en la clase Sujeto, y que
tendrán distintas implementaciones en las clases que heredan de ésta:
Proxy
   * y SujetoReal.
   */
     public abstract void metodo1();
     public abstract void metodo2();

  /**
   * Este método llama al método toString() de la clase Proxy. Se le
pasa un objeto de la clase Sujeto, pero se considera que se trata de un
objeto proxy.
   */

     public void status (Sujeto sujeto) {
          Proxy p;
          p = (Proxy) sujeto;
          p.toString();
     }

  /**
   *  La clase Sujeto tiene el atributo _nombre , que indica el nombre
de un sujeto,  tanto si se trata de un proxy
   *  como de un sujeto real.
   */
     private String  _nombre;
```

```
}

/**
 * Éste es el objeto Proxy. Este proxy es simultáneamente un
 *
 * (a) proxy virtual que retrasa la creación del objeto real hasta que
 *     se invoca alguno de sus métodos.
 * (b) referencia inteligente, realizando labores de contabilización
 *     del número de veces que se invoca un método.
 */

public class Proxy extends Sujeto {
   /**
    * el constructor de la clase, además de inicializar a la parte
    * correspondiente a la superclase, establece a null la referencia
    * al sujeto real e inicializa la contabilización.
    */

  public Proxy (String nombre) {
     super(nombre);
    _sujetoReal = null;
   _accesosMetodo1 = 0;
   _accesosMetodo2 = 0;
  }

  /**
   * En lugar de realizar de cada vez una comprobación de si el
   * sujeto real esta creado y en caso contrario crearlo, se define
   * este método privado.
   */
  private SujetoReal obtenerSujetoReal() {
    if (_sujetoReal == null)
      _sujetoReal = new SujetoReal(this + " (Real)");

    return _sujetoReal;
  }
 /**
   * Los métodos delegan en el sujeto real.
   */
  public void metodo1() {
    _accesosMetodo1++;
    obtenerSujetoReal().metodo1();
  }
  public void metodo2() {
    _accesosMetodo2++;
    obtenerSujetoReal().metodo2();
  }
```

```
  /**
    * Este método permite presentar información de contabilización
    * de uso del objeto.
    */
  public String toString() {
    if (_sujetoReal != null)
      System.out.println("Accesos a " + _sujetoReal +
                         ": metodo1=" + _accesosMetodo1 +
                         ", metodo2=" + _accesosMetodo2);
    else
      System.out.println("Sujeto Real (" + this + ") no creado.");

    return "";
  }

  /**
   * Atributos privados: _sujetoReal que le permite a la clase Proxy
tener una referencia al sujeto real y los contabilizadores de los
accesos a
  * los métodos 1 y 2.
   */

  private SujetoReal _sujetoReal;
  private int  _accesosMetodo1, _accesosMetodo2;
}
/**
 * La clase SujetoReal es el objeto sobre el que queremos
 * implementar un proxy. Extiende la clase Sujeto implementando los
 * métodos del sujeto (en realidad es el sujeto el que presenta
 * la interfaz de los métodos del sujeto real...)
 */

public class SujetoReal extends Sujeto {
  public SujetoReal(String nombre) {
     super(nombre);
     // aquí aparece el código de una inicialización costosa: por
ejemplo,  añadir un objeto (que se le pasase como parámetro al
constructor) a
     // un vector que tuviese como atributo esta misma clase
(SujetoReal), y luego ordenar dicho vector de
     // mayor a menor en función de un atributo entero que tuviese la
clase a la que pertenecen los objetos que contiene el vector.
     // Otro ejemplo de inicialización costosa sería el llamar en el
constructor a un método de esta clase: por
     //  ejemplo loadImageFromDisk() lo cúal sería lógico si se
tratase de una clase ImagenReal que tuviese como proxy la clase
ProxyReal y
     // como clase abstracta de la que hereda, la clase Imagen.
```

```
    }
    public void metodo1()  {
        System.out.println("Ejecutando metodo1 en " + this);
    }
    public void metodo2() {
        System.out.println("Ejecutando metodo2 en " + this);
    }
}

public class Main {
    public static void main(String argv[]) {
      Sujeto objetoA = new Proxy("objetoA");
      Cliente c = new Cliente(objetoA);
      objetoA.status(objetoA);
      c.ejecutar();
      objetoA.status(objetoA);

      Sujeto objetoB = new SujetoReal("objetoB");
      Cliente d = new Cliente(objetoB);
      d.ejecutar();
    }
}
```

Bridge (Puente) (Tiempo estimado: 2 hs)

Nota: sugerido para trabajo autónomo. Referencia a ejercicio con Swing (skins multiplataforma).