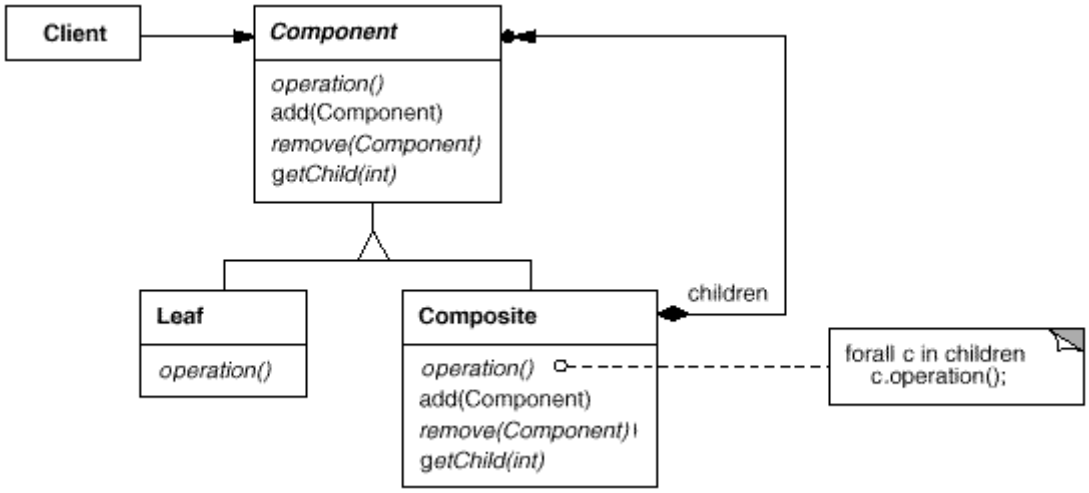
Composite

Diagrama de clases e interfaces



Intención

El patrón de diseño Composite (Compuesto) permite construir estructuras de objetos para representar jerarquías todo/parte. Por ejemplo, los directorios de un disco duro contienen entradas, cada una de las cuales puede ser un fichero o ser a su vez un directorio.

Composite permite que el código cliente gestione uniformemente tanto a los objetos simples (a los ficheros) como a los objetos compuestos (los directorios). Para ello utiliza recursividad y estructuras en forma de árbol.

Motivación

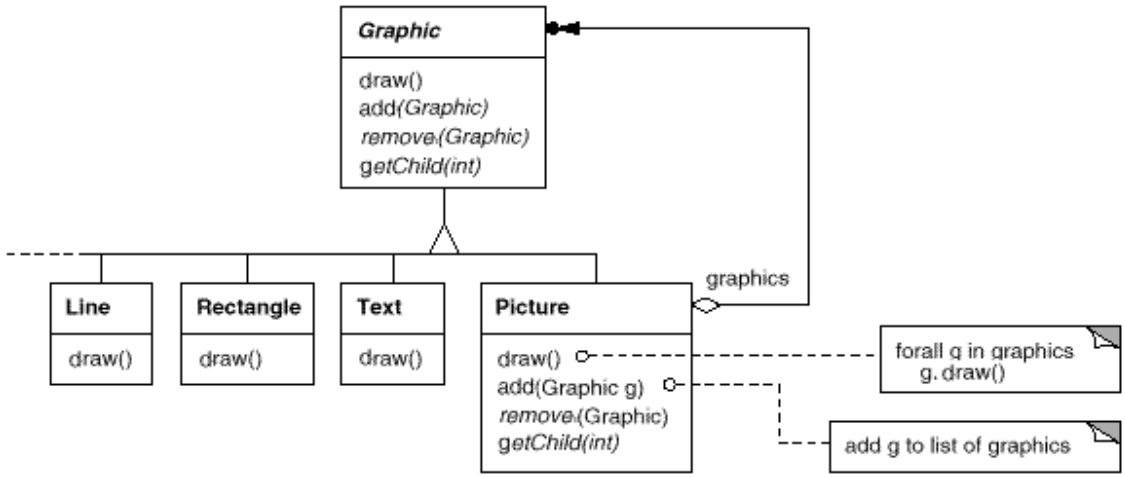
Muchas aplicaciones permiten a los usuarios construir componentes complejos a partir de componentes simples. El usuario puede agrupar componentes para formar nuevos componentes más grandes, que a su vez se pueden agrupar para formar componentes aún más grandes.

Las aplicaciones de edición gráfica son un ejemplo claro de este tipo de programas: se definen clases de elementos gráficos elementales, como la clase Linea, la clase Texto,

etc., además de otras clases que actúan como contenedores de éstas. Un contenedor sirve como recipiente a varios elementos gráficos, agrupándolos de alguna manera.

Sin embargo, hay un problema con este enfoque: El código cliente que usa estas clases debe tratar a los objetos elementales y a sus contenedores de manera diferenciada, aunque la mayoría de las veces el usuario los trata de forma idéntica. Tener que distinguir estos objetos complica la lógica de la aplicación.

El patrón Composite describe cómo utilizando composición recursiva se evita que el código cliente tenga que hacer esta distinción entre elemento simple y contenedor. En la figura siguiente tenemos un ejemplo de aplicación del patrón en una aplicación de edición gráfica.



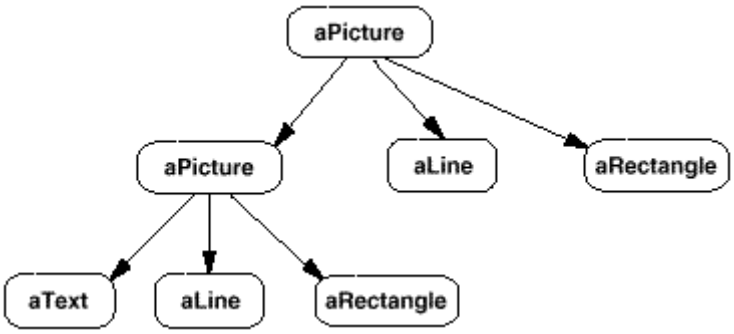
La clave del patrón Composite es una clase abstracta (o interfaz) que sirve de superclase tanto para las subclases simples como para las compuestas. Para el caso de la aplicación gráfica, esta clase es Graphic, la cual declara operaciones como draw() que son específicas de los objetos gráficos. También declara operaciones que serán redefinidas en las clases contenedoras, como las operaciones para acceder, añadir y eliminar los objetos que contienen.

Las subclases Linea, Rectangulo y Texto definen objetos gráficos elementales. Estas clases implementan el método draw() para dibujar líneas, rectángulos y texto, respectivamente. Ya que un objeto gráfico básico no contiene a otros elementos gráficos, ninguna de estas subclases implementa las operaciones relacionadas con la manipulación de colecciones, como add(), remove(), etc.

La clase Picture es un compuesto de objetos gráficos. Su implementación del método draw() consiste en invocar al método draw() de cada uno de sus componentes hijos, es decir, utiliza el mecanismo de Delegación. Por otro lado, implementa normalmente las operaciones relacionadas con la manipulación de colecciones, como add(), remove(), etc.

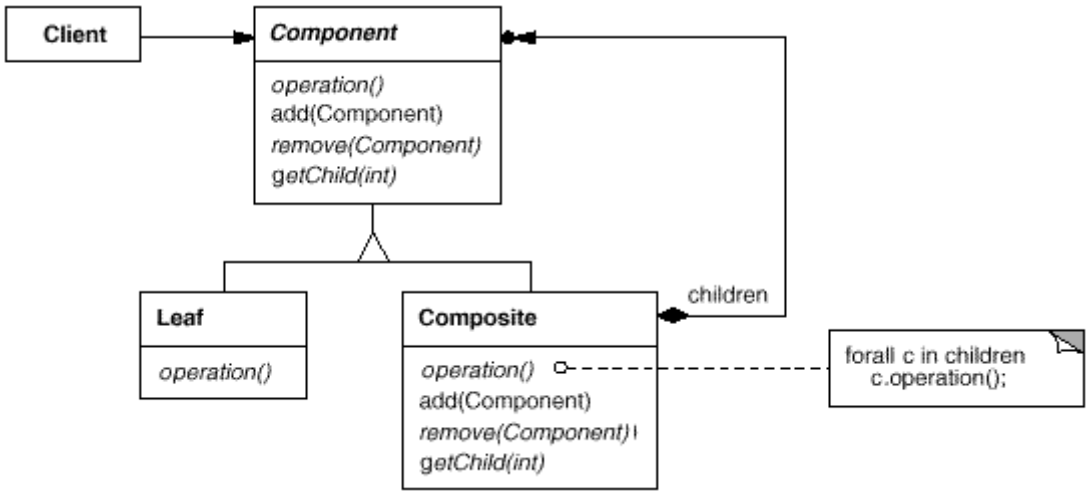
Podría parecer que los objetos de Picture se componen exclusivamente de objetos gráficos simples, pero esto no es así: dado que Picture se ajusta a la interfaz de Graphic, porque que ésta es su superclase, los objetos de Picture se pueden componer de otros objetos Picture de forma recursiva. Lo normal es que un objeto Picture se componga tanto de objetos simples de otros objetos compuestos.

El siguiente diagrama muestra una estructura típica para un objeto complejo, formado recursivamente por otros objetos gráficos:



Implementación

Diagrama de clases para el patrón Composite:



Descripción de los participantes:

Component (Graphic): Es la abstracción para la composición, tanto para los elementos simples (hojas o terminales) como para los compuestos. Por ejemplo, una clase Component llamada SistemaFicheros, que representara un sistema de ficheros, definiría operaciones como mover(), copiar(), renombrar(), getSize(), etc., tanto para la subclase Archivo (clase simple) como para la subclase Directorio (clase compuesta). Existen variantes sobre las operaciones que se pueden presentar, dependiendo de si se trata de una clase abstracta o de una interfaz. En caso de tratarse de una clase abstracta, puede proporcionar una implementación conveniente para las operaciones que son comunes para todas las subclases.

Hoja (Rectangle, Line, Text, etc): Son objetos que no tienen hijos. Definen el comportamiento de los objetos elementales de la composición. Siguiendo con el ejemplo del sistema de archivos, la clase Archivo sería una clase Leaf, e implementaría los métodos mover(), copiar(), renombrar() y getSize(), declarados en la interfaz de Component. Leaf puede que no implemente todos los métodos de Component, depende de si Component se trata de una clase abstracta o de una interfaz.

Composite (Picture): Son objetos que tienen hijos, los cuales almacena en una colección. Implementa los métodos definidos por la interfaz de Component, algunos de ellos se delegan en los componentes hijos (como el método draw() del ejemplo del editor gráfico) y otros se implementan directamente, como los métodos relativos a la gestión de la colección de hijos: add(), remove(), etc. Siguiendo con el ejemplo del sistema de archivos, la clase Directorio sería un caso de Composite: tendría una

colección de elementos hijos: algunos de tipo Archivo y otros de tipo Directorio. Las operaciones relativas a archivos las delegaría en Archivo e implementaría las relativas a la manipulación de la colección de archivos.

Client: El cliente manipula objetos de la jerarquía utilizando la interfaz de Component.

Aplicabilidad y Ejemplos

Usar el patrón Composite cuando:

- Se deben representar jerarquías de objetos todo-parte.
- Se necesita que el código cliente maneje uniformemente cualquier subclase de Component, sin tener que distinguir si se trata de una Leaf o de una Composite.

Pasemos a ver el código de algunos ejemplos.

Ejemplo 1 – Sistema de ficheros

Este ejemplo ilustra de manera sencilla el propósito del patrón Composite. Supongamos que nos solicitan lo siguiente: Crear un programa que permita crear un directorio raíz (C:\) con cuatro niveles de profundidad, tal y cómo se muestra en la siguiente figura:

```
+C: (nivel 0)
   leeme.txt(nivel 1)
   +videos (nivel 1)
      +entretenimiento (nivel 2)
         Cars.avi(nivel 3)
         Cars 2.avi(nivel 3)
      +formacion (nivel 2)
         +2011 (nivel 3)
            Curso Office.docx(nivel 4)
   +musica (nivel 1)
      +clasica (nivel 2)
         Berliotz.mp3(nivel 3)
```

En el primer nivel se creará un archivo de texto llamado ‘leeme.txt’, junto con dos directorios: ‘videos’ y ‘música’.

En el segundo nivel, para el directorio ‘videos’ se crearán dos subdirectorios: ‘entretenimiento’ y ‘formacion’. Para el directorio ‘musica’ se creará el subdirectorio ‘clasica’.

En el tercer nivel, para el directorio ‘entretenimiento’ se crearán dos archivos: ‘Cars.avi’ y ‘Cars 2.avi’; para el directorio ‘formacion’ se creará el subdirectorio ‘2011’; para el subdirectorio ‘clasica’ se creará el archivo ‘Berliotz.mp3’.

En el cuarto nivel sólo tenemos un recurso: el fichero ‘Curso Office.docx’, perteneciente al directorio ‘2011’.

Dado que tenemos una jerarquía todo/parte entre directorios y archivos, este programa puede realizarse de manera sencilla mediante el patrón Composite. Veamos el código.

Comenzamos con la abstracción Component. En este caso utilizaremos una clase abstracta que:

Component.java

- Proporcionará una implementación conveniente para todas las subclases. Por ejemplo, tanto archivos como directorios tienen un nombre, por lo que la clase Component almacenará este valor (en lugar de que lo haga cada clase). Otro ejemplo: ambas subclases necesitan una operación para mostrarse con un nivel de sangrado determinado, por lo que este código puede residir aquí y ser compartido.
- Declarará la operación mostrar() como abstracta, y cada subclase la implementará de una manera: Archivo se imprimirá en pantalla directamente, mientras que directorio recorrerá la colección de sus hijos para invocar la operación mostrar() de estos (delegará en ellos).
- Definirá una implementación trivial para las operaciones agregar() y eliminar (), que en la práctica tendrán que ser redefinidas en la clase Directorio, ya que es la única subclase donde tienen sentido estas operaciones.

```
package estructurales.composite.sistemaficheros;

/** Component **/
public abstract class Componente {

	// Nombre del directorio o del archivo
	protected String nombre;
```

```
	/*
	 * Las subclases proporcionaran un nombre para la clase base
	 */
	public Componente(String nombre) {
		this.nombre = nombre;
	}

	/*
	 * Operacion abstracta que cada subclase de implementar
	 */
	abstract public void mostrar(int profundidad);

	/*
	 * Implementación por defecto. Solo la soporta la clase
Directorio
	 */
	public void agregar(Componente c) {
		throw new RuntimeException("Operacion no soportada por el
componente.");
	}

	/*
	 * Implementación por defecto. Solo la soporta la clase
Directorio
	 */
	public void eliminar(Componente c) {
		throw new RuntimeException("Operacion no soportada por el
componente.");
	}

	/*
	 * Codigo conveniente para las subclases
	 */
	protected static final String ESPACIOS =
		"                                   ";
	protected String espacios(int nivel) {
		return ESPACIOS.substring(0, nivel*3);
	}
}
```

Veamos ahora las subclases.

Archivo.java

Se trata de una clase muy sencilla. Notad que para imprimir el nombre del archivo utilizar el método espacios() definido en la superclase Component. Este método necesita un argumento de tipo *int* para saber cómo debe ser la tabulación del texto.

```
package estructurales.composite.sistemaficheros;

/** Leaf **/
public class Archivo extends Componente {
```

```
	public Archivo(String nombre) {
		super(nombre);
	}

	@Override
	public void mostrar(int profundidad) {
		System.out.println(espacios(profundidad) + nombre);

		// DEBUG
		//System.out.println(espacios(profundidad) + nombre +
"(nivel " + profundidad + ")");
	}

}
```

Directorio.java

- Contiene una colección de componentes hijos.
- Implementa las operaciones definidas en Component para gestionar la colección.
- Delega en sus componentes la impresión por pantalla. Esto lo hace de manera recursiva, por lo que si un componente es del tipo Directorio, entonces recursivamente se imprimirán sus componentes.

```
package estructurales.composite.sistemaficheros;

import java.util.ArrayList;
import java.util.List;

/** Composite **/
public class Directorio extends Componente {

	// Contenedor para almacenar a los elementos hijos
	private List<Componente> componentes =
		new ArrayList<Componente>();

	public Directorio(String name) {
		super(name);
	}

	/*
	 * Imprime el nombre del directorio y recorre recursivamente
	 * sus hijos para que estos se muestren.
	 */
	@Override
	public void mostrar(int profundidad) {
		System.out.println(espacios(profundidad) + "+" + nombre);

		// DEBUG
		//System.out.println(espacios(profundidad) + "+" + nombre +
" (nivel " + profundidad + ")");
```

```
            for (Componente componente : componentes) {
            componente.mostrar(profundidad + 1);
            }
      }

      /* ****** Operaciones para manejar la coleccion de componentes
*/
      @Override
      public void agregar(Componente componente) {
            componentes.add(componente);
      }

      @Override
      public void eliminar(Componente componente) {
            componentes.remove(componente);
      }

}
```

Veamos la clase cliente.

MainClient.java

El siguiente código crea los componentes necesarios según la especificación del enunciado.

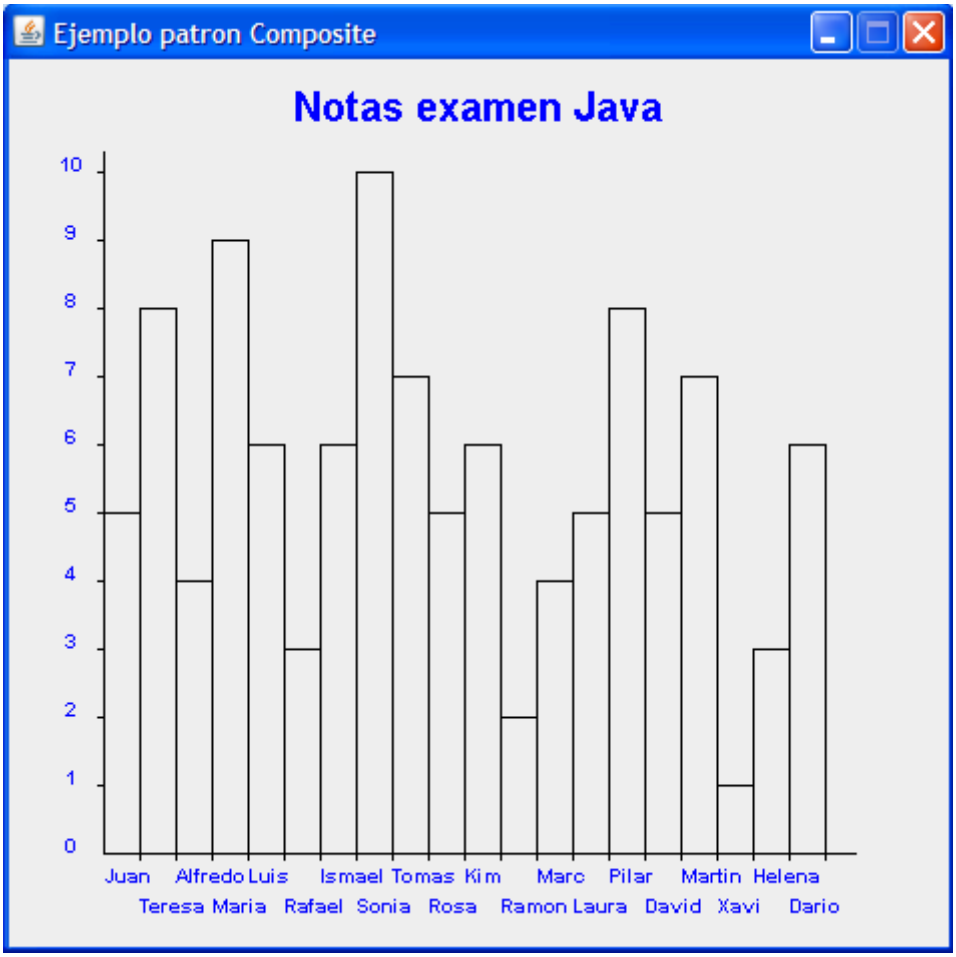
La última línea del programa muestra todo el árbol de componentes que se ha ido generando. Esto se consigue llamando al método mostrar() sobre el objeto compuesto 'raiz'. Lo importante aquí es percibir que no se hace ninguna distinción para el caso en que el componente es un Archivo o es un Directorio, es decir, hay un tratamiento uniforme, lo que simplifica el código cliente.

```
package estructurales.composite.sistemaficheros;

public class MainClient {

      public static void main(String[] args) {

            /* Raiz */
            Directorio raiz = new Directorio("C:");

            /* Nivel 1 ************************/

            // Primer hijo
            Componente arc1niv1 = new Archivo("leeme.txt");
            raiz.agregar(arc1niv1);

            // Segundo hijo
            Componente dir1niv1 = new Directorio("videos");
            raiz.agregar(dir1niv1);

            // Tercer hijo
            Componente dir2niv1 = new Directorio("musica");
```

```
            raiz.agregar(dir2niv1);

            /* Nivel 2 ***************************/

            // Primer hijo de 'videos'
            Componente dir1niv2 = new Directorio("entretenimiento");
            dir1niv1.agregar(dir1niv2);

            // Segundo hijo de 'videos'
            Componente dir2niv2 = new Directorio("formacion");
            dir1niv1.agregar(dir2niv2);

            // Unico hijo de 'musica'
            Componente dir3niv2 = new Directorio("clasica");
            dir2niv1.agregar(dir3niv2);

            /* Nivel 3 ***************************/

            // Primer hijo de 'entretenimiento'
            Componente arc1niv3 = new Archivo("Cars.avi");
            dir1niv2.agregar(arc1niv3);

            // Segundo hijo de 'entretenimiento'
            Componente arc2niv3 = new Archivo("Cars 2.avi");
            dir1niv2.agregar(arc2niv3);

            // Unico hijo de 'formacion'
            Componente dir1niv3 = new Directorio("2011");
            dir2niv2.agregar(dir1niv3);

            // Unico hijo de 'clasica'
            Componente dir2niv3 = new Archivo("Berliotz.mp3");
            dir3niv2.agregar(dir2niv3);

            /* Nivel 4 ***************************/

            // Unico hijo de '2011'
            Componente arc1niv4 = new Archivo("Curso Office.docx");
            dir1niv3.agregar(arc1niv4);

            // ejemplo
            // raiz.eliminar(3);

            /*
             * Aqui podemos ver que el código cliente no hace
distinciones
             * para tratar con la clase Archivo o con la clase
Directorio.
             * Esto simplifica enormemente la programacion del codigo
cliente.
             */
            raiz.mostrar(0);
      }
}
```

Ejemplo 2 – Aplicación para crear diagramas de barras

Supongamos que nos han pedido un programa que a partir de unos datos muestre un diagrama de barras como el de la siguiente figura:



Un diagrama de este tipo, consiste de los siguientes elementos:

Un titulo: Una breve descripción del diagrama, con letra grande, situado en la parte superior de la ventana y centrado horizontalmente.

Un eje horizontal: En este eje se muestran los distintos elementos. Por ejemplo, si se trata de representar las notas obtenidas en un examen, en este eje se mostrarían los nombres de los alumnos. El eje se muestra divido en segmentos, tantos como elementos haya. En el caso de los alumnos, a cada segmento le corresponde un nombre de alumno.

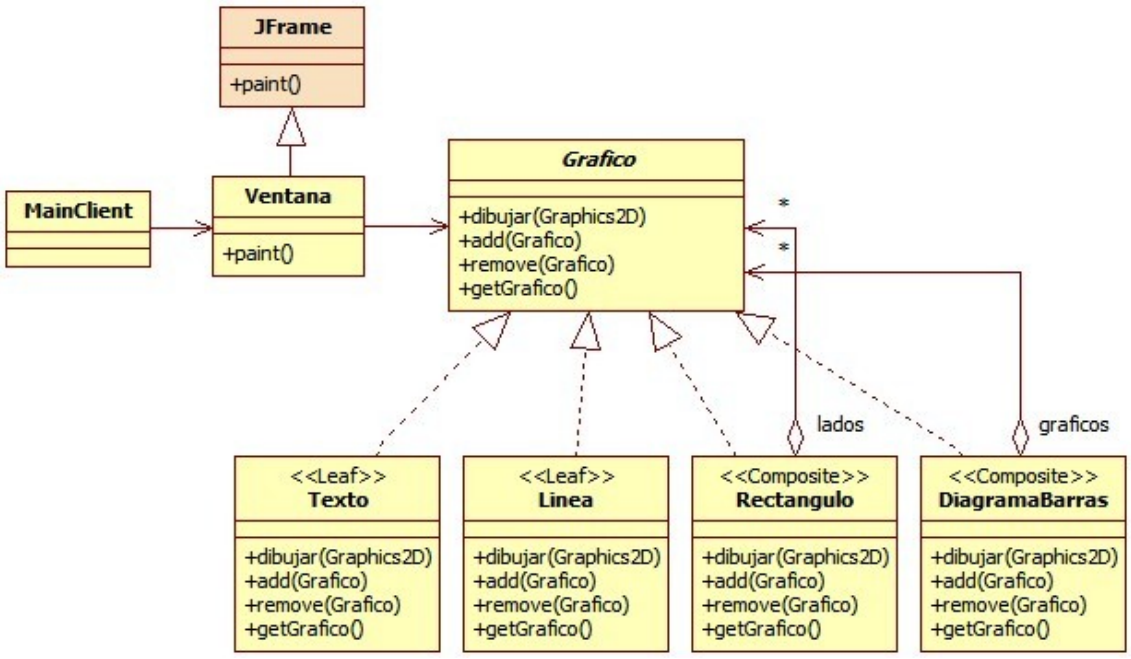
Un eje vertical: Siguiendo con el ejemplo de las notas, en este eje se representan los valores posibles (0-10). El eje se divide en segmentos según estos valores, por lo que se tendrían 10 segmentos.

Un conjunto de barras: Se trata de una colección de rectángulos, donde cada rectángulo representa la nota obtenida por un alumno. Por tanto, se tendrán tantos rectángulos como alumnos.

Podemos decir lo siguiente sobre estos elementos que se acaban de comentar:

- Un titulo es un texto, por tanto, es un componente elemental.
- Tanto los ejes como los segmentos son líneas. Una línea es un componente elemental.
- Una barra o rectángulo es una composición específica de cuatro líneas, por tanto, es un componente compuesto.
- El diagrama de barras resultante (el todo), es una composición específica de texto, líneas y rectángulos, luego es un componente compuesto.

Dado que tenemos una jerarquía todo/parte, este programa puede diseñarse aplicando el patrón Composite. El diagrama de clases queda como sigue:



Veamos el funcionamiento general:

MainClient

Una clase cliente contiene los datos que se deben mostrar en el diagrama de barras. Estos datos se pueden obtener de una fuente externa o estar incluidos directamente en la propia clase. La clase cliente crea un objeto Ventana y le pasa los datos.

Ventana

La clase Ventana extiende a javax.swing.JFrame, es otra clase cliente y es la responsable de conducir todo el proceso de creación del diagrama de barras y de mostrarlo. Para ello, se encargará de crear todos los elementos gráficos necesarios: el título del diagrama, los ejes, los segmentos de los ejes y las barras.

Básicamente, lo que hace es crear un objeto (compuesto, Composite) DiagramaBarras como contenedor general para todo el resto de componentes. A continuación, crea el título (objeto simple, Leaf) y lo añade al objeto DiagramaBarras. Después analiza los datos recibidos de la clase cliente y procede a crear los ejes (también se trata de objetos simples, Leaf) y a añadirlos al objeto DiagramaBarras. Finalmente, dibuja las barras (que también se trata de objetos compuestos, Composite), y las añade, como no, al objeto DiagramaBarras.

Lo más importante, es apreciar que gracias al uso del patrón Composite, el código de Ventana trata de manera uniforme a todos los objetos derivados de Component, lo que la convierte en una clase simple, fácil de entender y de modificar.

Grafico

Es una interfaz que define la abstracción que cualquier subclase debe implementar. No obstante, las subclases implementan las operaciones de Grafico con una intención muy diferente, dependiendo si son subclases simples o compuestas.

Por ejemplo, una subclase simple (Leaf) tendrá una implementación dummy para un método relativo al manejo de una colección de componentes (normalmente lanzará una excepción), ya que por definición una clase Leaf no puede contener elementos.

Otro ejemplo: una clase compuesta (Component) no implementará propiamente el código para dibujarse en pantalla (ya que por definición no tiene sentido pintar un compuesto), sino que utilizará el mecanismo de Delegación para invocar al método dibujar() de cada uno de los elementos de la colección que incluye.

Como se ha explicado extensamente la clase Grafico, el comentario para sus subclases podrá realizarse brevemente:

Texto

Clase simple (Leaf) que dibuja un texto en la pantalla. Se utiliza para mostrar el titulo del grafico y los valores para los ejes X e Y del mismo.

Linea

Clase simple (Leaf) que dibuja una línea en las coordenadas especificadas.

Rectangulo

Es una clase compuesta (Composite) que contiene una lista formada por cuatro objetos de tipo Linea

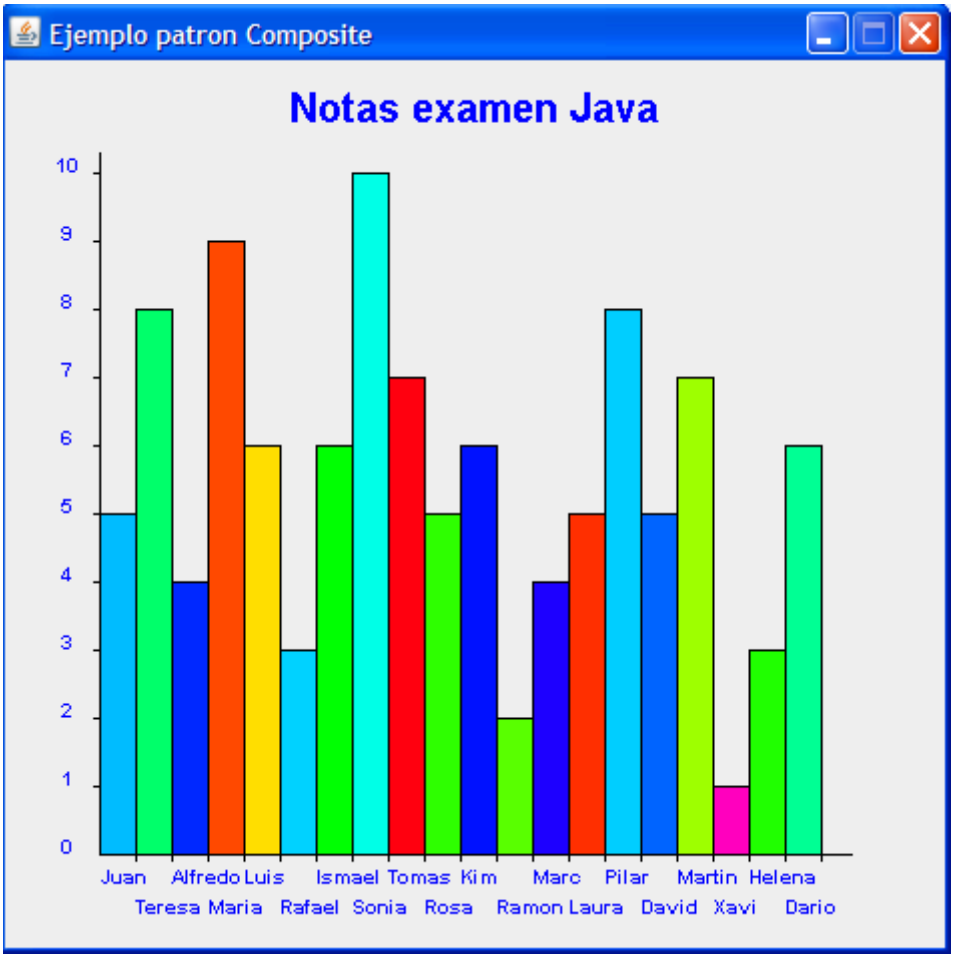
DiagramaBarras

Es una clase compuesta (Composite) que contiene una lista formada por un número arbitrario de rectángulos, líneas y texto.

Antes de ver el código fuente de las clases, supongamos que nos piden lo siguiente:

- El diagrama de barras presentado anteriormente es hueco y sin colores, lo que permite que se genere en un tiempo muy pequeño. No obstante, nos han pedido si sería posible obtener un diagrama de barras adicional: sólido, con cada barra de un color diferente.
- Nos dicen que sería muy cómodo que el usuario pudiera escribir en un fichero de propiedades el nombre del tipo de diagrama de barras que quiere utilizar. La aplicación podría leer este fichero en tiempo de ejecución y crear uno u otro diagrama.

La figura siguiente es un ejemplo del nuevo diagrama de barras que tendremos que generar:



Lo anterior implica ciertos cambios en la aplicación:

- Por un lado, necesitamos una clase que especialice a Rectangulo para mostrar los rectángulos sólidos y de colores, que llamaremos RectanguloSolido.
- Por otro lado, necesitamos una clase FactoriaRectangulo que pueda crear instancias de cualquiera de las clases que construyen rectángulos.

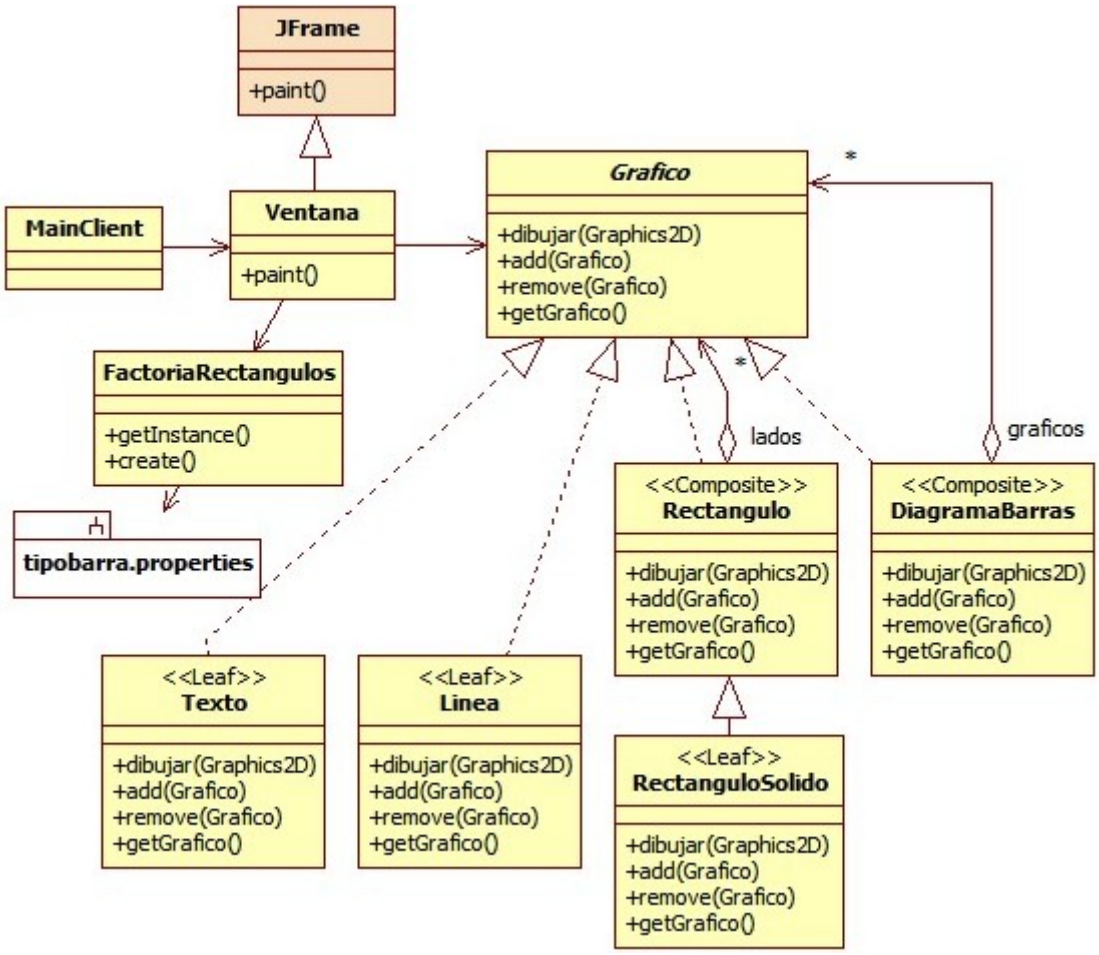
RectanguloSolido

Esta clase, igual que rectángulo, también delega en las líneas que contiene para dibujar el rectángulo. Sin embargo, para pintarlo no delega en nadie, sino que se encarga ella misma. Desde este punto de vista se trata de una clase Leaf, a pesar de que extiende a una clase Composite.

FactoriaRectangulos

Factoria implementada como Singleton que en función del valor de una propiedad del archivo de propiedades 'tipobarra.properties', que debe encontrarse en el mismo paquete que la clase, instancia dinámicamente el tipo de rectángulo adecuado. Esto permite que el código cliente no tenga que variar para mostrar uno u otro tipo de barras.

El nuevo diagrama de clases es el siguiente:



Ahora sí, pasemos a ver el código fuente. Comenzamos por las clases cliente:

MainClient.java

```
package estructurales.composite.diagramabarras.client;

import estructurales.composite.diagramabarras.Ventana;

public class MainClient {

	/*
	 * Arrays con los datos a mostrar en el diagrama.
	 * En una version mas sofisticada se tendrían que
```

```
	 * obtener de una fuente externa (fichero, base
	 * de datos, etc)
	 */
	private static String[] elementosX = new String[] {
		"Juan", "Teresa", "Alfredo", "Maria", "Luis",
		"Rafael", "Ismael", "Sonia", "Tomas", "Rosa",
		"Kim", "Ramon", "Marc", "Laura", "Pilar",
		"David", "Martin", "Xavi", "Helena", "Dario"
	};

	private static float[] elementosY = new float[] {
		5.5f, 8.0f, 4.0f, 9.0f, 6.25f, 3.5f, 6.0f,
		10.0f, 7.5f, 5.0f, 6.0f, 2.5f, 4.0f, 5.0f,
		8.0f, 5.5f, 7.0f, 1.0f, 3.5f, 6.5f
	};

	private static String[] elementosDeRefenciaY = new String[] {
		"  0", "  1", "  2", "  3", "  4", "  5", "  6", "  7", "
8", "  9", " 10"
	};

	public static void main(String ...args) {
		new Ventana(elementosX, elementosY, elementosDeRefenciaY);
	}
}
```

Ventana.java

```
package estructurales.composite.diagramabarras;

import java.awt.Graphics;
import java.awt.Graphics2D;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

/*
 * Clase que representa la ventana donde se pinta
 * el diagrama de barras. Tambien se encarga de crear todos
 * los elementos graficos necesarios. Conduce todo el
 * proceso de creacion del diagrama.
 *
 * Permite crear diagramas con barras (rectangulos) huecas
 * o solidas. Para ello utiliza una factoria que lee la clase
 * de rectangulo a utilizar desde un fichero de propiedades.
 */
public class Ventana extends JFrame {

	/********** DATOS *********************************/

	private static final long serialVersionUID = 1L;

	public static int FRAME_WIDTH = 475, FRAME_HEIGHT = 475;

	/*
	 * Arrays con los datos a mostrar en el diagrama.
	 * Los debe proporcionar una clase cliente
```

```
 */
private String[] elementosX;
private float[] elementosY;
private String[] elementosDeRefenciaY;

/*
 * Objecto compuesto que contendrá todos los elementos
 * graficos que conforman el diagrama de barras.
 *
 * Desde el punto de vista de esta clase cliente, la
 * construccion del diagrama de barras consiste en
 * ir llamando al metodo add() del objeto 'diagrama'
 * cada vez que quiere agregar un elemento grafico
 * que tenga que mostrarse en el diagrama de barras.
 */
private DiagramaBarras diagrama = new DiagramaBarras();

/*
 * Coordenadas para los ejes X e Y. Los ocho
 * puntos se pueden representar con cuatro
 * valores, ya que comparten varios de ellos.
 * y
 * |
 * |___x
 */
private static int COORD_X1_EJE_X = 50, COORD_Y_EJE_X = 425,
      COORD_X2_EJE_X = 425, COORD_Y2_EJE_Y = 75;

/*
 * Lineas que representan los ejes X e Y del diagrama.
 */
private static Linea ejeX =
      new Linea(COORD_X1_EJE_X, COORD_Y_EJE_X,
                  COORD_X2_EJE_X, COORD_Y_EJE_X);

private static Linea ejeY =
      new Linea(COORD_X1_EJE_X, COORD_Y_EJE_X,
                  COORD_X1_EJE_X, COORD_Y2_EJE_Y);

// Longitud de cada eje (X e Y miden lo mismo)
private static int len_eje;

// Numero de segmentos para el ejeX
private int total_segmentos_horizontales;

// Numero de segmentos para el ejeY
private int total_segmentos_verticales;

// Ancho de cada segmento del ejeX
private int ancho_segmento_horizontal;

// Ancho de cada segmento del ejeY
private int ancho_segmento_vertical;

/********** METODOS *********************************/

/*
 * Constructor
 */
```

```
	public Ventana(String[] elementosX_,
			float[] elementosY_, String[] elementosDeRefenciaY_ )
	{
		super();

		// Volcado de los datos del diagrama procedentes de la
clase cliente
		elementosX = elementosX_;
		elementosY = elementosY_;
		elementosDeRefenciaY = elementosDeRefenciaY_;

		SwingUtilities.invokeLater(new Runnable() {
			public void run() {
				iniciarPrograma();
			}
		});
	}

	/*
	 * Metodo invocado por la MVJ para pintar graficos.
	 * Al arrancar la aplicacion la MVJ llama una vez
	 * a este metodo. Nosotros somos responsables de
	 * llamarlo mediante repaint() cuando necesitemos
	 * reflejar algún cambio en el escenario.
	 */
	@Override
	public void paint(Graphics g) {
		super.paint(g);
		Graphics2D g2 = (Graphics2D) g;

		/*
		 * A tener en cuenta que a pesar de que
		 * tenemos figuras simples y compuestas,
		 * este metodos las trata a todas por igual,
		 * usando la interfaz Grafico
		 */
		diagrama.dibujar(g2);
	}

	/*
	 * Ejecucion de las rutinas principales
	 */
	private void iniciarPrograma() {
		inicializaciones();
		initGUI();
		crearTitulo("Notas examen Java");
		crearEjes();
		crearBarras();

		// Esta llamada produce que la MVJ llame a paint(),
		// lo que produce que se recorran todos los hijos
		// del objeto compuesto 'diagrama'
		repaint();
	}

	private void inicializaciones() {
		/*
		 * Añadimos los ejes X e Y al diagrama de barras.
		 */
```

```
        diagrama.add(ejeX);
        diagrama.add(ejeY);

        /*
         * Longitud de cada eje (X e Y miden lo mismo)
         */
        len_eje = (int) (ejeX.getLinea().x2 - ejeX.getLinea().x1);

        /*
         * Numero de segmentos para el ejeX
         */
        total_segmentos_horizontales = elementosX.length;

        /*
         * Numero de segmentos para el ejeY
         */
        total_segmentos_verticales =
              elementosDeRefenciaY.length;

        /*
         * Ancho de cada segmento del ejeX
         */
        ancho_segmento_horizontal =
              len_eje / total_segmentos_horizontales;

        /*
         * Ancho de cada segmento del ejeY
         */
        ancho_segmento_vertical =
              len_eje / total_segmentos_verticales;

  }

  /*
   * Establecer la interfaz de usuario
   */
  private void initGUI() {
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
        setResizable(false); // No dejamos redimensionar
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null); // Centrar
        setTitle("Ejemplo patron Composite");
        setVisible(true);
  }

  /*
   * Crear el titulo y añadirlo al objeto 'diagrama'.
   * El titulo aparecera con letra grande,
   * centrado horizontalmente, en la parte superior.
   */
  private void crearTitulo(String titulo) {
        boolean esUnTitulo = true;
        diagrama.add(new Texto(titulo, esUnTitulo));
  }

  /*
   * Crear los segmentos de los ejes X e Y y los
   * valores textuales asociados a cada uno.
   */
```

```
	private void crearEjes() {
		crearSegmentosEjeX();
		crearValoresEjeX();
		crearSegmentosEjeY();
		crearValoresEjeY();
	}

	/*
	 * Crear los segmentos del eje X y añadirlos al objeto
'diagrama'
	 */
	private void crearSegmentosEjeX() {
		int posX_base = (int)ejeX.getLinea().x1;
		int ancho_acumulado = posX_base +
ancho_segmento_horizontal;

		for (int i=0; i<total_segmentos_horizontales; i++) {

			diagrama.add(
					new Linea(ancho_acumulado,
(int)ejeX.getLinea().y1,
							ancho_acumulado,
(int)ejeX.getLinea().y1 + 3)
			);
			ancho_acumulado += ancho_segmento_horizontal;
		}
	}

	/*
	 * Crear los valores del eje X y añadirlos al objeto 'diagrama'
	 */
	private void crearValoresEjeX() {
		int posX_base = (int)ejeX.getLinea().x1;
		int ancho_acumulado = posX_base +
ancho_segmento_horizontal;

		for (int i=0; i<elementosX.length; i++) {
			/*
			 * Para evitar que se solapen horizontalmente los
textos,
			 * alternamos su coordenada vertical.
			 */
			int posYtexto = (int)ejeX.getLinea().y1;
			posYtexto = (i % 2 == 0) ? posYtexto + 15 : posYtexto
+ 30;

			diagrama.add(
					new Texto(

	String.valueOf(elementosX[i]),
							ancho_acumulado -
ancho_segmento_horizontal,
							posYtexto
							)
					);
			ancho_acumulado += ancho_segmento_horizontal;
		}
	}
```

```
	/*
	 * Crear los segmentos del eje Y y añadirlos al objeto
'diagrama'
	 */
	private void crearSegmentosEjeY() {
		int posY_base = (int)ejeY.getLinea().y1;
		int alto_acumulado = posY_base - ancho_segmento_vertical;

		for (int i=0; i<total_segmentos_verticales - 1; i++) {

			diagrama.add(
					new Linea((int)ejeY.getLinea().x1,
alto_acumulado,
							(int)ejeY.getLinea().x1-3,
alto_acumulado)
			);
			alto_acumulado -= ancho_segmento_vertical;
		}
	}

	/*
	 * Crear los valores del eje Y y añadirlos al objeto 'diagrama'
	 */
	private void crearValoresEjeY() {
		int posY_base = (int)ejeY.getLinea().y1;
		int alto_acumulado = posY_base - ancho_segmento_vertical;

		for (int i=0; i<total_segmentos_verticales; i++) {
			// Mostrar los valores de referencia en cada segmento
del eje Y
			diagrama.add(
					new Texto(

	String.valueOf(elementosDeRefenciaY[i]),
							(int)ejeY.getLinea().x1-25,
// el -25 es para evitar que quede pegado al eje
							alto_acumulado +
ancho_segmento_vertical
							)
					);
			alto_acumulado -= ancho_segmento_vertical;
		}
	}

	/*
	 * Crear los rectangulos que representan las barras
	 * y añadirlos al objeto 'diagrama'.
	 */
	private void crearBarras() {
		int x1 = (int)ejeX.getLinea().x1;
		int y1 = (int)ejeY.getLinea().y1;

		// Se crean tantas barras como elementos haya para el eje X
		for (int i=0; i<elementosX.length; i++) {

			// Calculo de la Y que le pertenece al elemento X en
curso
			int y2 = y1 - ((int) elementosY[i] *
ancho_segmento_vertical);
```

```
                    /*
                     * Uso de la factoria para la obtencion de la clase
concreta
                     * a utilizar para represtar las barras. Tenemos de
dos tipos:
                     * RectanguloHueco y RectanguloRelleno, ambas tienen
el mismo
                     * constructor, por lo que los parametros son los
mismos.
                     *
                     * El primer parametro, tipobarra, es el nombre de la
propiedad
                     * que la factoria necesita conocer para leer del
fichero de
                     * propiedades donde se especifica el nombre de la
clase a instanciar.
                     */
                    Grafico barra =
FactoriaRectangulos.getInstance().create(
                                    "tipobarra",
                                    new Linea(x1, y1, x1, y2),
                                    new Linea(x1, y2, x1 +
ancho_segmento_horizontal, y2),
                                    new Linea(x1 + ancho_segmento_horizontal,
y2, x1 + ancho_segmento_horizontal, y1),
                                    new Linea(x1 + ancho_segmento_horizontal,
y1, x1, y1));
                    diagrama.add(barra);
                    x1 += ancho_segmento_horizontal;
                }
            }
}
```

Ahora la clase de factoría:

FactoriaRectangulos.java

Es importante observar cómo es capaz de crear un objeto de Rectangulo o RectanguloSolido invocando a su constructor. Para ello se utiliza la clase Constructor, en lugar de Class.newInstance(), que sólo permite invocar al constructor por defecto.

```
package estructurales.composite.diagramabarras;

import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.Constructor;

/*
 * Factoria implementada como Singleton.
 * En funcion del valor de una propiedad del archivo de propiedades
 * 'tipobarra.properties', que debe encontrarse en el mismo paquete
 * que la clase, instancia dinamicamente el tipo de Rectangulo
adecuado.
```

```
 * Esto permite que el código cliente no tenga que variar para mostrar
 * uno u otro tipo de barras.
 */
public class FactoriaRectangulos {

	private static FactoriaRectangulos factory = new
FactoriaRectangulos();

	// true indica que ya se han cargado los nombres de las clases
	// desde el fichero de propiedades
	private static boolean propiedadesCargadas = false;

	// Propiedades
	private static java.util.Properties prop = new
java.util.Properties();

	public static FactoriaRectangulos getInstance() {
		return factory;
	}

	public Grafico create(String nombrePropiedad, Linea linea1,
Linea linea2, Linea linea3, Linea linea4) {
		try {
			// La clase se obtiene leyendo del archivo properties
			Class<?> clase =
Class.forName(getClase(nombrePropiedad));
			Constructor<?> ctor = clase.getDeclaredConstructors()
[0];

			// Crear una instancia
			return (Grafico) ctor.newInstance(linea1, linea2,
linea3, linea4);
		} catch (ClassNotFoundException e) { // No existe la clase
			e.printStackTrace();
			return null;
		} catch (Exception e) { // No se puede instanciar la clase
			e.printStackTrace();
			return null;
		}
	}

	// Lee un archivo properties donde se indican las clases
instanciables
	private static String getClase(String nombrePropiedad) {
		try {
			// Carga de propiedades desde archivo -solo la
primera vez-
			if (!propiedadesCargadas) {
				InputStream is = FactoriaRectangulos.class
						.getResourceAsStream("tipobarra.pro
perties");

				prop.load(is);
				propiedadesCargadas = true;
			}

			// Lectura de propiedad
			String nombreClase =
prop.getProperty(nombrePropiedad, "");
```

```
                if (nombreClase.length() > 0)
                      return nombreClase;

          } catch (IOException e) {
                e.printStackTrace();
          }
          return null;
    }
}
```

Ahora el archivo de propiedades. Debe estar en el mismo lugar que el paquete que la factoría:

tipobarra.properties

```
tipobarra=estructurales.composite.diagramabarras.RectanguloSolido
```

Pasemos a ver la jerarquía de Component:

Grafico.java

```
package estructurales.composite.diagramabarras;

import java.awt.Graphics2D;

/** Interfaz para el "Component" **/
public interface Grafico {

      /*
       * Dibujar una figura en la pantalla.
       */
      public void dibujar(Graphics2D g2);

      /*
       * Este metodo se implementara de diferente manera
       * en una clases simple que en una clases compuesta.
       *
       * En una clase simple consiste en devolver el grafico
       * que representa la propia clase.
       *
       * Mientras que en una clase compuesta consiste en
       * devolver una lista con los graficos que la forman.
       * Por ejemplo, un rectangulo se compone de
       * cuatro objetos de tipo linea.
       */
      public Grafico[] getGrafico();

      /*
       * Este metodo no se implementara en las clases simples,
       * ya que no tienen hijos que añadir
       */
      public void add(Grafico grafico);
```

```
	/*
	 * Este metodo no se implementara en las clases simples,
	 * ya que no tienen hijos que eliminar
	 */
	public void remove(Grafico grafico);

}
```

Texto.java

```
package estructurales.composite.diagramabarras;

import java.awt.Color;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics2D;
import java.awt.geom.Rectangle2D;

/** "Leaf" **/

/*
 * Clase simple (terminal) que dibuja un
 * texto en la pantalla.
 *
 * Se utiliza para mostrar el titulo del grafico
 * y los valores para los ejes X e Y del mismo.
 */
public class Texto implements Grafico {

	private String texto;
	private boolean esUnTitulo;
	private int x, y;

	private Font font;

	/*
	 * Constructor para mostrar texto en
	 * una posicion determinada.
	 */
	public Texto(String texto, int x, int y) {
		this.texto = texto;
		this.x = x;
		this.y = y;
	}

	/*
	 * Constructor para cuando se trata de un titulo.
	 * Un titulo debe mostrarse con letra grande, en la
	 * parte superior del marco y centrado horizontalmente.
	 */
	public Texto(String texto, boolean esUntitulo) {
		this.texto = texto;
		this.esUnTitulo = esUntitulo;
	}

	/*
```

```
	 * Dibujar el texto, diferenciando si se trata
	 * o no de un titulo.
	 */
	@Override
	public void dibujar(Graphics2D g2) {
		if (esUnTitulo) {
			// Letra grande
			font = new Font("Arial", Font.BOLD, 20);

			// Obtener el tamaño del texto para la fuente actual
en el contexto grafico g2
			FontMetrics fm   = g2.getFontMetrics(font);
			Rectangle2D rect = fm.getStringBounds(texto, g2);

			// Centrar el texto horizontamente
			int anchoTexto = (int)(rect.getWidth());

			this.x = (Ventana.FRAME_WIDTH  - anchoTexto)  / 2;

			/*//Centrar el texto verticalmente
			int altoTexto = (int)(rect.getHeight());
			this.y = (MainClient.FRAME_HEIGHT - altoTexto) / 2  +
fm.getAscent();*/
			this.y = 60;

		} else { // Si no es un titulo, entonces la letra pequeña
			font = new Font("Arial", Font.BOLD, 10);
		}

		g2.setColor(Color.BLUE);
		g2.setFont(font);
		g2.drawString(texto, x, y); // Dibujar el texto
	}

	/*
	 * La lista de graficos de un Texto tan solo
	 * contiene un grafico: el mismo
	 */
	@Override
	public Grafico[] getGrafico() {
		return new Grafico[] { this };
	}

	/*
	 * Metodo no soportado para una clase simple
	 */
	@Override
	public void add(Grafico figura) {
		throw new UnsupportedOperationException("Motodo no
implementado en un objeto terminal");
	}

	/*
	 * Metodo no soportado para una clase simple
	 */
	@Override
	public void remove(Grafico figura) {
		throw new UnsupportedOperationException("Motodo no
implementado en un objeto terminal");
```

```
	}

}
```

Linea.java

```
package estructurales.composite.diagramabarras;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.Line2D;

/** "Leaf" **/

/*
 * Clase simple (terminal) que dibuja una linea
 * en las coordenadas especificadas.
 */
public class Linea implements Grafico {

	// Objecto subyacente que realmente se pinta
	private Line2D.Double linea;

	public Line2D.Double getLinea() {
		return linea;
	}

	/*
	 * Crear una linea segun las coordenadas
	 * recibidas en los parametros
	 */
	public Linea(int x1, int y1, int x2, int y2) {
		this.linea = new Line2D.Double(x1, y1, x2, y2);
	}

	/*
	 * Dibujar la linea
	 */
	public void dibujar(Graphics2D g2) {
		g2.setColor(Color.BLACK);
		g2.draw(linea);
	}

	/*
	 * La lista de graficos de una linea tan solo
	 * contiene un grafico: ella misma
	 */
	@Override
	public Grafico[] getGrafico() {
		return new Grafico[] { this };
	}

	/*
	 * Metodo no soportado para una clase simple
	 */
	@Override
	public void add(Grafico grafico) {
```

```
            throw new UnsupportedOperationException("Motodo no
implementado en un objeto terminal");
      }

      /*
       * Metodo no soportado para una clase simple
       */
      @Override
      public void remove(Grafico grafico) {
            throw new UnsupportedOperationException("Motodo no
implementado en un objeto terminal");
      }

}
```

Rectangulo.java

```
package estructurales.composite.diagramabarras;

import java.awt.Graphics2D;

/** "Composite" **/

/*
 * Rectangulo es una clase compuesta por
 * 4 objetos de tipo Linea
 */
public class Rectangulo implements Grafico {

      // Lista de lineas que forman el rectangulo
      private Grafico[] lados = new Grafico[4];

      public Rectangulo(Linea linea1, Linea linea2,
                  Linea linea3, Linea linea4)
      {
            lados[0] = linea1;
            lados[1] = linea2;
            lados[2] = linea3;
            lados[3] = linea4;
      }

      /*
       * En el caso de un grafico compuesto no
       * podemos dibujar directamente, sino que
       * hay que delegar en los graficos simples
       * que lo componen. En el caso del rectangulo
       * se delega en las lineas, que son sus lados.
       */
      @Override
      public void dibujar(Graphics2D g2) {
            for (Grafico lado : lados) {
                  // Delegar en los objetos hijos
                  lado.dibujar(g2);
            }
      }

      /*
```

```
	 * La lista de graficos de un rectangulo
	 * son las cuatro lineas que lo forman.
	 */
	@Override
	public Grafico[] getGrafico() {
		return lados;
	}

	/*
	 * A pesar de que un rectangulo es un objeto compuesto,
	 * no tiene sentido añadir nuevos hijos (nuevos lados),
	 * ya que el rectangulo se construye correctamente mediante
	 * su constructor y despues no se puede modificar.
	 * En otro tipo de objetos compuestos sí tiene sentido
	 * agregar objetos arbitrariamente, una vez creados.
	 */
	@Override
	public void add(Grafico grafico) {
		throw new UnsupportedOperationException("Motodo no
implementado en un Rectangulo");
	}

	/*
	 * Idem metodo add() pero para el caso de la eliminacion
	 * de objetos hijo.
	 */
	@Override
	public void remove(Grafico grafico) {
		throw new UnsupportedOperationException("Motodo no
implementado en un Rectangulo");
	}

}
```

RectanguloSolido.java

```
package estructurales.composite.diagramabarras;

import java.awt.Color;
import java.awt.Graphics2D;
import java.util.Random;

/** Es un mix entre un "Leaf" y un "Composite **/
public class RectanguloSolido extends Rectangulo {

	private int x, y, ancho, alto;
	private static Color colorPrevio;

	public RectanguloSolido(Linea linea1, Linea linea2,
			Linea linea3, Linea linea4)
	{
		super(linea1, linea2, linea3, linea4);

		/*
		 * Obtener los datos necesarios para pintar un rectangulo.
		 */
		x = (int) linea1.getLinea().x1;
```

```
			y = (int) linea1.getLinea().y2;
			ancho = (int) (linea2.getLinea().x2-x);
			alto =  (int)linea1.getLinea().y1-y;
		}

		@Override
		public void dibujar(Graphics2D g2) {
			Color color = getColorAleatorio();
			// Evitar que dos barras contiguas tengan el mismo color
			if (colorPrevio != null) {
				do {
					color = getColorAleatorio();
				} while ( color == colorPrevio);

			}

			g2.setColor(color);
			g2.fillRect(x, y, ancho, alto); // Pintar barra
			super.dibujar(g2); // Pintar los bordes
			colorPrevio = color;
		}

		// Devuelve un color al azar
		private Color getColorAleatorio() {
			return Color.getHSBColor(new Random().nextFloat(), 1.0F,
1.0F );
		}
}
```

DiagramaBarras.java

```
package estructurales.composite.diagramabarras;

import java.awt.Graphics2D;
import java.util.ArrayList;
import java.util.List;

/** "Composite" **/

/*
 * DiagramaBarras es una clase compuesta por
 * rectangulos, lineas y texto que se debe
 * mostrar graficamente.
 */
public class DiagramaBarras implements Grafico {

		/*
		 * Lista de graficos
		 */
		List<Grafico> graficos = new ArrayList<Grafico>();

		/*
		 * En el caso de una grafico compuesto no
		 * podemos dibujar directamente, sino que
		 * hay que delegar en los graficos simples
		 * que lo forman.
		 *
```

```
	 * Hay que tener en cuenta, que el metodo
	 * actua recursivamente hasta encontrar
	 * una clase simple que se pueda dibujar.
	 */
	@Override
	public void dibujar(Graphics2D g2) {
		for(Grafico grafico: graficos){
			// Delegar en los objetos hijos
			grafico.dibujar(g2);
		}
	}

	/*
	 * La lista de graficos de un diagrama de barras
	 * se puede componer de cualquier numero de
	 * rectangulos, lineas y texto grafico.
	 */
	@Override
	public Grafico[] getGrafico() {
		return graficos.toArray(new Grafico[] {});
	}

	/*
	 * Agregar un nuevo objeto a la lista de hijos
	 */
	@Override
	public void add(Grafico figura) {
		graficos.add(figura);
	}

	/*
	 * Eliminar de la lista de hijos el objeto especificado
	 */
	@Override
	public void remove(Grafico figura) {
		graficos.remove(figura);
	}

}
```

Problemas específicos e implementación

El patrón Composite:

- Define jerarquías consistentes de objetos elementales y de objetos compuestos. Los objetos elementales se pueden combinar para dar lugar a objetos más complejos, que a su vez se pueden combinar para dar lugar a objetos todavía más complejos y así recursivamente.
- Hace que el código cliente sea simple, manejando de la misma manera a los objetos básicos como a las estructuras complejas. Las clases clientes

normalmente desconocen si están tratando con una clase Leaf o con una Composite, lo cual claramente simplifica la programación.

- Facilita la adición de nuevas clases de componentes. Una nueva clase Leaf o Composite funcionará perfectamente con los componentes y clases cliente existente.

- Produce un diseño muy abierto, lo que puede ser un inconveniente. La facilidad con la que se pueden añadir nuevos componentes hace que sea difícil restringir el tipo de componentes que forman parte de una composición determinada.

Implementación

Hay ciertos temas a considerar cuando se necesita implementar el patrón Composite:

Referencias explícitas al padre

Hacer que una objeto hijo tenga una referencia al objeto que lo contiene puede simplificar las operación de recorrido de la estructura compuesta. La referencia hacia el objeto padre hace que sea sencillo ascender de nivel en la estructura y, por ejemplo, eliminar un componente. Las referencias hacia los objetos padre está muy relacionado con otro patrón de diseño que aún tenemos que ver: Chain of Resposibility (Cadena de Responsabilidad).

El lugar natural para definir la referencia al objeto padre es la clase Component, por supuesto, cuando se trata de una clase abstracta y no de una interfaz. De esta manera, las clases Leaf y Composite pueden heredar la referencia y las operaciones que permiten gestionarla.

Para que el mecanismo de la referencia al objeto padre sea consistente, es esencial asegurar que todos los objetos hijos de un Composite tengan a tal Composite como padre y, que a su vez, el Composite los tenga a ellos como hijos. La manera más fácil de garantizar esto es establecer/desestablecer el valor del padre de un componente sólo cuando éste está siendo añadido a un compuesto o eliminado de un compuesto. Si Component es una clase abstracta, esto se puede implementar (sólo una vez) en las operaciones de agregar y quitar de Component, de manera que se hereda en todas las subclases, quedando así garantizada de forma automática la doble navegabilidad.

Compartir componentes

A veces es útil compartir ciertos componentes, por ejemplo, para reducir los requerimientos de almacenamiento de la aplicación. No obstante, para que sea posible compartir un componente es necesario que tal objeto pueda tener más de un padre, ya que en caso contrario se complica el mecanismo de compartición.

Una posible solución es hacer que sean los objetos hijos quienes almacenen referencias a sus (múltiples) padres. Sin embargo, este diseño puede dar lugar a ambigüedades en la manera como una solicitud se propaga ascendentemente por la estructura. Cómo ya veremos, el patrón Flyweight (Peso Mosca) muestra cómo se puede modificar un diseño para evitar tener que almacenar las referencias a los padres. Funciona en los casos donde los objetos hijos externalizan parte o la totalidad de su estado, evitando así el envío de peticiones a los objetos padre.

Maximizar la interfaz de la clase Component

Uno de los objetivos del patrón Composite es hacer que el código cliente no sea consciente si el componente que está tratando es en realidad un Leaf o un Composite. Para conseguir esto, la clase Component debe definir tantas operaciones comunes para las subclases (Leaf y Composite) como sea posible. Component, cuando se trata de una clase abstracta, suele proporcionar implementaciones predeterminadas para estas operaciones, y las subclases las sobrescriben según sea necesario.

Sin embargo, este objetivo a veces entra en conflicto con el principio de diseño de jerarquías de clases, que dice que una clase sólo debe definir las operaciones que sean significativas para sus subclases. Por tanto, si en Component hay muchas operaciones que no tienen razón de ser para las subclases Leaf ¿Qué sentido tiene la implementación por defecto que proporciona Component para ellas (y para Composite, aunque para ésta si tiene sentido)?

Como ejemplo de una operación de este tipo tendríamos un método en Component que añade un componente en una colección. Esta operación, es muy práctica para una subclase Composite, pero ¿cómo se justifica para una subclase Leaf, si este tipo de clases no pueden -por definición- tener hijos? Una opción sería que la implementación predeterminada de Component lanzara una Excepción del tipo 'operación no soportada'. Obviamente, esta operación sería redefinida en las subclase Composite.

¿Dónde declarar las operaciones de gestión de los hijos (manejo de colección)?

Una opción diferente a la de hacer que las clases Leaf tengan que lanzar una excepción en las operaciones relativas al manejo de la colección de componentes hijos, es declarar estas operaciones exclusivamente para las subclases Composite. Esto implica que la clase Component no declare tales operaciones.

Esta decisión nos lleva a un compromiso entre seguridad y transparencia:

- Definir en Component, luego en la raíz de la jerarquía de clases, las operaciones relativas al manejo de la colección de componentes hijos, proporciona transparencia, ya que permite al código cliente tratar a todos los componentes de manera uniforme. La parte negativa de esto, es que tiene un coste en seguridad, debido a que hay un margen para que las clases cliente puedan hacer cosas sin sentido, como ejecutar en una clase Leaf las operaciones de añadir y eliminar objetos hijos.

- Definir exclusivamente en Composite las operaciones relativas al manejo de la colección de componentes hijos, proporciona seguridad, ya que el código cliente no podrá de ninguna manera invocar a esas operaciones sobre un clase Leaf. No obstante, se pierde transparencia porque Leaf y Composite tienen interfaces diferentes.

Desde el punto de vista del patrón Composite, tenemos que hacer énfasis en la transparencia, más que en la seguridad. Poner énfasis en la seguridad supondría hacer conversión de Component a Composite

Si las operaciones relativas al manejo de la colección de componentes hijos se definen en Componente, otra posibilidad que tenemos, es hacer que la implementación para Leaf tenga un significado particular: no añade ni elimina ningún componente hijo, sino que es ella misma quien se añade o elimina de un padre. No obstante, cautela con esto, ya que estamos dando dos significados diferentes para métodos homónimos.

¿Debe Component implementar una lista de componentes?

Si tenemos que Component es una clase abstracta podríamos pensar en definir un atributo tipo Collection que almacenase el conjunto de objetos hijos de un componente. De esta forma, todo objeto tendría su propia colección de objetos hijos. El problema con esto, es que se incurre en un desperdicio de espacio cuando el componente no tiene hijos, esto es, cuando se trata de un Leaf. No obstante, es una opción si hay relativamente pocos objetos hijos en la estructura.

Ordenación de los objetos hijos de un Composite

A veces es necesario que los elementos de un Composite se encuentren ordenados en función de algún criterio. Por ejemplo, en los editores gráficos es habitual que el orden determine la capa en la que se encuentre el elemento (orden Z), es decir, su posición relativa desde el frente hasta el fondo.

Cuando la ordenación de los componentes hijos es un tema importante, es fundamental diseñar cuidadosamente las operaciones de acceso a la colección para preservar el orden entre los objetos. En estos casos el patrón Iterator (pendiente de verlo aún) puede ser de ayuda.

Caché para mejorar el rendimiento

cuando se necesita frecuentemente recorrer o buscar en la colección de objetos, la clase Composite puede almacenar información relativa a sus hijos a modo de caché. Esta información puede tratarse de datos reales o simplemente información conveniente que evite tener que realizar las (lentas) tareas de búsqueda. Por ejemplo, siguiendo con el ejemplo de la aplicación de edición gráfica, una clase Composite podría almacenar las dimensiones de la caja que delimita a cada unos de los elementos gráficos que contiene. Con esto, durante las operaciones de dibujo o selección, la clase Composite podría consultar la caché y evitar así tener que buscar en la colección y recalcular cuando los objetos no se están visualizando en la ventana actual.

Las actualizaciones de caché, como el caso de su invalidación, se pueden hacer mejor cuando los objetos hijos saben quién es su objeto padre. Así, se puede definir una operación invalidarCache() que todo objeto hijo pueda invocar cuando sea necesario.

Patrones relacionados

A veces el enlace que tiene la clase Composite hacia la clase Component es utilizado por el patrón Chain of Responsability.

Decorator suele utilizarse con Composite. Cuando esto sucede, ambos tienen la misma clase base. De esta manera, los decoradores tienen de soportar la interfaz de Component, con operaciones como add(), remove() y getChild().

El patrón Flyweight sirve para compartir componentes, los cuales no pueden referenciar a sus padres.

El patrón Iterator se puede utilizar para recorer composiciones.

El patrón Visitor retiene operaciones y comportamientos que de otra manera estarían distribuidos por las clases Composite y Leaf.