Patrones de Diseño (Design Patterns)

Martín Pérez Mariñán

Sumario

Los patrones de diseño (del inglés Design Patterns) como veremos son modelos de trabajo enfocados a dividir un problema en partes de modo que nos sea posible abordar cada una de ellas por separado para simplificar su resolución. En este articulo intentaré describir los más importantes de modo que podamos comprobar como a veces el utilizar este tipo de modelos nos puede facilitar mucho la vida. Este articulo está enfocado a cualquier lenguaje de programación orientado a objetos (no sólo a Java) así como a programadores y desarrolladores de cualquier nivel. Espero que os guste.

Desde principios de 1980 cuando Smalltalk era “el rey” de la programación orientada a objetos y C++ estaba todavía en pañales se empezaron a buscar modelos como el archiconocido MVC encaminados a la división de un problema en partes para poder analizar cada una por separado. Dividir un problema en partes siempre ha sido uno de los objetivos de una buena programación orientada a objetos, si alguno de vosotros ha intentado hacer esto, probablemente ya haya utilizado muchos de los patrones que veremos.

Los patrones de diseño empezaron a reconocerse a partir de las descripciones de varios autores a principios de 1990. Este reconocimiento culmina en el año 1995 con la publi*cación del libro "Design Patterns -- Elements of Reusable Software"* de Gamma, Helm, Johnson y Vlissides; este libro puede considerarse como el más importante realizado sobre patrones de diseño hasta el momento.

Una posible definición de patrón de diseño sería la siguiente :

Un patrón de diseño es un conjunto de reglas que describen como afrontar tareas y solucionar problemas que surgen durante el desarrollo de software.

Existen varias definiciones alternativas pero creo que esta puede describir bastante bien este tipo de modelos. Vamos a considerar tres conjuntos de patrones según su finalidad :

- **Patrones de creación :** Estos patrones crearán objetos para nosotros de manera que ya no los tendremos que instanciar directamente, proporcionando a nuestros programas una mayor flexibilidad para decidir que objetos usar.
- **Patrones escructurales :** Nos permiten crear grupos de objetos para ayudarnos a realizar tareas complejas.
- **Patrones de comportamiento :** Nos permiten definir la comunicación entre los objetos de nuestro sistema y el flujo de la información entre los mismos.

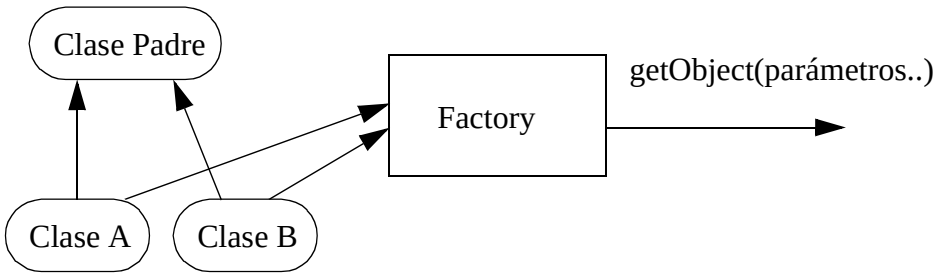
A continuación describiré algunos de los patrones más utilizados dentro de cada uno de los grupos junto con ejemplos de su utilización y su presencia dentro del lenguaje de programación Java. De todos modos, el lector puede consultar la bibliografía para obtener más información sobre el tema.

Patrones de creación

Como ya he comentado anteriormente todos los patrones de creación se encargan de crear instancias de objetos por nosotros. Los patrones de creación más conocidos son : ***Factory***, ***Abstract Factory***, ***Builder***, ***Prototype*** y ***Singleton***.

Factory

Este tipo de patrón se usa bastante debido a su utilidad. Su objetivo es devolver una instancia de múltiples tipos de objetos, normalmente todos estos objetos provienen de una misma clase padre mientras que se diferencian entre ellos por algún aspecto de comortamiento. El funcionamiento es muy simple y se puede observar en la siguiente figura (nótese que obviamente puede también devolver un único tipo de objeto) :



El objeto ***Factory*** será el encargado de decidir según los parámetros que le pasemos el tipo de objeto que nos devolverá.Veamos un pequeño programa de ejemplo :

```
public classVehiculo { // Clase padre
   double velocidad;
   double peso;
   .......
}

public class Camion extends Vehiculo { // primera clase hija
   String tipoMercancia;
   .....
}

public class Coche extends Vehiculo { // segunda clase hija
   String asientos;
}
```

```
public classVehiculoFactory {
   ....
   publicVehiculo getVehiculo(int tipo) {
      if (tipo = = VehiculoFactory.CAMION ) {
         return new Camion();
      } else {
         return new Coche();
      }
   }
}
```

Como se puede observar en este ejemplo tenemos una clase Factory que nos devolverá un tipo de vehículo determinado según el parámetro que le pasemos y en base a constantes definidas en la propia clase.

Abstract Factory

Este patrón añade un nivel más de complejidad. Si teníamos que una clase ***Factory*** nos devolvía objetos de diferentes tipos, este patrón lo que va a hacer es devolvernos diferentes clases ***Factory*** según algún parámetro que le proporcionemos. Un ejemplo son los Look&Feel. En nuestro sistema podemos tener una clase ***Abstract Factory*** que nos devuelva diferentes objetos Look&Feel, cada uno específico para una plataforma (Windows, Linux, Mac, ...). A su vez, estos objetos pueden ser clases ***Factory*** que nos devuelven los diferentes componentes correspondientes a cada una de esas plataformas (ej. el Look&Feel de Windows nos devolvería botones, diálogos, cuadros de texto con el aspecto de Windows, etc...).

Singleton

Este patrón es uno de los más utilizados y es muy común encontrarlo por toda la bibliografía sobre Java. Bien, un ***Singleton*** es una clase de la que tan sólo puede haber una **única** instancia.[1] Ejemplos típicos de esto son spools de impresión, servidores de bases de datos, etc.. Se puede afrontar este problema de varias formas, veamos las más comunes :

- **Crear una variable estática** dentro de la clase que nos indique si una instancia ha sido o no creada. Esta solución tiene el problema de como avisar desde el constructor de que no se ha podido crear una nueva instancia. Veamos un ejemplo, en el que se lanza una excepción si no se puede crear :

```
public class Singleton1 {
   public static boolean flag = false; // flag de creación

   public Singleton1() throws Exception {
      if (flag) {
         throw new Exception("Ya existe una instancia");
      } else {
         flag = true;
      }
   }

   public void finalize() {
      flag = false;
   }
}
```

- **.Crear una clase final :** El objetivo de esto es crear una clase final que tan sólo tenga métodos estáticos. De este modo la clase no se podrá extender. [2] Un ejemplo de esto es la clase ***java.lang.Math***, que agrupa métodos matemáticos de utilidad de manera que sólo haya una única forma de acceder a los mismos.

1. En este caso entendemos una única instancia dentro de la JVM, no de la aplicación.
2. Ciertamente en este caso se pueden crear más instancias de la clase, pero los métodos seguirán siendo los mismos para todas. Para conseguir un verdadero **Singleton** hay que hacer que el constructor de la clase sea privado de modo que no se pueda instanciar (esto es lo que hace la clase Math).

```
public final class Singleton2 {
   public static int mul(double a, double b) {
      return a*b
   }
   public static double div(double a, double b) {
      return a/b;
   }
}
```

- **Crear el *Singleton* con un método estático :** Esta aproximación lo que hace es hacer privado el constructor de la clase de manera que la única forma de conseguir una instancia de la misma sea con un método estático. Podemos ver un ejemplo de esta elegante idea :

```
public class Singleton3 {
   public static boolean flag = false;
   public Singleton3 instance = null;

   private Singleton3() { .... }

   public Singleton3 getInstance() {
      if (flag) {
         return instance;
      } else {
         instance = new Singleton3();
         flag = true;
        return instance;
      }
   }

   public void finalize() {
      flag = false;
   }
}
```

Patrones estructurales

Los patrones estructurales nos describen como formar estructuras complejas a partir de elementos más simples. Existen dos tipos de patrones de este tipo, de clase y de objeto. Los patrones de clase nos muestran como la herencia puede ser utilizada para proporcio-

nar mayor funcionalidad mientras que los patrones de objeto utilizan composición de objetos o inclusión de objetos dentro de otros para proporcionar también una mayor funcionalidad.

Los patrones más conocidos son : ***Adapter, Bridge, Composite, Decorator, FaÇade, Flyweight y Proxy.***

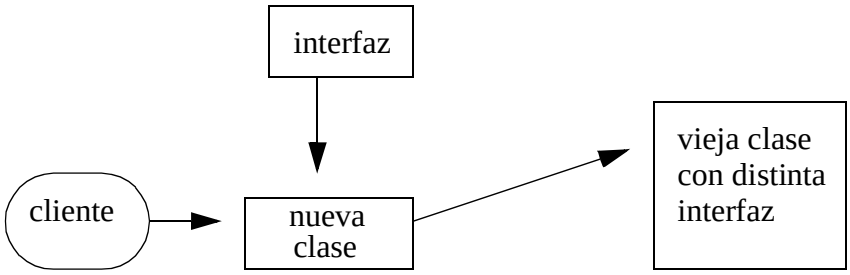
Adapter

Este patrón es de los más conocidos y utilizados dentro y fuera de Java. Su finalidad es transformar la interfaz de programación [1] de una clase en otra. Utilizaremos adaptadores cuando queramos que clases que no tienen nada que ver funcionen de la misma manera para un programa determinado. El concepto de un ***Adapter*** es simple : escribir una nueva clase con la interfaz de programación deseada y hacer que se comunique con la clase cuya interfaz de programación era diferente.

Existen dos formas de realizar esto, con **herencia** o con **composición de objetos**. En el primer caso vamos a crear una nueva clase que heredará de la que queremos adaptar y a esta nueva clase le añadiremos los métodos necesarios para que su interfaz de programación se corresponda con la que queremos utilizar. En la segunda aproximación vamos a incluir la clase original dentro de la nueva y crearemos los métodos de manera que accedan a la clase que hemos añadido como atributo. Estas dos formas se corresponden con los términos de **adaptadores de objeto** y **adaptadores de clase.**

1. Como interfaz de programación me refiero al conjunto de métodos que podemos invocar en una clase (nótese que no tiene nada que ver con **interface**)

La siguiente figura muestra el comportamiento de un adaptador de clase :



Veamos un ejemplo. Supongamos que tenemos una vieja clase con la que podíamos imprimir en nuestra vieja impresora textos simples en escala de grises. Como los tiempos cambían nuestra empresa habrá adquirido nuevas y modernas impresoras laser a color. Todos nuestros nuevos programas de impresión utilizan la siguiente interfaz :

```
public interface ImprimeColor {
   public void setColor(int r, int g, int b);
   public void setFont(Font f);
   public void setDuplex(boolean b);
   ..............
}
```

Nuestra vieja clase sin embargo no se parece en nada a esta interfaz :

```
public class ViejaImpresora {

   public void setGrayscale( double porcentaje) {
      .......
   }

   public void setFontNumber( int fontNumber) {
        .....
   }
   ..............
}
```

Imaginémonos que nuestro jefe nos avisa de que en caso de que haya problemas con alguna impresora nueva deberíamos sustituirla por una vieja y que al menos se pudiese imprimir en alguna de las viejas (obviamente no con la misma calidad ni formato) para de ese modo poder aprovechar todas esas impresoras y por lo tanto no tirarlas.

Para un problema de este tipo, una solución ideal sería crear un ***Adapter*** que adapte nuestra vieja clase a la nueva interfaz. La solución utilizando un adaptador de clase sería la siguiente :

```
public class NuevaImpresora extends ViejaImpresora implements ImprimeColor {

   public void setColor(int r, int g, int b) {
      setGrayscale((r+g+b)/3);
   }

   public void setDuplex(boolean b) {
      // la impresora no soporta duplex, no haremos nada
   }

   public void setFont(Font f) {
      // transformaríamos esta fuente a un número de fuente conocido por la vieja
      // impresora
      if (f.getName().equals("Arial")).setFontNumber(1);
      if (f.getName().equals("Times")).setFontNumber(2);
      ........
   }
}
```

y la solución con un adaptador de objeto sería muy parecida :

```
public class NuevaImpresora implements ImprimeColor {
   ViejaImpresora vieja = new ViejaImpresora();

   public void setColor(int r, int g, int b) {
      vieja.setGrayscale((r+g+b)/3);
   }
   ..... lo mismo pero utilizando vieja .....
```

Como veis con esta solución podremos utilizar nuestra vieja impresora sin tener que modificar ni una linea de código de todos los programas que hemos realizado, fantástico ¿ no creéis ?

Adaptadores en Java

Por último alguién todavía puede estar preguntandose en que se parecen estos ***Adapter*** a los que se pueden ver en Java. Bueno, como sabréis toda interfaz ***Listener*** con varios métodos tiene su correspondiente clase ***Adapter***. ¿ Por qué ? Bueno, si observáis por ejemplo la interfaz ***WindowListener*** veréis como tiene un gran número de métodos. Cuando queremos cerrar una ventana, tendremos que añadirle el correspondiente ***Listener*** a dicha ventana y sobreescribir el método ***windowClosing()*** del mismo :

```
public class MiFrame extends Frame implements WindowListener {
   public MiFrame() {
      addWindowListener(this);
   }
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
   public void windowClosed(WindowEvent e) { }
   public void windowOpened(WindowEvent e) {}
  ..... 4 métodos más ......
}
```

La pregunta es ¿ Por qué tengo que implementar todos los métodos de ***WindowListener*** si sólo me interesa el ***windowClosing()*** ? Aquí es donde entran en juego los ***Adapters***. Cada ***Listener*** con más de un método tiene su correspondiente clase ***Adapter***[1], de modo que éste adapta una nueva clase a lo que queremos, es decir crea una nueva clase con

1. No tendría sentido añadir adaptadores a listeners con un único método ya que sería el que el listener sobreescribe.

todos los métodos de la interfaz vacíos de modo que podamos sobreescribir los métodos que querramos y sólamente eses, ejemplo :

```
public class WindowAdapter implements WindowListener {
   public void windowClosing(WindowEvent e) { }
   public void windowClosed(WindowEvent e) { }
   public void windowOpened(WindowEvent e) { }
   ..... 4 métodos más .....
}
public class MiFrame extends Frame implements WindowAdapter {
   public MiFrame() {
      addWindowListener(this);
   }
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}
```

Como se habrá podido comprobar este es un patrón de diseño muy simple que nos proporciona grandes posibilidades

Bridge

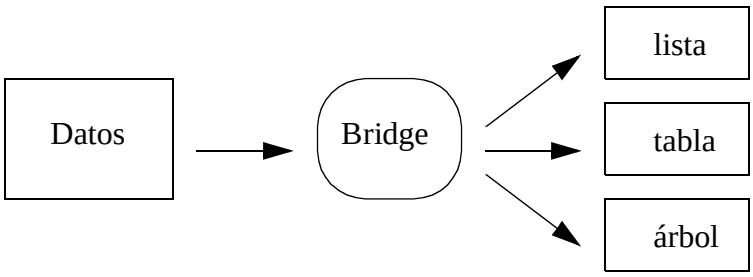
Un ***Bridge*** se utiliza para separar la interfaz de una clase de su implementación de forma que ambas puedan ser modificadas de manera separada, el objetivo es poder modificar la implementación de la clase sin tener que modificar el código del cliente de la misma.

El funcionamiento del ***Bridge*** es simple, imaginémonos que tenemos datos de los clientes de nuestra empresa y los queremos mostrar en pantalla. En una opción de nuestro programa queremos mostrar esos datos en una lista con sus nombres, mientras que en otra queremos mostrar los datos en una tabla con nombre y apellidos. Con un ***Bridge*** tendríamos lo siguiente :

```
public void muestraDatos(Vector datos) {
   MiBridge bridge = new MiBridge(datos, MiBridge.LIST);
   Frame f = new Frame();
   f.add(brige);
}

public class MiBridge extends JScrollPane {
   public static final LIST = 0;
   public static final TABLE = 1;
   public MiBridge(Vector datos, int modo) {
      if (modo == LIST) {
         add(new JList());
      } else {
         add(newJTable());
      }
   }
}
```

Ni que decir tiene que habría que añadir toda la creación e inicialización de listas y tablas. Lo importante de este patrón es que si ahora quisiésemos mostrar información de nuestros clientes en un árbol, no tendríamos que modificar el cliente para nada sino que en nuestro ***Bridge*** añadiríamos un parámetro que nos crearía el árbol por nosotros.

Gráficamente :



Proxy

Un patrón ***Proxy*** lo que hace es cambiar un objeto complejo por otro más simple. Si crear un objeto es demasiado costoso en tiempo o recursos, ***Proxy*** nos permite posponer su creación hasta que sea realmente necesitado.Un ***Proxy*** tiene (normalmente) los mismos métodos que el objeto al que representa pero estos métodos sólamente son llamados cuando el objeto ha sido cargado por completo. Hay muchos casos donde un ***Proxy*** nos es útil :

- Si un objeto, como una imagen grande, puede tardar mucho en cargarse.
- Si un objeto se encuentra en una máquina remota sólamente accesible por red.
- Si el objeto tiene el acceso restringido el ***Proxy*** puede encargarse de validar los permisos.

Veamos un ejemplo muy ilustrativo. Imaginémonos que tenemos un interfaz como el de cualquier navegador de internet y que tenemos un panel en el cual queremos mostrar una imagen que es muy grande. Como sabemos que va a tardar bastante en cargarse utilizaremos un ***Proxy*** :

```
public class MiProxy extends Frame() {
   JPanel p = new JPanel();
   p.setLayout(new BorderLayout());
   this.getContentPane().add(p);
   ImageProxy imagen = new ImageProxy(this,"java.jpg",320,200);
   p.add(imagen);
}

// Dentro de ImageProxy... por motivos de espacio no pondré todo el código
public void paint(Graphics g) {
   if (tracker.checkId(0)) {
      // En este caso ya se ha cargado la imagen
      height = image.getHeight(...);
      width = image.getWidth(...);
```

```
        g.setColor(Color.lightGray);
        g.fillRect(0,0,width,height);
        g.drawImage(image,0,0,frame);
    } else {
        // No hemos podido cargar todavía la imagen
        g.drawRect(0,0,width-1,height-1);
    }
```

Como se puede intuir el ***Proxy*** en este caso abre un nuevo hilo en el que utilizando el ***MediaTracker*** intenta cargar la imagen. Mientras en el método ***paint()*** comprueba si se ha cargado la imagen, en caso afirmativo la muestra y si todavía no hemos podido cargarla muestra un rectángulo vacío.

Patrones de comportamiento

Los patrones de comportamiento fundamentalmente especifican el comportamiento entre los objetos de nuestro sistema. Los patrones más conocidos son : ***Chain, Observer, Mediator, Template, Interpreter, Strategy, Visitor, State, Command*** e ***Iterator.*** Veremos a continuación algunos de ellos :

Command

El patrón ***Command*** especifica una forma simple de separar la ejecución de un comando del entorno que generó dicho comando. ¿ Por qué utilizar este patrón ? Bien, muchas veces en una interfaz nos podemos encontrar con algo como lo siguiente :

```
public void actionPerformed(ActionEvent e) {
    Object obj = e.getSource();
    if (e == menu) {
        menuElegido();
    }
```

```
    if (e == boton) {
        botonPulsado();
    }
    if (e == combo) {
        opcionElegida();
    }
}
```

Si el número de eventos que pueden producirse en nuestra interfaz es pequeño esta aproximación puede ser aceptable, pero si el número de eventos que pueden producirse es considerablemente grande está aproximación nos originará un código muy confuso y extenso. Además no parece demasiado orientado a objetos él que las acciones sean elegidas y controladas por el interfaz de usuario sino que deberían ser independientes.

Una forma para corregir esto es el patrón ***Command***. Un objeto ***Command*** siempre consta de un método ***Execute()*** que se llama siempre que se produzca una acción sobre dicho objeto. La interfaz de este patrón es pues la siguiente :

```
public interface Command {
    public void execute();
}
```

De este modo, al utilizar este patrón, el método ***actionPerformed()*** se reduce a :

```
public void actionPerformed(ActionEvent e) {
    Command accion = (Command)e.getSource();
    accion.execute();
}
```

Como se puede apreciar, no hay nada en el interfaz que nos indique que acción estamos realizando, es decir, hemos conseguido separar nuestras acciones del interfaz gráfico. Ahora ya podemos crear un método ***execute()*** para cada uno de los objetos que quiera realizar una acción de modo que sea dicho objeto el que controle su funcionamiento y no

otra parte cualquiera del programa como vimos en el primer ejemplo. Por ejemplo, un botón que siguiese este patrón sería :

```
import java.util.Random;

public class NuevoBoton extends JButton implements Command {
   Random r = new Random();
   public NuevoBoton(String caption) {
      super(caption);
   }
   .......
   public void execute() {
      setBackground(r.nextInt(255),r.nextInt(255),r.nextInt(255));
   }
}
```

Podríamos ahora construir un frame de muestra :

```
public class MiFrame extends JFrame implements ActionListener {
   public MiFrame() {
      MiBoton b = new MiBoton("Pulsame ya");
      b.addActionListener(this);
      this.getContentPane().add(b);
      setSize(500,400);
      setVisible(true);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
   public void actionPerformed(ActionEvent e) {
      Command accion = (Command)e.getSource();
      accion.execute();
   }
}
```

Iterator

Este patrón es uno de los más simples y más frecuentes. Su función es permitirnos recorre un conjunto de datos mediante una interfaz estandar sin tener que conocer los detalles de la implementación de los datos. Además siempre podremos definir iteradores espe-

ciales que realicen algún tipo de acción con los datos o que sólo nos devuelvan elementos determinados. En java este patrón se encuentra implementado en las interfaces ***Enumeration e Iterator*** :

```
public interface Enumeration {
   public boolean hasMoreElements();
   public Object nextElement();
}
```

Una desventaja de esta interfaz respecto a la implementación de ***Iterator*** en otros lenguajes es que no posee métodos para ir al principio y al final de la enumeración de modo que si se quiere volver a empezar habrá que adquirir una nueva instancia de la clase.

En Java tenemos el patrón ***Enumeration*** presente en clases como ***Vector*** o ***Hashtable***. Para obtener una enumeración de todos los elementos de un vector llamaríamos al método ***elements()*** que devuelve un tipo ***Enumeration***. Este método ***elements()*** es un método que ejerce como ***Factory*** devolviendonos distintas instancias de tipo ***Enumeration***. Para recorrer la enumeración haríamos :

```
for (Enumeration e = vector.elements() ; e.hasMoreElements(); ) {
   String objeto = e.nextElement().toString();
   System.out.println(objeto);
}
```

Una opción muy interesante es añadir ***filtros de iteración*** de modo que antes de devolver una enumeración con los elementos realicemos cualquier otro tipo de acción con ellos como por ejemplo una ordenación por algún criterio. Imaginemonos que hemos hecho un programa que maneja los datos de los chicos de una escuela, en este programa tenemos una clase que contiene los datos de los chicos :

```
public class DatosChicos {
```

```
    Vector chicos;
    public DatosChicos(String fichero) {
        ..... cargar datos del fichero en el vector .....
    }
    public Enumeration elements() {
        return kids.elements();
    }
}
```

Ahora supongamos que queremos listar los chicos que estudian en un aula determinada, podríamos hacerlo como a continuación :

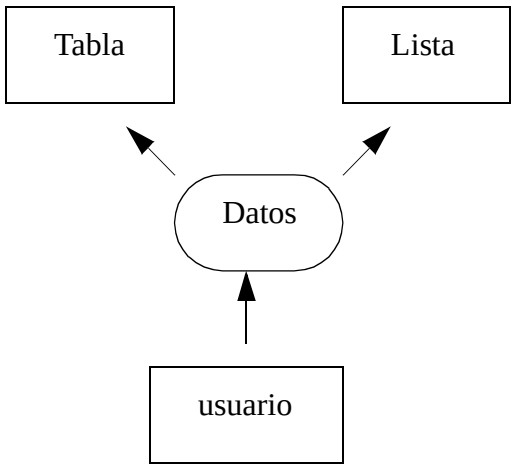
```
public class AulaChicos implements Enumeration {
    String aula;                // filtro
    Chico chicoActual;          // chico que devolvemos
    Enumeration eChicos;        // enumeración
    DatosChicos chicos;         // datos de todos

    public AulaChicos(DatosChicos dc, String filtro) {
        aula = filtro;
        chicos = dc;
        chicoActual = null;
        eChicos = dc.elements();
    }

    public boolean hasMoreElements() {
        // busca el siguiente chico que pertenezca al aula del filtro
        boolean found = false;
        while (eChicos.hasMoreElements() && !found) {
            chicoActual = (Chico)eChicos.nextElement();
            found = chicoActual.getAula().equals(aula);
        }
        return found;
    }

    public Object nextElement() {
        if (chicoActual != null)
            return chicoActual;
        else
            throw new NoSuchElementException();
    }
}
```

Ahora simplemente tendríamos que añadir en la clase ***DatosChicos*** un nuevo método que nos devuelva otra ***Enumeration*** :

```
public Enumeration chicosAula(String aula) {
   return new AulaChicos(this,aula);
}
```

Observer

En la actualidad es común encontrarnos con aplicaciones que muestran simultáneamente un conjunto de datos de múltiples formas diferentes en el mismo interfaz (por ejemplo en forma de árbol, tabla, lista, ...). El patrón ***Observer*** asume que el objeto que contiene los datos es independiente de los objetos que muestran los datos de manera que son estos objetos los que "*observan*" los cambios en dichos datos :



Al implementar el patrón ***Observer*** el objeto que posee los datos se conoce como **sujeto** mientras que cada una de las vistas se conocen como **observadores**. Cada uno de estos observadores, registra su interés en los datos llamando a un método público del **sujeto**.

Entonces, cuando estos datos cambian, el **sujeto** llama a un método conocido de la interfaz de los observadores de modo que estos reciben el mensaje de que los datos han cambiado y actualizan la vista.

```
abstract interface Observer {
   // avisa a los observadores de que se ha producido un cambio
   public void sendNotify(String s);
}

abstract interface Subject {
   // avisa al sujeto de tu interés
   public void registerInterest(Observer obs);
}
```

Podemos ver a continuación un ejemplo muy simple, tendremos un botón que será nuestro sujeto y varios paneles que serán los observadores y que cambiarán el color con el botón :

```
public class MiBoton extends JButton implements Subject {
   Random r = new Random();
   ArrayList observadores = new ArrayList();
   Color c = Color.red;

   public MiBoton(String nombre) {
      super(nombre);
      setForeground(c);
   }

   public void cambiaColor() {
      c = new Color(r.nextInt(255),r.nextInt(255),r.nextInt(255));
      for (int i=0;i<observadores.size();i++) {
         ((Observer)observadores.get(i)).sendNotify("cambio de color");
      }
   }

   public Color getColor() {
      return c;
   }
```

```
    public void registerInterest(Observer obs) {
      observadores.add(obs);
    }
}

class MiPanel extends JPanel implements Observer {

    MiBoton b = null;

    public MiPanel(MiBoton b) {
      super();
      this.b = b;
      b.registerInterest(this);
    }

    public void sendNotify(String mensaje) {
      setBackground(b.getColor());
    }
}

public class MiFrame extends JFrame {

    public MiFrame() {
      final MiBoton b = new MiBoton("Pulsame ya");
      b.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
             b.cambiaColor();
          }
      });
      this.getContentPane().add(b);
      this.getContentPane().add(new MiPanel(b),BorderLayout.NORTH);
      this.getContentPane().add(new MiPanel(b),BorderLayout.SOUTH);
      this.getContentPane().add(new MiPanel(b),BorderLayout.EAST);
      this.getContentPane().add(new MiPanel(b),BorderLayout.WEST);
      setSize(500,400);
      setVisible(true);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
      new B();
    }
}
```

En Swing los objetos ***JList, JTable*** y ***JTree*** actúan como observadores de un modelo de datos De hecho todos los objetos que dervien de ***JComponent*** pueden realizar esta separación entre vista y datos, esto se conoce como **arquitectura MVC** (Modelo-Vista-Controlador), donde los datos son representados por el modelo y la vista por el componente visual. En este caso el controlador es la comunicación entre el modelo y la vista. Por último decir podemos usar las clases Observer y Observable del paquete java.util.

Strategy

El patrón ***Strategy*** consiste en un conjunto de algoritmos encapsulados en un contexto determinado ***Context***. El cliente puede elegir el algoritmo que prefiera de entre los disponibles o puede ser el mismo objeto ***Context*** el que elija el más apropiado para cada situación. Cualquier programa que ofrezca un servicio o función determinada, la cual puede ser realizada de varias maneras, es candidato a utilizar el patrón ***Strategy***. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento. Hay muchos casos en los que este patrón puede ser útil :

- Grabar ficheros en diferentes formatos
- Utilizar diferentes algoritmos de compresión
- Capturar video utilizando esquemas de captura diferentes
- Mostrar datos utilizando formatos diversos
- .....

Como ejemplo (no veremos el código) podemos imaginarnos un programa matemático que tiene en una estructura los datos de una función. Queremos mostrar los datos de esta función utilizando diagramas de barras, lineas, de tarta, ...

Como cada dibujo saldrá en un ***Frame*** vamos a hacer que nuestra estrategia herede de dicha clase :

```
public abstract class EstrategiaDibujo extends JFrame {
   protected float[] x,y;
   protected Color c;
   protected int width,height;

   public EstrategiaDibujo(String titulo) {
      ....
   }
   public abstract void plot(float[] px, float[] py);
   ......
}
```

Lo importante de esta clase es que cada una de las estrategias que diseñemos tendrá que sobreescribir el método ***plot()*** y proveer un algoritmo concreto para dicha estrategia.

El **contexto** es la clase que decide que estrategía utilizar en cada momento, la decisión se realiza normalmente mediante algún parámetro que le envía el cliente aunque como hemos dicho puede ser él mismo el que elija la estrategia más adecuada :

```
public class Contexto {
    private EstrategiaDibujo estrategia;
    float[] x,y;

   public Contexto() {
      // Establecer estrategia por defecto
   }
   public void setDibujoBarras() {
      estrategia = new EstrategiaDibujoBarras();
   }
   public void setDibujoLineas() {
      estrategia = new EstrategiaDibujoLineas();
   }
   ..............
   public void dibuja() {
      estrategia.(x,y);
   }
```

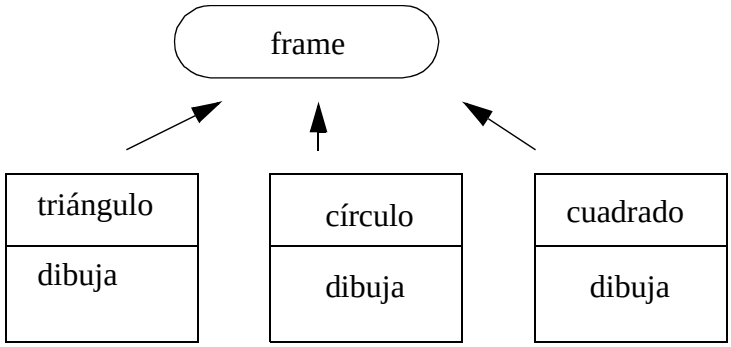
```
}
```

Como podemos comprobar el funcionamiento de este patrón es muy simple y el añadir nuevas estrategias a nuestro es muy sencillo y apenas implica modificación de código alguna.

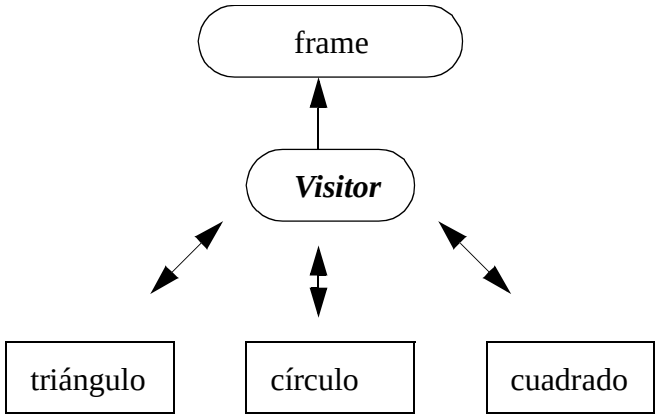
Visitor

Este patrón es muy curioso ya que aparenta romper con la programación orientada a objetos en el sentido de que crea una clase externa que va a actuar sobre los datos de otras clases. En un principio no parece correcto el poner operaciones que deberían estar en una clase en otra diferente, pero a veces existen razones de peso para esto.

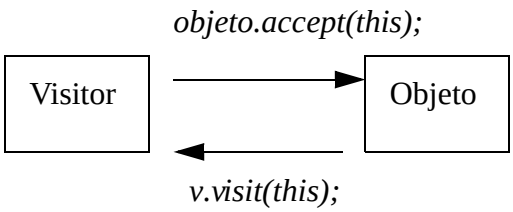
Imaginemos que tenemos varios objetos gráficos con métodos de dibujo muy similares. Los métodos de dibujo pueden ser diferentes pero seguramente haya una serie de funciones comunes en todos ellos, funciones que probablemente tendremos que repetir una y otra vez en su código, probablemente tendríamos un esquema así :



Sin embargo si escribimos una clase ***Visitor*** será ella la que contenga los métodos de dibujo y tendrá que ir *"visitando"* los objetos en orden y dibujándolos :



Lo que mucha gente al ver este patrón se pregunta es : ¿ qué significa visitar a los objetos ?; bien, esto lo que quiere decir es que tenemos que llamar a un método público de cada uno de los objetos diseñado para tal efecto, ***accept()***. El método ***accept()*** tiene un único argumento, la instancia del objeto ***Visito*** , y en su interior se hace una llamada al método ***visit()*** del ***Visitor*** pasándole como parámetro la instancia del objeto que estamos visitando.



Es decir, todo objeto que tenga que ser visitado ha de tener el siguiente método :

```
public void accept(Visitor v) {
   v.visit(this);
}
```

De esta forma el objeto ***Visitor*** obtiene una referencia a cada una de las instancias de los objetos una a una y puede llamar a los métodos públicos de las mismas para obtener los datos necesarios, realizar cálculos, generar informes o imprimir el objeto en pantalla.

Veamos un ejemplo, supongamos que tenemos un fichero con los empleados de nuestra empresa y queremos saber cuantos días de vacaciones han tomado en todal. Nuestra clase empleado puede ser así :

```
public class Empleado {
   int diasLibres, diasTrabajando;
   float sueldo;
   String nombre;

   ..... constructores, métodos get y set ...
   public void accept(Visitor v) {
      v.visit(this);
   }
}
```

Para calcular el total de días libres, vamos a utilizar un ***Visitor*** :

```
public abstract class Visitor {
   public abstract void visit(Empleado e);
}
```

Es importante notar que no hay nada ni en el empleado ni en la clase abstracta que nos diga lo que vamos a hacer con los empleados. Esto nos permite crear muchos ***Visitors*** diferentes los cuales pueden hacer diversas operaciones, uno puede ser el siguiente :

```
public class VacacionesVisitor extends Visitor {
   public int total;
   public VacacionesVisitor { total = 0; }
```

```
    public void visit(Empleado e) {
        total+=e.getDiasLibres();
    }

    public int getDiasLibres() {
        return total;
    }
}
```

Ahora dentro del programa principal lo único que tendríamos que hacer sería recorrernos todos los empleados y visitarlos :

```
VacacionesVisitor vacac = newVacacionesVisitor();
for (int i = 0; i < empleados.length; i++) {
    empleados[i].accept(vacac);
}
System.out.println("Total de días : " + vacac.getDiasLibres());
```

Conclusiones

Bueno espero que este artículo haya sido de gran utilidad para todos y que al menos alguno de todos los patrones que he explicado os sirva para vuestros desarrollos. Como habréis podido observar elegir el diseño de un problema antes de realizarlo y utilizar algunos patrones para resolverlo puede facilitar el desarrollo y solución del mismo, además de aumentar la escalabilidad del proyecto.

Los patrones de diseño como hemos podido comprobar nos permiten dividir nuestras aplicaciones en partes de manera que aunque en un primer momento nos puedan parecer demasiado abstractas, la mantenibilidad de las mismas crece exponencialmente.

A continuación se pueden ver enlaces sobre patrones de diseño donde se puede adquirir más información sobre los mismos. En este artículo tan sólo hemos visto tan sólo una pequeña introducción y una aproximación a algunos de los más utilizados.

Saludos, espero que os haya gustado.

Bibliografía y enlaces

Eric Gamma, Richard Helm, Ralph Johnson y John Vlissides, *Design Patterns. Elements of Reusable Software.*, Addison-Wesley, Reading, MA, 1995

Wolfgang Pree, *Design Patterns for Object Oriented Software Development*, Addison-Wesley, 1994.

S. Alpert, K. Brown y B. Woolf, *The Design Patterns Smalltalk Companion*, Addison-Wesley, 1998

Patrones en Java y C++, http://www.vico.org/pages/PatronsDisseny/Patrones.html

James W. Cooper, *The Design Patterns Java Companion*, http://www.patterndepot.com/put/8/JavaPatterns.htm

Bruce Eckel, *Thinking in Patterns*, http://www.mindview.net/Books/TIPatterns/