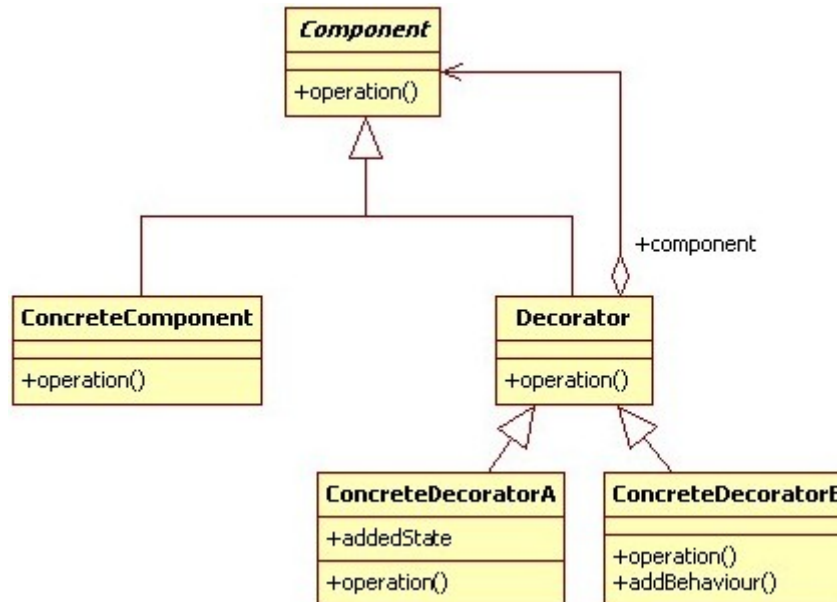


Decorator

Diagrama de clases e interfaces



Intención

El objetivo del patrón Decorador es asignar responsabilidades adicionales a un objeto de forma dinámica, es decir, en tiempo de ejecución. Los decoradores proporcionan una alternativa mucho más flexible para extender la funcionalidad de los objetos que la que proporciona la herencia mediante la creación de subclases.

La subclasificación que proporciona la herencia es un mecanismo muy poderoso, pero:

- Actúa en tiempo de compilación, lo cual puede resultar restrictivo y limitante, ya que hace que nuestro diseño sea inamovible durante la ejecución del programa.
- Resulta inapropiado cuando el número de subclases que se podrían necesitar es elevado. Un diseño que resulte en una compleja jerarquía de clases implica un código difícil de mantener y de probar.

En cambio, con el patrón Decorador podemos extender la funcionalidad en tiempo de ejecución para determinados objetos, los que deseemos, y no para todos los objetos de la clase.

Motivación

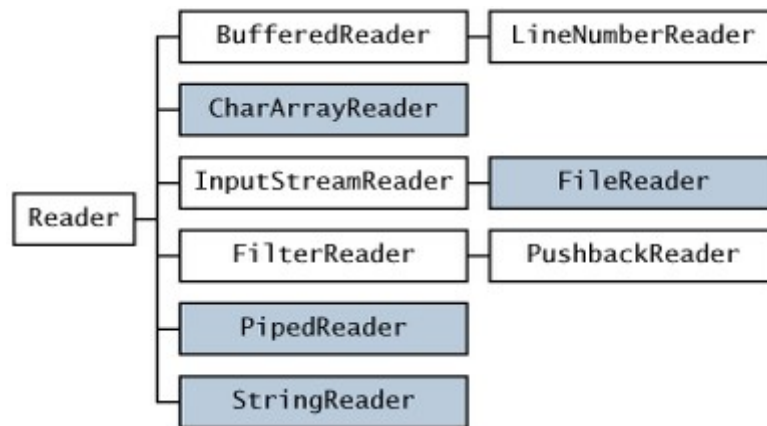
El paquete `java.io` proporciona, entre otras cosas, un conjunto de clases para la lectura de flujos (streams) de entrada. Estas clases se denominan `readers` (lectores) y se caracterizan por tener un nombre que sigue el patrón: `xxxReader`.

Cada lector ofrece una funcionalidad especializada, por ejemplo, existe un lector que lee de un archivo, otro sigue la pista del número de cada línea leída, otro es capaz de retornar al flujo de entrada el último carácter leído, etc. En total, hay diez lectores diferentes para leer los flujos de entrada.

A menudo es necesario combinar las capacidades ofrecidas por los lectores de `java.io`, por ejemplo, es posible que queramos al mismo tiempo leer de un archivo, realizar un seguimiento de los números de línea y, según el valor de cada línea, devolver al flujo el último carácter leído.

Los diseñadores del paquete `java.io` podrían haber utilizado la herencia para proporcionar clases que combinaran todas las posibilidades que ofrece cada lector (o las más interesantes). Por ejemplo, una clase `FileReader` podría tener una subclase `LineNumberFileReader`, que a su vez tuviera una subclase `PushBackLineNumberFileReader`. Sin embargo, el uso de la herencia para combinar las combinaciones con las funcionalidades más útiles de los lectores habría dado lugar a una autentica explosión de las clases.

En la figura siguiente se presentan las clases lectoras de datos divididas en dos categorías: en color azul se muestran las clases que están especializadas en leer de una fuente de datos concreta y en color blanco se muestran las que realizan algún tipo de procesamiento con los datos leídos.



Entonces, ¿cómo consiguieron con tan sólo 10 clases de lectores ofrecer toda la funcionalidad que pueda llegar a requerir un programador? Respuesta: como podemos imaginar, utilizando el patrón Decorator.

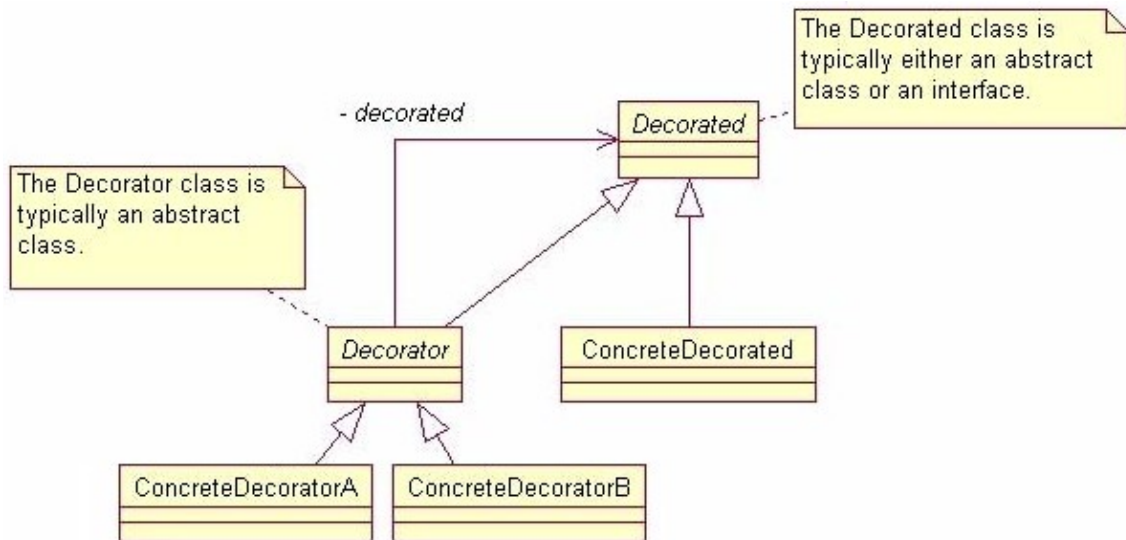
En lugar de utilizar la herencia para extender la funcionalidad de las clases en tiempo de compilación, el patrón Decorator permite agregar funcionalidad a objetos individuales en tiempo de ejecución.

Este se logra envolviendo un objeto dentro de otro objeto. El objeto que hace de envoltorio captura las llamadas dirigidas al objeto envuelto, realiza algún tipo de proceso previo y a continuación redirige la llamada al objeto envuelto. Una vez finalizada la tarea del objeto envuelto, el objeto envolvente puede realizar algún tipo de post proceso.

El objeto envolvente -conocido como decorador- presenta la misma interfaz que la del objeto envuelto, de manera que las clases cliente pueden utilizar el decorador de la misma manera que utilizarían el objeto encapsulado. Esto es muy conveniente, ya que nos permite mantener simple el código cliente.

Implementación

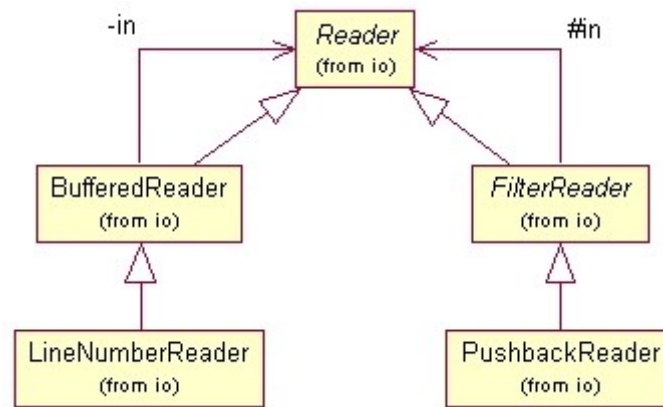
Diagrama de clases para el patrón Decorator:



Descripción de los participantes:

- **Component/Decorated:** Define la interfaz que tiene que implementar todo objeto que deba o tenga la capacidad de extender su funcionalidad o estado dinámicamente, en tiempo de ejecución.
- **ConcreteComponent/ConcreteDecorated:** Define un objeto al que se le pueden añadir nuevas responsabilidades dinámicamente. Es el objeto que será envuelto por el decorador. ConcreteComponent desconoce por completo la existencia de cualquier decorador.
- **Decorator:** Mantiene una referencia a un objeto Component y define una interfaz que se corresponde a la interfaz de Component. Es la interfaz del objeto envoltente.
- **Concrete Decorators:** Estas clases extienden a la clase abstracta Decorator (o la implementan, en caso de ser una interfaz), extendiendo la funcionalidad de Component/Decorated mediante la adición nuevo estado comportamiento.

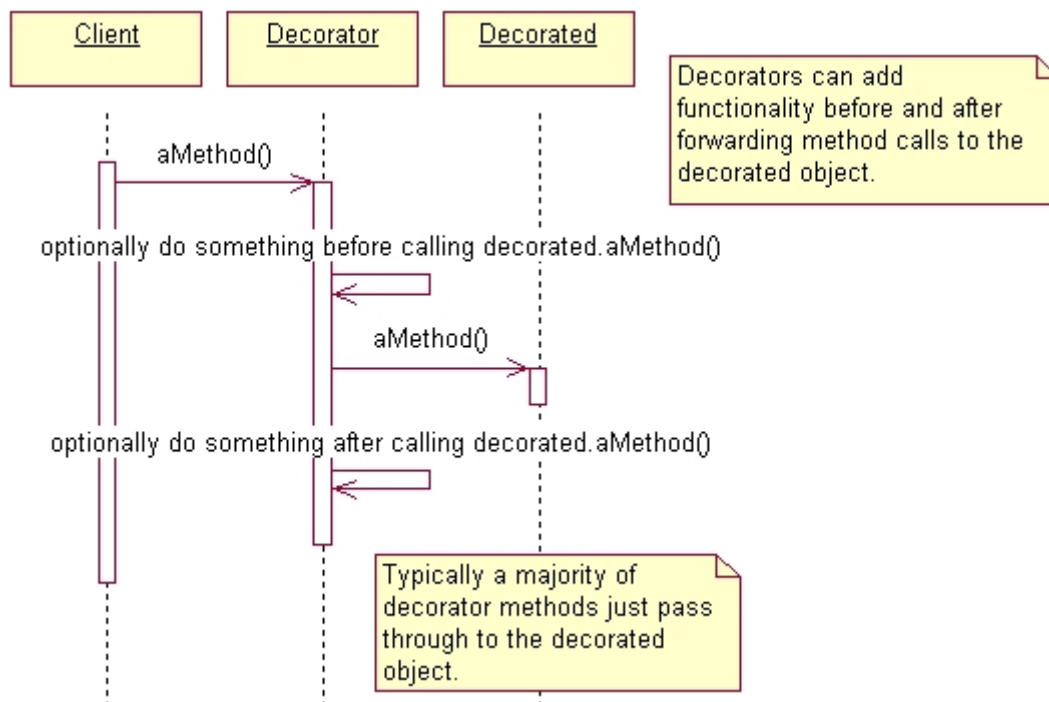
Como ejemplo de cómo se relacionan las clases en el patrón Decorator, en la figura siguiente se presentan cuatro decoradores del paquete java.io:



La clase Reader es un Component/Decorated, que en el caso de java.io se materializa en una clase abstracta.

BufferedReader y FilterReader son decoradores. Ambas clases extienden la clase abstracta Reader, por lo que presentan su misma interfaz, y ambas sostienen una referencia a Reader, a la que reenvían las llamadas que reciben. LineNumberReader y PushbackReader también son decoradores, ya que extienden a BufferedReader y FilterReader, respectivamente.

La figura siguiente muestra como en tiempo de ejecución, los decoradores reenvían las llamadas que reciben a los objetos que contienen (envuelven):



Aplicabilidad y Ejemplos

Podemos usar el patrón Decorator cuando necesitemos:

- Añadir o retirar responsabilidades a objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos de la clase. Esto con la herencia no es posible, ya que para añadir responsabilidades se requiere crear nuevas clases, lo que resulta en un sistema complejo, con muchas clases.
- Cuando la extensión mediante subclases no sea práctico. Por ejemplo, si el diseño de una jerarquía de clases, con tal de admitir todas las combinaciones posibles de funcionalidad o las más importantes, resulta en un gran número de subclases, entonces estamos ante un diseño problemático. Con el patrón Decorador podemos combinar diferentes responsabilidades para un componente, obteniendo mucha variabilidad a partir de muy pocas clases.
- Cuando la extensión mediante subclases no sea posible, por ejemplo, porque la superclase esté oculta o no disponible por cualquier otro motivo para permitir la creación de subclases.

Ejemplo 1. Algunas clases del paquete java.io

Comenzaremos por analizar un ejemplo basado en las explicaciones vistas anteriormente en la sección *Motivación*. No vamos a crear las diferentes clases e interfaces que necesitaríamos para implementar el patrón, sino que vamos a partir de un diseño existente: algunas de las clases de la jerarquía java.io.Reader. Por tanto, el énfasis en este primer ejemplo, lo pondremos en el código cliente, esto es, en ver cómo se utiliza un conjunto de clases que siguen el patrón Decorator.

El siguiente ejemplo lee e imprime por pantalla un archivo de texto cuyo nombre tiene que ser proporcionado por el usuario en el momento de ejecutar el programa. El archivo debe localizarse en el mismo paquete que la clase.

El programa utiliza un objeto FileReader para leer un stream de caracteres desde un fichero. Además, se utiliza un decorador, la clase LineNumberReader, para aumentar la funcionalidad de FileReader. LineNumberReader no sabe leer caracteres desde una fuente real de datos, por lo que tiene que redirigir las peticiones que le llegan al objeto

que encierra (envuelve), como por ejemplo, el método `readLine()`. `readLine()` se traduce en una redirección hacia el método `read()` de `FileReader`.

Lo que sí sabe hacer muy bien `LineNumberReader` es obtener los números de línea a partir de un lector, los cuales se pueden acceder con `LineNumberReader.getLineNumber()`. Debido a que `LineNumberReader` es un decorador, podemos fácilmente obtener los números de línea de cualquier tipo de lector, no solamente de lectores de ficheros.

El decorador también imprime los números de línea y transforma los tabuladores existentes en el fichero por tres espacios en blanco.

Ejemplo1Decorator.java

```
package estructurales.decorator.ficheros;

import java.io.FileReader;
import java.io.LineNumberReader;
import java.net.URL;

public class Ejemplo1Decorator {

    public static void main(String args[]) {
        if(args.length < 1) {
            System.err
                .println("Usar: " + "java Ejemplo1Decorator NomFichero");
            System.exit(1);
        }
        new Ejemplo1Decorator(args[0]);
    }

    public Ejemplo1Decorator(String filename) {
        try {
            URL url = this.getClass().getResource(filename);

            FileReader frdr = new FileReader(url.getFile());
            LineNumberReader lrdr = new LineNumberReader(frdr);

            for(String linea; (linea = lrdr.readLine()) != null;) {
                // Mostrar numero de linea
                System.out.print(lrdr.getLineNumber() + ":\t");
                // Mostrar la linea
                mostrarLinea(linea);
            }
        }
        catch(java.io.FileNotFoundException fnfx) {
            fnfx.printStackTrace();
        }
        catch(java.io.IOException iox) {
            iox.printStackTrace();
        }
    }
}
```

```

private void mostrarLinea(String s) {
    for (int c, i=0; i < s.length(); ++i) {
        c = s.charAt(i);
        if (c == '\t') {
            System.out.print("  ");
        } else {
            System.out.print((char)c);
        }
    }
    System.out.println();
}
}

```

Salida (parcial):

```

<terminated> Ejemplo1Decorator [Java Application] D:\java\jdk\jdk1.6.0_29\bin\javaw.exe (27/11/2011 10:43:06)
1:      Tengo el honor de estar hoy aquí presente en la ceremonia de g
2:
3:      El primer relato es acerca de unir los distintos puntos.
4:      Abandoné los estudios en Reed College después de los primeros
5:      Todo comenzó antes de que yo naciera. Mi madre biológica era u
6:      Y 17 años más tarde fui a la universidad. Pero ingenuamente el
7:      No todo fue romántico. No tenía un dormitorio, así que dormía

```

Alternativamente podríamos utilizar el siguiente estilo de codificación, más conciso:

```

LineNumberReader lrd =
    new LineNumberReader(new FileReader(url.getFile()));

```

Si nos fijamos en las clases Reader de java.io (mostradas anteriormente en la figura del principio del documento), encontraremos otro decorador: `BufferedReader`. Esta clase almacena en un buffer varios caracteres, obtenidos todos en un sólo acceso, en lugar de obtener de la fuente de datos los caracteres de uno en uno. Por esto es más eficiente que un lector sin buffer. Por tanto, podríamos decidir hacer más eficiente el código anterior de esta manera:

```

...
FileReader frdr = new FileReader(url.getFile());
BufferedReader brdr =
    new BufferedReader(frdr); // almacenar en un buffer
LineNumberReader lrd = new LineNumberReader(brdr);
...

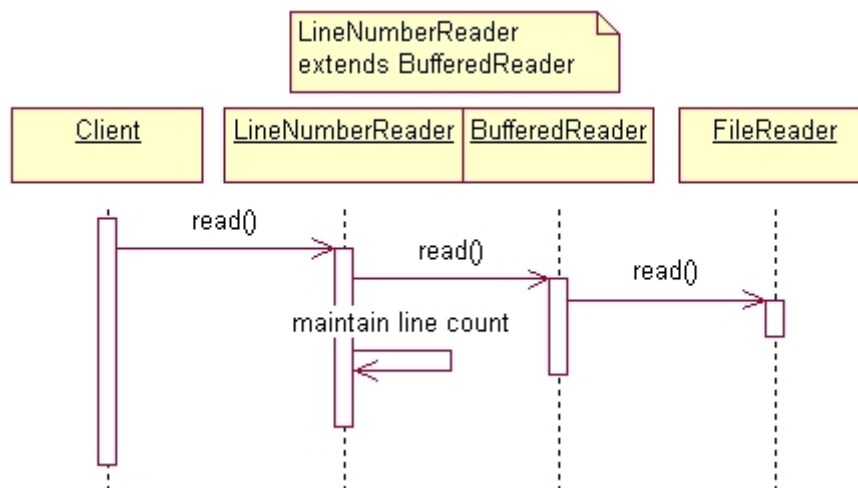
```

Esto pone de manifiesto que es posible encadenar decoradores.

Nota: En realidad, en nuestro ejemplo el programa se ejecutará más lento al utilizar el buffer intermedio (`BufferedReader`). Esto se debe a que `LineNumberReader` es una

subclase de `BufferedReader`. Por tanto, `LineNumberReader` ya utiliza buffer y lo que estamos haciendo es sobrecargar el programa, ya que por un lado, `Buffered` se encargará de almacenar los caracteres en buffer mientras que `LineNumberReader` hará lo mismo por otro lado.

La figura siguiente muestra las interacciones en tiempo de ejecución de las clases del ejemplo:



Ejemplo 2. Decoradores para ordenar y filtrar tablas Swing

Las tablas Swing son un medio muy utilizado para mostrar información tabular en aplicaciones GUI. Estas tablas pueden ser objetos interesantes de envolver mediante decoradores que puedan extender la funcionalidad básica de las tablas (la presentación de los datos), por ejemplo, aportando la posibilidad de ordenar y filtrar la información de la tabla.

Antes de que podamos ver en detalle cómo los decoradores pueden extender las tablas Swing, hagamos un repaso básico de este tipo de tablas.

Tablas Swing

Los componentes Swing se implementan con el patrón de diseño Model-View-Controller (MVC). Cada componente está formado por un modelo (los datos), las vistas que muestran los datos, y los controladores que reaccionan a los eventos. Por ejemplo, cuando creamos un objeto `JTable` normalmente lo hacemos proporcionando

explícitamente el modelo para la tabla. El siguiente programa muestra una aplicación que hace precisamente eso: crea una tabla a partir de un modelo. La tabla se ubica dentro de un contenedor que mostrará una barra de desplazamiento en caso de que las filas excedan la altura de la ventana principal.

TestTabla.java

```
package estructurales.decorator.tablas;

import javax.swing.*.*;
import javax.swing.table.*;

public class TestTabla extends JFrame {

    private static final long serialVersionUID = 1L;

    public static void main(String args[]) {
        TestTabla frame = new TestTabla();
        frame.setTitle("Tablas y Modelos");
        frame.setBounds(300, 300, 450, 300);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.setVisible(true);
    }

    public TestTabla() {
        // Creamos un objeto Modelo
        TableModel modelo = new Modelo();

        // Creamos una tabla y proporcionamos su modelo
        JTable tabla = new JTable(modelo);

        // Creamos un marco para la tabla capaz de permitir scroll
        JScrollPane marco = new JScrollPane(tabla);

        // Añadimos el marco al JFrame
        getContentPane().add(marco);
    }

    /*
     * Clase privada
     */
    private static class Modelo extends AbstractTableModel {

        private static final long serialVersionUID = 1L;

        final int rows = 100, cols = 10;

        @Override public int getRowCount() {
            return rows;
        }

        @Override public int getColumnCount() {
            return cols;
        }

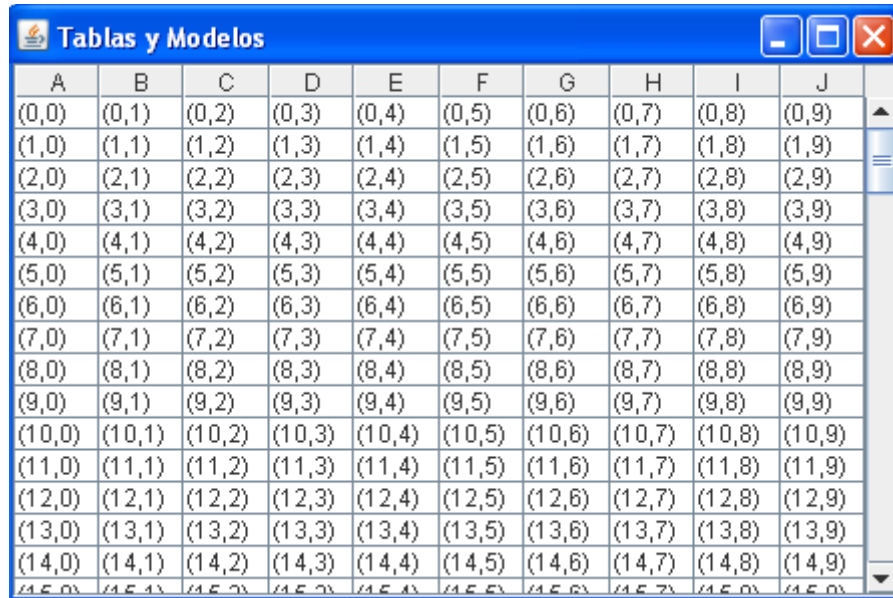
        @Override public Object getValueAt(int row, int col) {
```

```

        }
        return "(" + row + "," + col + ")";
    }
}

```

Salida:



| A | B | C | D | E | F | G | H | I | J |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| (0,0) | (0,1) | (0,2) | (0,3) | (0,4) | (0,5) | (0,6) | (0,7) | (0,8) | (0,9) |
| (1,0) | (1,1) | (1,2) | (1,3) | (1,4) | (1,5) | (1,6) | (1,7) | (1,8) | (1,9) |
| (2,0) | (2,1) | (2,2) | (2,3) | (2,4) | (2,5) | (2,6) | (2,7) | (2,8) | (2,9) |
| (3,0) | (3,1) | (3,2) | (3,3) | (3,4) | (3,5) | (3,6) | (3,7) | (3,8) | (3,9) |
| (4,0) | (4,1) | (4,2) | (4,3) | (4,4) | (4,5) | (4,6) | (4,7) | (4,8) | (4,9) |
| (5,0) | (5,1) | (5,2) | (5,3) | (5,4) | (5,5) | (5,6) | (5,7) | (5,8) | (5,9) |
| (6,0) | (6,1) | (6,2) | (6,3) | (6,4) | (6,5) | (6,6) | (6,7) | (6,8) | (6,9) |
| (7,0) | (7,1) | (7,2) | (7,3) | (7,4) | (7,5) | (7,6) | (7,7) | (7,8) | (7,9) |
| (8,0) | (8,1) | (8,2) | (8,3) | (8,4) | (8,5) | (8,6) | (8,7) | (8,8) | (8,9) |
| (9,0) | (9,1) | (9,2) | (9,3) | (9,4) | (9,5) | (9,6) | (9,7) | (9,8) | (9,9) |
| (10,0) | (10,1) | (10,2) | (10,3) | (10,4) | (10,5) | (10,6) | (10,7) | (10,8) | (10,9) |
| (11,0) | (11,1) | (11,2) | (11,3) | (11,4) | (11,5) | (11,6) | (11,7) | (11,8) | (11,9) |
| (12,0) | (12,1) | (12,2) | (12,3) | (12,4) | (12,5) | (12,6) | (12,7) | (12,8) | (12,9) |
| (13,0) | (13,1) | (13,2) | (13,3) | (13,4) | (13,5) | (13,6) | (13,7) | (13,8) | (13,9) |
| (14,0) | (14,1) | (14,2) | (14,3) | (14,4) | (14,5) | (14,6) | (14,7) | (14,8) | (14,9) |

La aplicación mostrada:

- Crea un modelo con 100 filas y 10 columnas. Cuando se pregunta por el valor de una celda, el modelo crea un String a partir de la fila y la columna actual.
- Usa ese modelo para construir la tabla Swing (una instancia de JTable).
- Los modelos de tabla, además de metadatos como el número de filas y columnas, proporcionan los datos para la tabla.

Es importante observar que las tablas Swing necesitan modelos que mantienen los datos de las tablas. Sin embargo, como muestra el ejemplo, realmente los modelos no tienen que almacenar los datos; basta con que proporcionen un valor para una fila y una columna dada.

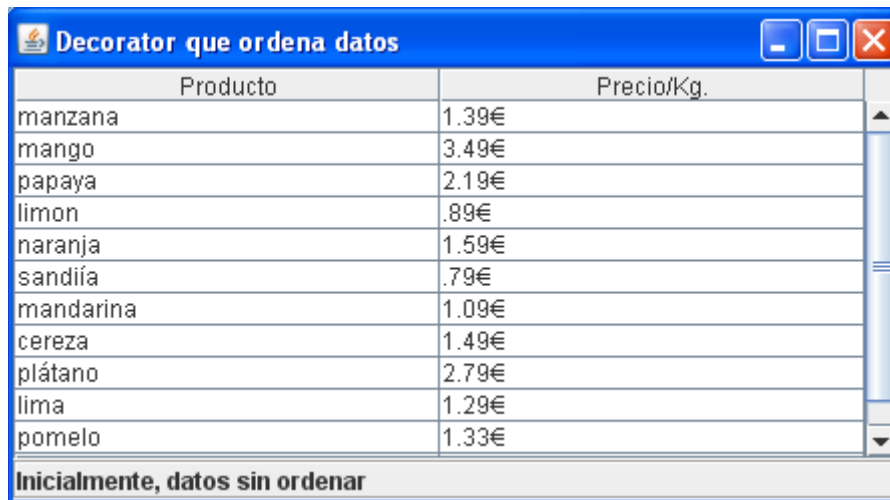
Un decorador para ordenar datos

Con un conocimiento básico del patrón Decorador y de tablas Swing, vamos a implementar un decorador que añadirá la funcionalidad de ordenar una tabla.

NOTA: Las versiones de Swing actuales (de la 1.5 para adelante) ya incluyen en el API clases para ordenar y filtrar tablas. Estas clases simplifican mucho el trabajo con tablas y deben ser nuestra primera opción. No obstante, para ilustrar el trabajo con decoradores no vamos a usar estas facilidades que nos proporciona el lenguaje, sino que las vamos a implementar nosotros mismos.

En el ejemplo anterior construíamos la tabla Swing con un modelo que mostraba en cada celda los valores para los índices de la fila y la columna en curso. En esta nueva aplicación crearemos la tabla con un decorador capaz de ordenar una serie de productos tanto por su descripción como por su precio correspondiente.

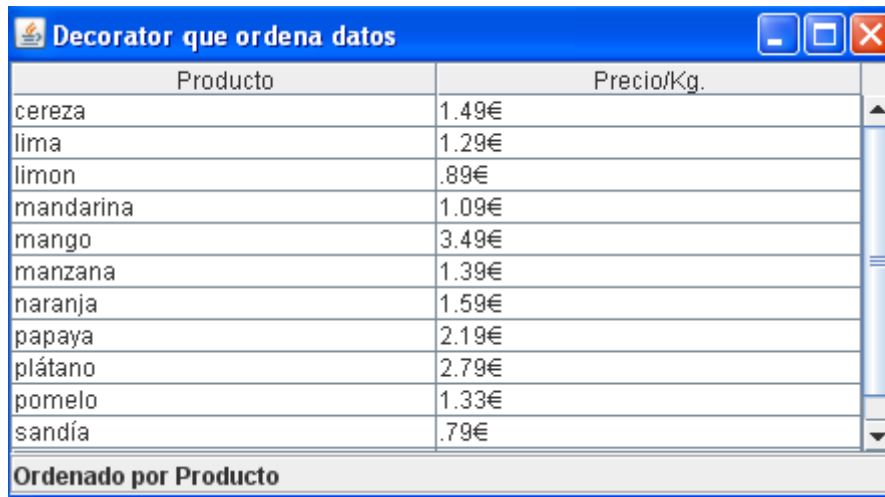
Cuando iniciemos la aplicación los datos aparecerán sin ningún orden en especial, sencillamente en el mismo orden en el que se han leído. En la barra de estado (parte inferior de la ventana) podemos ver un mensaje indicando esta circunstancia:



| Producto | Precio/Kg. |
|-----------|------------|
| manzana | 1.39€ |
| mango | 3.49€ |
| papaya | 2.19€ |
| limon | .89€ |
| naranja | 1.59€ |
| sandiía | .79€ |
| mandarina | 1.09€ |
| cereza | 1.49€ |
| plátano | 2.79€ |
| lima | 1.29€ |
| pomelo | 1.33€ |

Inicialmente, datos sin ordenar

Al pulsar sobre la cabecera de la primera columna los datos aparecerán ordenados alfabéticamente por la descripción. En la barra de estado podemos ver el mensaje que nos informa del orden que siguen ahora los datos:



| Producto | Precio/Kg. |
|-----------|------------|
| cereza | 1.49€ |
| lima | 1.29€ |
| limon | .89€ |
| mandarina | 1.09€ |
| mango | 3.49€ |
| manzana | 1.39€ |
| naranja | 1.59€ |
| papaya | 2.19€ |
| plátano | 2.79€ |
| pomelo | 1.33€ |
| sandía | .79€ |

Ordenado por Producto

Al pulsar sobre la cabecera de la segunda columna los datos aparecerán ordenados de menor a mayor precio. En la barra de estado podemos ver el mensaje que nos informa del orden que siguen ahora los datos:



| Producto | Precio/Kg. |
|-----------|------------|
| sandía | .79€ |
| limon | .89€ |
| mandarina | 1.09€ |
| lima | 1.29€ |
| pomelo | 1.33€ |
| manzana | 1.39€ |
| cereza | 1.49€ |
| naranja | 1.59€ |
| uva | 2.03€ |
| papaya | 2.19€ |
| plátano | 2.79€ |

Ordenado por Precio/Kg.

Diseño del programa

El programa tiene dos clases cliente que se encargan de:

- Dibujar la interfaz gráfica: la ventana que visualiza el usuario, la tabla, una barra de estado, etc. La creación de la tabla implica proporcionarle un modelo de datos (las frutas y sus precios) que está desordenado.
- Obtener los textos literales de la aplicación a partir de ficheros de propiedades (un fichero por cada idioma, según la configuración regional del ordenador). Los textos abarcan: el título, los mensajes de la barra de estado, los productos (frutas) y precios.

- Crear el decorador a partir del modelo real de la tabla que permitirá ordenar sus datos.
- Capturar y gestionar el evento que se produce al hacer clic sobre la tabla. Esto implica invocar al método de ordenación del decorador.

Como vemos las responsabilidades de las clases cliente son diversas:

- Iniciar la aplicación
- Dar un aspecto gráfico al programa.
- Crear varios objetos. Principalmente, la tabla y el decorador. Hacer que el decorador encapsule el modelo de la tabla.
- Interactuar con el usuario: capturar los clics de ratón para llevar a cabo la ordenación.

A continuación se muestran dos ficheros de propiedades, el primero para el español y el segundo para el inglés. Ambos de estar ubicados en el mismo paquete que el código cliente:

recursos_es.properties

```

titulo=Decorator que ordena datos
msgInicial=Inicialmente, datos sin ordenar
msgOrden=Ordenado por
cabeceraProducto=Producto
cabeceraPrecio=Precio/Kg.
pro001=manzana
pro002=mango
pro003=papaya
pro004=limon
pro005=naranja
pro006=sand\u00E9da
pro007=mandarina
pro008=cereza
pro009=pl\u00E9tano
pro010=lima
pro011=pomelo
pro012=uva
pre001=1.39\u20AC
pre002=3.49\u20AC
pre003=2.19\u20AC
pre004=.89\u20AC
pre005=1.59\u20AC
pre006=.79\u20AC
pre007=1.09\u20AC
pre008=1.49\u20AC
pre009=2.79\u20AC

```

```
pre010=1.29\u20AC
pre011=1.33\u20AC
pre012=2.03\u20AC
```

recursos_en.properties

```
titulo=Sorter Decorator
msgInicial=Nothing sorted originally
msgOrden=Sorted by
cabeceraProducto=Item
cabeceraPrecio=Price/Lb.
pro001=apple
pro002=mango
pro003=papaya
pro004=lemon
pro005=orange
pro006=watermelon
pro007=tangerine
pro008=cherry
pro009=banana
pro010=lima
pro011=grapefruit
pro012=grapes
pre001=$1.39
pre002=$3.49
pre003=$2.19
pre004=$.89
pre005=$1.59
pre006=$.79
pre007=$1.09
pre008=$1.49
pre009=$2.79
pre010=$1.29
pre011=$1.33
pre012=$2.03
```

Veamos el código fuente de estas clases:

MainClient.java

```
package estructurales.decorator.tablas.client;

import java.awt.*;
import java.awt.event.*;

import javax.swing.*;
import javax.swing.table.*;

import estructurales.decorator.tablas.TableSortDecorator;

public class MainClient extends JFrame {

    private static final long serialVersionUID = 1L;

    /*
     * Definición de la tabla.
     */
}
```

```

    * Se obtienen los datos de la tabla desde un archivo
    * de propiedades. La clase App es una clase de ayuda
    * para MainClient.
    */
    final String[] cabeceras = {
        App.getValor("cabeceraProducto"),
        App.getValor("cabeceraPrecio")
    };

    JTable table = new JTable(new Object[][] {
        { App.getValor("pro001"), App.getValor("pre001") },
        { App.getValor("pro002"), App.getValor("pre002") },
        { App.getValor("pro003"), App.getValor("pre003") },
        { App.getValor("pro004"), App.getValor("pre004") },
        { App.getValor("pro005"), App.getValor("pre005") },
        { App.getValor("pro006"), App.getValor("pre006") },
        { App.getValor("pro007"), App.getValor("pre007") },
        { App.getValor("pro008"), App.getValor("pre008") },
        { App.getValor("pro009"), App.getValor("pre009") },
        { App.getValor("pro010"), App.getValor("pre010") },
        { App.getValor("pro011"), App.getValor("pre011") },
        { App.getValor("pro012"), App.getValor("pre012") }, },
        cabeceras);

    public static void main(String args[]) {
        App.launch(new MainClient(), "titulo", 300, 300, 450, 250);
    }

    /*
     * Constructor.
     */
    public MainClient() {
        // Crea el decorador que decorará el modelo real de la
        tabla.
        // Notad cómo se construye a partir del modelo real (cómo
        lo envuelve).

        final TableSortDecorator decorador =
            new TableSortDecorator(table.getModel());

        /*
         * Establecer el decorador como el modelo de la tabla. Esto
         * es posible porque el decorador implementa la interfaz
        TableModel.
        */
        table.setModel(decorador);

        // Hacer que la tabla quede dentro de un panel con scroll.
        getContentPane().add(new JScrollPane(table),
        BorderLayout.CENTER);

        // Añadir un area de estado a la ventana.
        getContentPane().add(App.getStatusArea(),
        BorderLayout.SOUTH);

        App.showStatus(App.getValor("msgInicial"));

        // Obtener una referencia a la cabecera de la tabla

```



```

        JTableHeader cabeceraTabla = (JTableHeader)
table.getTableHeader();

        // Captar los click de raton sobre la cabecera de la tabla
        // y llamar al metodo de ordenacion del decorador.
        cabeceraTabla.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                TableColumnModel tcm = table.getColumnModel();

                /*
                * Ahora atencion: Los indices de las columnas
                * tienen porque corresponder con las columnas
                * que hay que realizar un mapeo. El motivo de
                * corresponderse es variado: podrian mostrarse
                * en la vista en orden
                * inverso, ocultas, etc.
                */

                // Obtener el indice de la columna clicada
                int col_en_vista =
tcm.getColumnIndexAtX(e.getX());

                // Convertir ese indice en el que equivalente
del modelo
                int col_en_modelo =
table.convertColumnIndexToModel(col_en_vista);

                // Ordenar por la columna pulsada
                decorador.ordenar(col_en_modelo);

                // Actualizar el area de estado
                App.showStatus(App.getValor("msgOrden")+ " " +
cabeceras[col_en_modelo]);
            }
        });
    }
}
-

```

App.java

```

package estructurales.decorator.tablas.client;

import java.awt.FlowLayout;
import java.awt.event.WindowAdapter;
import java.util.Locale;
import java.util.ResourceBundle;

import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

```

```

/*
 * Esta clase es de apoyo para la clase MainClient.
 * Basicamente se encarga de:
 * -Gestionar el fichero de propiedades, que contiene
 * los literales para la aplicación.
 * -Dibujar la interfaz gráfica.
 *
 * Notad que tiene acceso de paquete
 */
class App extends WindowAdapter {

    static private JPanel barraEstado = new JPanel();
    static private JLabel lblEstado = new JLabel(" ");

    // Fichero de recursos
    static private ResourceBundle recursos;

    // Nombre del paquete de la clase
    static private String nomPaquete =
        App.class.getPackage().getName();

    // Ubicacion para cargar fichero de recursos
    static private String nomBaseFicheroRecursos =
        nomPaquete + ".recursos";

    /*
     * Inicializador estático responsable de cargar el fichero
     * de propiedades que contiene los literales de la aplicación.
     *
     * El fichero que se selecciona es el que se corresponde con
     * la configuración regional del ordenador donde se ejecuta
     * la aplicación.
     *
     * Los ficheros de propiedades (uno por cada idioma) deben
     * encontrarse en el mismo paquete que la clase App, es
     * decir en 'estructurales.decorator.tablas.client' y deben
     * llamar 'recursos_XX_YY', donde XX es el código de idioma
     * e YY la posible variación del lenguaje. Por ejemplo,
     * "español España" sería es_ES.
     */
    static {
        // Configuración regional en Español
        recursos = ResourceBundle
            .getBundle(nomBaseFicheroRecursos,
Locale.getDefault());
        /*
         * Si queremos simular que ejecutamos la aplicación con una
         * configuración anglosajona, tan solo tenemos que utilizar
         * la siguiente sentencia y comentar la anterior.
         */
        /*recursos = ResourceBundle
            .getBundle(nomBaseFicheroRecursos, Locale.ENGLISH);*/
    }

    // Constructor privado. De esta manera evitamos que la clase
    // se puede instanciar
    private App() {

    }
}

```

```

    /**
     * Metodo que acondiciona gráficamente a MainClient (el JFrame).
     * Le añade una barra de estado, bordes, etc.
     */
    static void launch(final JFrame f, String title,
                       final int x, final int y, final int w, int h)
    {
        barraEstado.setBorder(BorderFactory.createEtchedBorder());
        barraEstado.setLayout(new FlowLayout(FlowLayout.LEFT, 0,
0));
        barraEstado.add(lblEstado);
        lblEstado.setHorizontalAlignment(JLabel.LEFT);
        f.setTitle(getValor(title));
        f.setBounds(x, y, w, h);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }

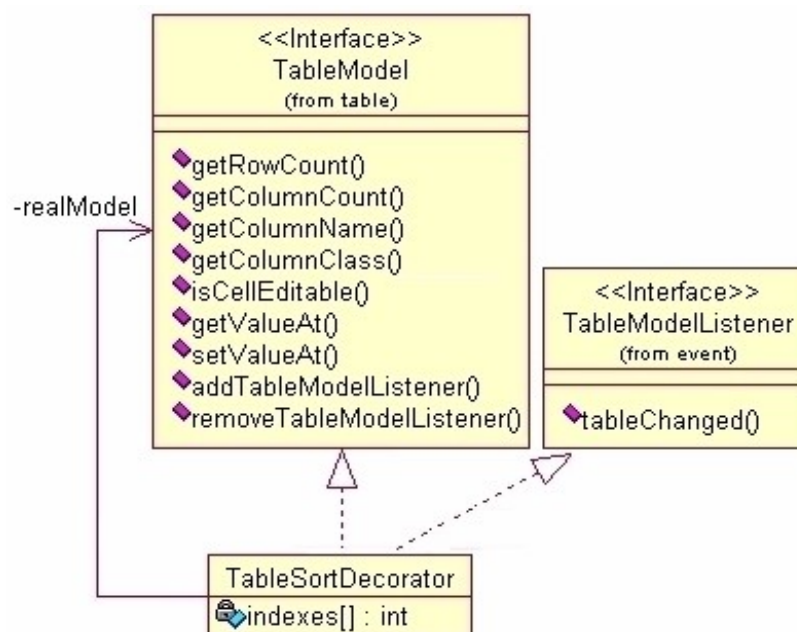
    static JPanel getStatusArea() { return barraEstado; }

    static void showStatus(String s) {
        lblEstado.setText(s);
    }

    /**
     * Este método retorna el valor leído del fichero de propiedades
     * que corresponde a la clave recibida por parámetro.
     */
    static String getValor(String clave) {
        if (recursos != null) {
            return recursos.getString(clave);
        }
        return null;
    }
}

```

Sin embargo, el código interesante para este ejemplo no son las clases cliente, sino el decorador, al que hemos llamado TableSortDecorator. Para entender esta clase es necesario ver su diagrama de clases:



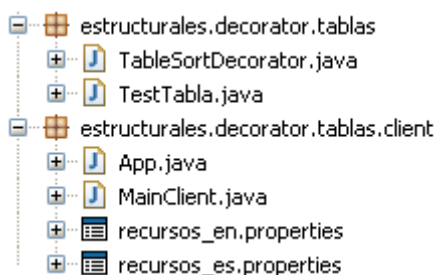
Nota: Las interfaces `TableModel` y `TableModelListener` se encuentran en la API de Java, en el paquete `javax.swing.table` y `javax.swing.event`, respectivamente.

Para que un objeto de TableSortDecorator pueda decorar el modelo de JTable es necesario que TableSortDecorator:

- Implemente la interfaz TableModel, de esta manera está obligado a implementar los mismos métodos que soporta un modelo de una JTable. La interfaz TableModelListener no es estrictamente necesaria, aunque en la práctica, su método tablaChanged() es clave para que TableSortDecorator sepa cuando ha cambiado el modelo real (los datos o la propia estructura del modelo).
- Sostenga una referencia al modelo que decora. Esa referencia apuntará al modelo real de una tabla.

Notad que esto es consistente con el diseño del patrón Decorator.

Para que no haya dudas sobre las clases y otros ficheros que tenemos que crear, a continuación se muestra una imagen de los paquetes y del contenido de cada uno:



Recordad que la clase TestTabla la creamos cuando vimos cómo crear una JTable y proporcionar un modelo. Para el ejemplo actual es totalmente prescindible.

Bien, veamos ahora el código:

TableSortDecorator.java

```
package estructurales.decorator.tablas;

import javax.swing.event.TableModelEvent;
import javax.swing.event.TableModelListener;
import javax.swing.table.TableModel;

/*
 * Decorador que envuelve el modelo real y le añade
 * la capacidad de ordenarlo.
 */
public class TableSortDecorator
    implements TableModel, TableModelListener {
```

```

// Modelo real que vamos a decorar
private TableModel realModel;

/*
 * Este metodo es utilizado por las subclases para
 * acceder al modelo real
 */
protected TableModel getRealModel() {
    return realModel;
}

/*
 * Array de enteros que permite al decorador controlar
 * la ordenación del modelo real sin alterarlo.
 * Cada posición del array se corresponde con una fila
 * de la tabla.
 */
private int indices[];

/*
 * Constructor.
 * Recibe el modelo real como parametro.
 */
public TableSortDecorator(TableModel model) {
    this.realModel = model;
    /*
     * La propia clase TableSortDecorator se registra
     * como oyente de eventos que se produzcan sobre
     * el modelo real.
     */
    realModel.addTableModelListener(this);
    //
    inicializarIndices();
}

/* *****
 * El siguientes metodo esta definidos en la interfaz
 * TableModelListener.
 */

/*
 * Cuando el modelo cambia se debe llamar al metodo que
 * inicializa el array de indices.
 */
@Override
public void tableChanged(TableModelEvent e) {
    inicializarIndices();
}

/* *****
 * Los siguientes nueve metodos estan definidos en la
 * interfaz TableModel. Todos invocan a los metodos
 * del modelo real.
 */

/*
 * 1°. Retorna el objeto Class de la columna del modelo
 * real, según el indice recibido por parametro.
 */

```

```

@Override
public Class<?> getColumnClass(int columnIndex) {
    return realModel.getColumnClass(columnIndex);
}

/*
 * 2°. Retorna el numero total de columnas del modelo real.
 */
@Override
public int getColumnCount() {
    return realModel.getColumnCount();
}

/*
 * 3°. Retorna el nombre de la columna del modelo
 * real, según el indice recibido por parametro.
 */
@Override
public String getColumnName(int columnIndex) {
    return realModel.getColumnName(columnIndex);
}

/*
 * 4°. Retorna el total de filas del modelo real.
 */
@Override
public int getRowCount() {
    return realModel.getRowCount();
}

/*
 * 5°. Retorna un booleano para indicar si la celda
 * indicada por los indices de los parametros es o no editable.
 */
@Override
public boolean isCellEditable(int row, int column) {
    return realModel.isCellEditable(row, column);
}

/*
 * 6°. Retorna el valor de la celda indicada por los
 * indices de los parametros.
 */
@Override
public Object getValueAt(int row, int column) {
    return realModel.getValueAt(indices[row], column);
}

/*
 * 7°. Establece el valor recibido como primer parametro
 * en la la celda indicada por el resto de parametros.
 */
@Override
public void setValueAt(Object aValue, int row, int column) {
    realModel.setValueAt(aValue, indices[row], column);
}

/*
 * 8°. Añade el objeto recibido por parametro a la lista

```

```

    * de oyentes de eventos de modificacion en el modelo real.
    */
@Override
public void addTableModelListener(TableModelListener l) {
    realModel.addTableModelListener(l);
}

/*
 * 9. Quita el objeto recibido por parametro de la lista
 * de oyentes de eventos de modificacion en el modelo real.
 */
@Override
public void removeTableModelListener(TableModelListener l) {
    realModel.removeTableModelListener(l);
}

/* *****
 * Los siguientes metodos sirven para ordenar mediante el
 * algoritmo de la burbuja.
 * ***** */

/*
 * Este método es invocado desde el codigo cliente al hacer
 * clic sobre alguna columna de la tabla.
 *
 * Parametros:
 *     column: la columna a ordear (la fruta o el precio)
 *
 * Utiliza dos bucles para recorrer las celdas de la columna
 * correspondiente y comparar sus valores. Cuando encuentra
 * que debe ordenar, llama al método swap para intercambiar
 * las posiciones del array indices.
 */
public void ordenar(int column) {
    int rowCount = getRowCount();
    for (int i = 0; i < rowCount; i++) {
        for (int j = i + 1; j < rowCount; j++) {
            if (compare(indices[i], indices[j], column) <
0) {
                swap(i, j); // Hay que ordenar
            }
        }
    }
}

/*
 * Para una columna dada, compara los valores de dos filas
 * del modelo real.
 * Parametros:
 *     i: una fila
 *     j: otra fila
 *     column: la columna (fruta o precio)
 * El método compareTo() devolverá:
 *     -1 si valorFila_i < valorFila_j
 *     0 si valorFila_i = valorFila_j
 *     1 si valorFila_i > valorFila_j
 */

```



```

        private int compare(int i, int j, int column) {
            TableModel realModel = this.realModel;
            Object valorFila_i = realModel.getValueAt(i, column);
            Object valorFila_j = realModel.getValueAt(j, column);
            int resultado =
valorFila_j.toString().compareTo(valorFila_i.toString());
            return (resultado < 0) ? -1 : ((resultado > 0) ? 1 : 0);
        }

        /**
         * Intercambia las posiciones del array
         */
        private void swap(int i, int j) {
            int tmp = indices[i];
            indices[i] = indices[j];
            indices[j] = tmp;
        }

        /**
         * Rellenar el array de indices, utilizando como
         * valor la misma posicion del indice.
         */
        private void inicializarIndices() {
            indices = new int[getRowCount()];
            for (int i = 0; i < indices.length; ++i) {
                indices[i] = i;
            }
        }
    }
}

```

Fijaos en los últimos cuatro métodos. El decorador los usa para ordenar los datos del modelo que encapsula sin alterarlo. Para ello, utiliza un array de enteros que representan posiciones de filas ordenadas.

La característica básica de todo decorador es hacerse pasar por el objeto que envuelve. En nuestro caso, el decorador se hace pasar por el modelo del JTable, por lo que en algún momento se le preguntará por el valor de una celda (intersección de una fila y una columna).

El decorador utilizará el valor de la fila de la tabla como índice de su array de índices interno. Cuando el usuario clique en una columna para ordenar la tabla por ese criterio, el decorador utilizará el método de ordenación de la burbuja (bubble sort) para ordenar su array interno en función de la ordenación que le correspondería al modelo real, pero sin tocar el modelo real. De esta manera, consigue mantener el modelo en su estado inicial, por si fuera de interés.

Por otro lado, también es importante observar que `TableSortDecorator` implementa la interfaz `TableModelListener` y se registra como un oyente (listener) del modelo real para eventos de cambio en la tabla. Cuando el modelo real dispara un evento de este tipo, el decorador vuelve a inicializar los índices de su array interno. Esto se produce en el método `tableChanged()`.

`TableSortDecorator` tiene once métodos públicos. Nueve de esos métodos realizan una cierta tarea y finalmente delegan el resto del trabajo en el modelo real. De los nueve métodos, sólo dos métodos - `getValueAt()` y `setValueAt()` - no realizan tarea alguna en `TableSortDecorator` y se limitan a delegar toda la responsabilidad en el modelo real. Esto es normal en los decoradores, ya que la mayoría suelen añadir una pequeña cantidad de comportamiento al objeto decorado, el cual ya de por sí realiza varias funciones.

Refactorizando el decorador de ordenación

El decorador que hemos visto, `TableSortDecorator`, funciona según lo explicado: decora el modelo de un `JTable` en tiempo de ejecución, sin alterar el modelo real, lo que nos da flexibilidad.

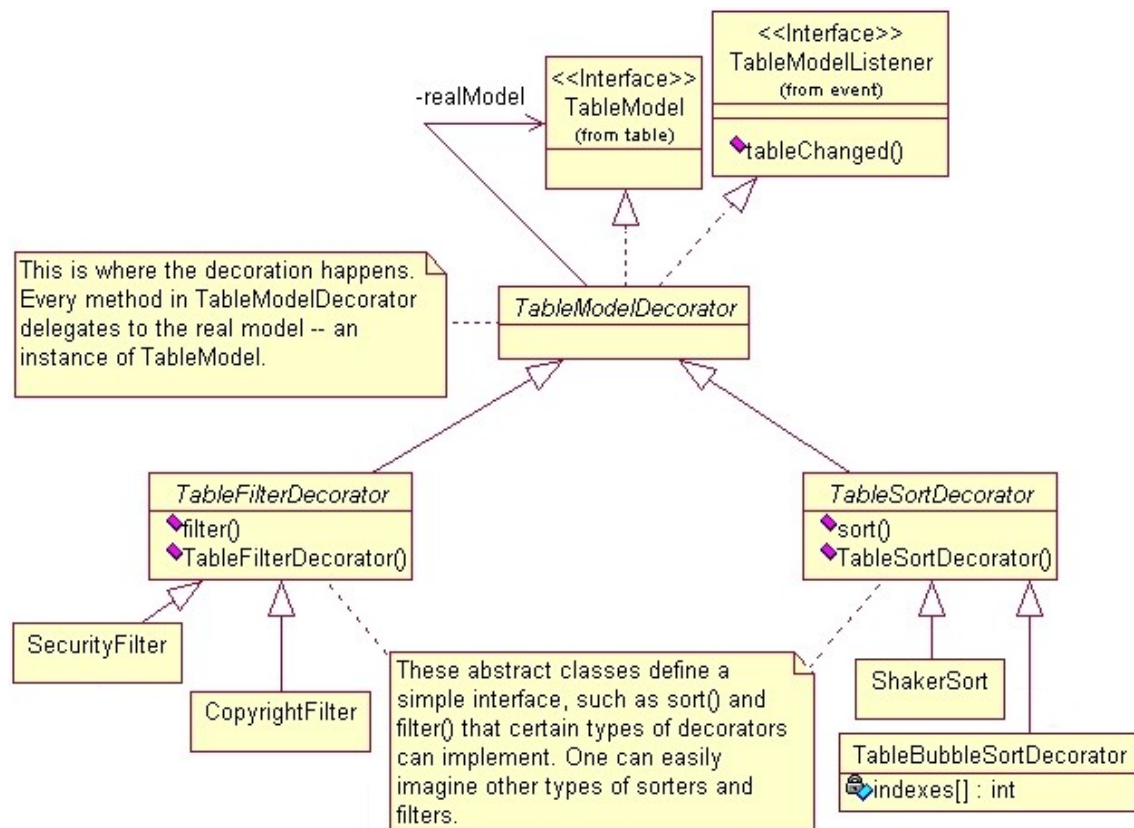
Sin embargo, tal y como se ha implementado el `TableSortDecorator` no nos facilita su reutilización en un entorno más complejo, en el que necesitemos una jerarquía de decoradores, donde cada subclase encapsule, por ejemplo, un método de ordenación distinto. La causa de esto es que nuestro decorador implementa dos responsabilidades para las que no se acaba de ajustar bien del todo.

La primera responsabilidad es la delegación que hace en el comportamiento del modelo real. Se delega en el modelo real porque la funcionalidad que nos ofrece es esencial para la resolución del problema. Si la aplicación crece, necesitaremos que otros decoradores de tablas tengan un código muy parecido (casi exacto), pues se trata de una responsabilidad que es demasiado general; por esto mismo, habría que moverla hacia arriba en la jerarquía.

La segunda responsabilidad es la ordenación, que en el ejemplo se lleva a cabo mediante el algoritmo de la burbuja. El algoritmo está codificado directamente en la clase `TableSortDecorator`, lo que impide cambiarlo (supongamos un algoritmo `Shaker Sort*`) sin tener que reescribir la clase por completo. Dado que el algoritmo de ordenación es muy específico, lo mejor sería bajarlo a una subclase en la jerarquía.

*Shaker Sort** es también, y mejor conocido, como *Cocktail Sort* y en castellano como *Burbuja Didireccional*.

La siguiente figura muestra el resultado de refactorizar el diseño de nuestro programa según lo comentado:



La figura anterior muestra un diseño bastante sofisticado, donde además del decorador de ordenación, tenemos un decorador para filtrar datos.

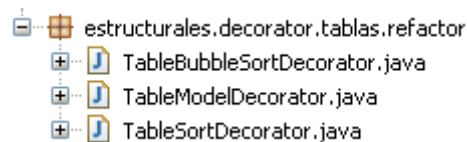
La facilidad con la que ahora podemos crear nuevos decoradores es posible gracias a que hemos encapsulado en la nueva clase TableModelDecorator el código que reenvía las peticiones al modelo real.

La jerarquía del decorador de ordenación

Comenzaremos por centrarnos en el decorador de ordenación, en cuyo caso lo importante es apreciar que la refactorización ha consistido en dividir la clase TableSortDecorator original en tres clases diferentes:

- **TableModelDecorator**: Clase abstracta que implementa **TableModel** con lo que consigue tener la misma interfaz que un modelo de tabla. No obstante, sostiene una referencia de tipo **TableModel** a la que reenviará las peticiones efectuadas por el código cliente. La idea de esta clase es mantener todo el comportamiento que sea común para los decoradores de modelos de tablas (orden, filtrado, etc.)
- **TableSortDecorator**: Extiende **TableModelDecorator** y añade un método abstract para ordenar –el método **sort()**– que las subclases tendrán que implementar.
- **TableBubbleSortDecorator**: Extiende **TableSortDecorator** y sobrescribe el método de ordenación abstracto de **TableSortDecorator** –el método **sort()**– mediante el algoritmo de la burbuja.
- **ShakerSort**: (No implementada). Esta subclase llevaría a cabo una variante del método de la burbuja conocida como burbuja bidireccional.

Antes de mostrar el nuevo código, veamos la disposición de las clases (en un nuevo paquete para no perder la clase anterior):



TableModelDecorator.java

Notad que se trata de una clase no instanciable, concebida para reunir funcionalidad común para sus hijas.

```
package estructurales.decorator.tablas.refactor;

import javax.swing.table.TableModel;
import javax.swing.event.TableModelListener;

/*
 * TableModelDecorator implementa a TableModel y
 * a TableModelListener. TableModelListener es una
 * interfaz que define un metodo muy interesante:
 * tableChanged(), el cual es llamado cuando cambia
 * el modelo de la tabla. tableChanged() no está
 * implementado en la clase abstracta, ya que se
 * deja que las subclases lo implementen.
 */
public abstract class TableModelDecorator
```

```

        implements TableModel, TableModelListener {

// Objeto al que hay que decorar
private TableModel realModel;

/*
 * Este metodo es utilizado por las subclases para
 * acceder al modelo real.
 */
protected TableModel getRealModel() {
    return realModel;
}

/*
 * Constructor.
 * -Recibimos el modelo real y lo asociamos a
 * nuestra referencia.
 * -La clase se suscribe como oyente a los eventos
 * que se produzcan en al cambiar el modelo real.
 */
public TableModelDecorator(TableModel model) {
    this.realModel = model;
    realModel.addTableModelListener(this);
}

/*
 * Los siguientes nueve metodos estan definidos en
 * la interfaz TableModel. Todos son redirigidos
 * hacia el modelo real.
 */

/*
 * El parametro recibido es this. Queremos que la
 * clase actual se registre en los cambios que se
 * produzcan en el modelo real.
 */
@Override
public void addTableModelListener(TableModelListener l) {
    realModel.addTableModelListener(l);
}

/*
 * Retorna la clase de la columna del modelo real
 * recibida por parametro.
 */
@Override
public Class<?> getColumnClass(int columnIndex) {
    return realModel.getColumnClass(columnIndex);
}

/*
 * Retorna el numero de columnas del modelo real.
 */
@Override
public int getColumnCount() {
    return realModel.getColumnCount();
}

/*

```

```

        * Retorna el nombre de la columna real recibida
        * por parametro.
        */
@Override
public String getColumnName(int columnIndex) {
    return realModel.getColumnName(columnIndex);
}

/**
 * Retorna el numero de filas del modelo real.
 */
@Override
public int getRowCount() {
    return realModel.getRowCount();
}

/**
 * Retorna el valor de la celda del modelo real
 * segun los valores de fila y columna pasados
 * por parametro.
 */
@Override
public Object getValueAt(int rowIndex, int columnIndex) {
    return realModel.getValueAt(rowIndex, columnIndex);
}

/**
 * Retorna un boolean indicando si la celda del modelo
 * real es o no editable.
 */
@Override
public boolean isCellEditable(int rowIndex, int columnIndex) {
    return realModel.isCellEditable(rowIndex, columnIndex);
}

/**
 * Desregistra como oyente al objeto recibido por parametro
 * de los eventos producidos al cambiar el modelo real.
 */
@Override
public void removeTableModelListener(TableModelListener l) {
    realModel.removeTableModelListener(l);
}

/**
 * Establece el valor pasado por parametro en la celda del
 * modelo real segun la fila y columna, tambien pasados
 * por parametro.
 */
@Override
public void setValueAt(Object aValue, int rowIndex, int
columnIndex) {
    realModel.setValueAt(aValue, rowIndex, columnIndex);
}
}

```

Hay que apreciar que los métodos públicos de `TableModelDecorator` tan sólo reenvían las peticiones al modelo real, sin añadir nada por su parte. Este es un comportamiento predeterminado, que será heredado, y las subclases lo reescribirán cuando sea necesario.

TableSortDecorator.java

También es una clase abstracta, encargada de definir los métodos abstractos que la diferencian de sus decoradores hermanos. Por ejemplo, dado que este decorador es específico para ordenar, definirá un método `ordenar()` que sus subclases tendrán que implementar, sin embargo no aparecerá nada en absoluto sobre `filtrar()`.

```
package estructurales.decorator.tablas.refactor;

import javax.swing.table.TableModel;

/*
 * Las subclases de TableSortDecorator deben implementar el
 * metodo abstracto ordenar(), además de tableChanged().
 *
 * tableChanged() es necesario porque TableModelDecorator
 * implementa la interfaz TableModelListener.
 */

public abstract class TableSortDecorator extends TableModelDecorator {

    public abstract void ordenar(int column);

    public TableSortDecorator(TableModel realModel) {
        super(realModel);
    }
}
```

TableBubbleSortDecorator.java

```
package estructurales.decorator.tablas.refactor;

import javax.swing.table.TableModel;
import javax.swing.event.TableModelEvent;

/*
 * Esta clase decora el modelo para ofrecer
 * una ordenacion de sus datos mediante una
 * implementación del algoritmo de la burbuja
 */

public class TableBubbleSortDecorator
    extends TableSortDecorator {
```

```

/*
 * Array de enteros que permite al decorador controlar
 * la ordenación del modelo real sin alterarlo.
 * Cada posición del array se corresponde con una fila
 * de la tabla.
 */
private int indices[];

/*
 * Constructor.
 * Recibe el modelo real como parametro.
 * Notad que se la pasa a su superclase
 */
public TableBubbleSortDecorator(TableModel model) {
    super(model);
    inicializarIndices();
}

/*
 * Cuando el modelo cambia se debe llamar al metodo que
 * inicializa el array de indices.
 */
@Override
public void tableChanged(TableModelEvent e) {
    inicializarIndices();
}

/*
 * Retorna el valor de la celda indicada por los
 * indices de los parametros.
 */
@Override
public Object getValueAt(int row, int column) {
    return getRealModel().getValueAt(indices[row], column);
}

/*
 * Establece el valor recibido como primer parametro
 * en la celda indicada por el resto de parametros.
 */
@Override
public void setValueAt(Object aValue, int row, int column) {
    getRealModel().setValueAt(aValue, indices[row], column);
}

/* *****
 * Los siguientes metodos sirven para ordenar mediante el
 * algoritmo de la burbuja.
 * *****/

/*
 * Este método es invocado desde el código cliente al hacer
 * clic sobre alguna columna de la tabla.
 *
 * Parametros:
 *     column: la columna a ordear (la fruta o el precio)
 *
 * Utiliza dos bucles para recorrer las celdas de la columna

```



```

    * correspondiente y comparar sus valores. Cuando encuentra
    * que debe ordenar, llama al método swap para intercambiar
    * las posiciones del array indices.
    */
    public void ordenar(int column) {
        int rowCount = getRowCount();

        for (int i = 0; i < rowCount; i++) {
            for (int j = i + 1; j < rowCount; j++) {
                if (compare(indices[i], indices[j], column) <
0) {
                    swap(i, j);
                }
            }
        }
    }

    private void swap(int i, int j) {
        int tmp = indices[i];
        indices[i] = indices[j];
        indices[j] = tmp;
    }

    /**
     * Para una columna dada, compara los valores de dos filas
     * del modelo real.
     * Parametros:
     *     i: una fila
     *     j: otra fila
     *     column: la columna (fruta o precio)
     * El método compareTo() devolverá:
     *     -1 si valorFila_i < valorFila_j
     *     0 si valorFila_i = valorFila_j
     *     1 si valorFila_i > valorFila_j
     */
    private int compare(int i, int j, int column) {
        TableModel realModel = getRealModel();
        Object valorFila_i = realModel.getValueAt(i, column);
        Object valorFila_j = realModel.getValueAt(j, column);
        int resultado = valorFila_j.toString()
            .compareTo(valorFila_i.toString());
        return (resultado < 0) ? -1 : ((resultado > 0) ? 1 : 0);
    }

    private void inicializarIndices() {
        indices = new int[getRowCount()];
        for (int i = 0; i < indices.length; ++i) {
            indices[i] = i;
        }
    }
}

```

Las clases clientes serán las mismas, con alguna mínima modificación:

MainClient.java

```
...

import
estructurales.decorator.tablas.refactor.TableBubbleSortDecorator;
//import estructurales.decorator.tablas.TableSortDecorator;
import estructurales.decorator.tablas.refactor.TableSortDecorator;

...

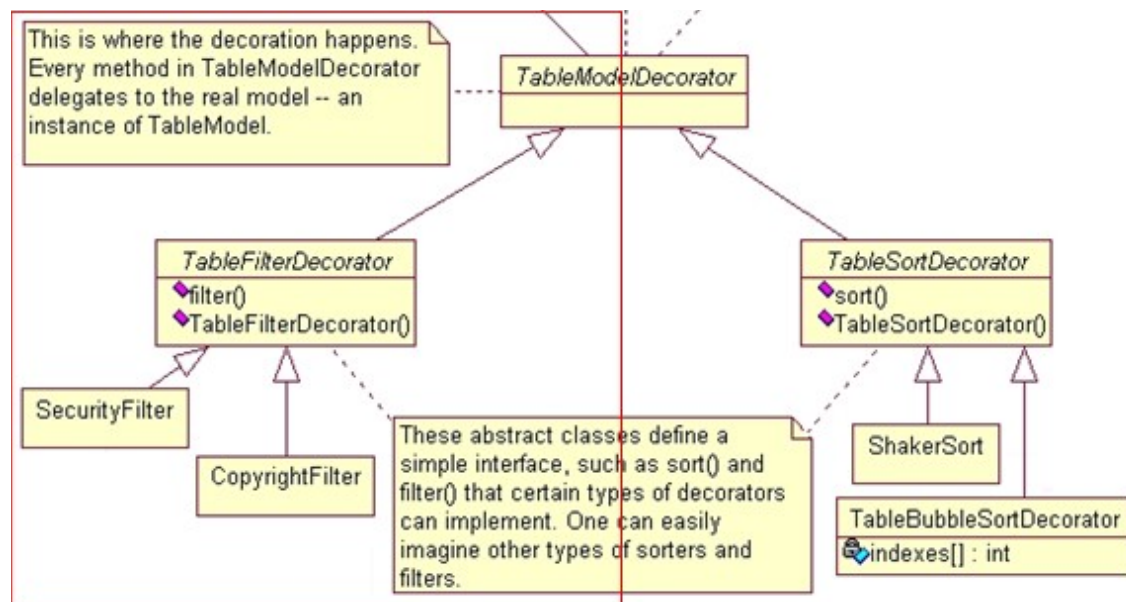
/*
 * Constructor.
 */
public MainClient() {
    // Crea el decorador que decorará el modelo real de la tabla.
    // Notad cómo se construye a partir del modelo real (cómo lo
    // envuelve).

    final TableSortDecorator decorador =
        new TableBubbleSortDecorator(table.getModel());
    /*final TableSortDecorator decorador =
        new TableSortDecorator(table.getModel());*/
}

...
```

La jerarquía del decorador de filtrado

Ahora es el turno de centrarnos en la parte del diseño que se encarga de decorar el modelo real filtrando datos.



Vamos a ver cómo podemos dotar a la aplicación de un decorador que filtre las filas cuyo precio sobrepasen un precio determinado por las clases cliente. Por ejemplo, según nuestros datos podríamos hacer que no se muestren aquellas frutas cuyo precio sea mayor a 2 euros.

Pasos a seguir:

- Creamos la clase `TableFilterDecorator`, que extienda a `TableModelDecorator` y que defina un método abstracto de filtrado que sus subclasses tendrán que implementar.
- Subclasses que implementan diversos filtros. Para nuestro ejemplo es suficiente con crear una única clase, que llamaremos `SecurityFilter`. Esta clase extiende a `TableFilterDecorator` y será la responsable de implementar el filtrado de las filas de las tablas que no se deban visualizar. Para ello implementará el método de filtrado y sobrescribirá cualquier método de sus antecesoras cuando sea necesario.
- Modificamos el código cliente para que se utilice el decorador de filtrado.

Comenzamos con la clase abstracta. Lo más notable es observar que define el método de filtrado como abstracto:

TableFilterDecorator.java

```
package estructurales.decorator.tablas.refactor;

import javax.swing.table.TableModel;

/*
 * Las subclasses de TableFilterDecorator deben implementar el
 * metodo abstracto filtrar(), además de tableChanged().
 *
 * tableChanged() es necesario porque TableModelDecorator
 * implementa la interfaz TableModelListener.
 */
public abstract class TableFilterDecorator
    extends TableModelDecorator {

    public abstract void filtrar(int fila, boolean estado);

    public TableFilterDecorator(TableModel model) {
        super(model);
    }
}
```

Ahora la clase que oculta las filas que sobrepasen el precio establecido por el código cliente.

SecurityFilter.java

```
package estructurales.decorator.tablas.refactor;

import javax.swing.event.TableModelEvent;
import javax.swing.table.TableModel;

public class SecurityFilter extends TableFilterDecorator {

    // Array donde cada posición indica si
    private boolean[] arrayFilasOcultas;
    private int numFilasVisibles;

    public SecurityFilter(TableModel model) {
        super(model);
        inicializar();
    }

    /**
     * El código cliente llama a este método cuando necesita que se
    oculte o se
     * vuelva a mostrar alguna fila.
     */
    @Override
    public void filtrar(int fila, boolean estado) {
        arrayFilasOcultas[fila] = !estado;
        if (estado)
            numFilasVisibles++;
        else
            numFilasVisibles--;
    }

    /**
     * Método que se ejecuta al construir el objeto y cuando cambia
    el modelo
     */
    private void inicializar() {

        // En un principio, todas las filas son visibles
        numFilasVisibles = getRealModel().getRowCount();

        // Pueden haber tantas filas ocultas como filas existan
        arrayFilasOcultas = new boolean[numFilasVisibles];
    }

    /**
     * Retornar el número de filas visibles
     */
    @Override
    public int getRowCount() {
        return numFilasVisibles;
    }

    /**
```

```

        * Devolver la correspondiente fila no oculta
        */
@Override
public Object getValueAt(int row, int col) {
    // BUCLE INFINITO
    for (int realRow = 0; true; realRow++) {
        if (!arrayFilasOcultas[realRow]) {
            if (row == 0) {
                return getRealModel().getValueAt(realRow,
col);
            }
            row--;
        }
    }
}

@Override
public void tableChanged(TableModelEvent e) {
    inicializar();
}
}

```

La verdad es el que el código de esta clase es un poco enrevesado. No obstante, para la comprensión del patrón Decorador lo importante es entender cuál es la finalidad de esta clase y no tanto cómo se lleva a cabo una determinada funcionalidad en particular.

Por último nos resta por ver el código cliente. Se muestra sólo en color amarillo el nuevo código.

MainClient.java

```

...
/*
 * Constructor.
 */
public MainClient() {
    // Crea el decorador que decorará el modelo real de la tabla.
    // Notad cómo se construye a partir del modelo real.

    final TableSortDecorator decorador =
        new TableBubbleSortDecorator(table.getModel());
    /*final TableSortDecorator decorador =
        new TableSortDecorator(table.getModel());*/

    /*
     * Establecer el decorador como el modelo de la tabla. Esto
     * es posible porque el decorador implementa TableModel.
     */
    table.setModel(decorador);
}
/*

```

```

    * Creamos el decorador de filtrado.
    * Notad que se crea encapsulando al decorador de ordenación.
    */
    TableFilterDecorator decoradorFiltroPrecio =
        new SecurityFilter(decorador);

    /*
    * Informamos al decorador de filtrado qué filas debe
    ocultar.
    * Recorremos el modelo real, obtenemos el valor de la
    columna precio.
    * Como el valor contiene el símbolo del euro al final, hay
    que quitarlo
    * y convertirlo a float. A continuación, hay que
    compararlo con 2.0, que
    * es el precio a partir del cual una fila debe quedar
    oculta.
    */
    final int COL_PRECIO = 1;
    for (int i=0; i<table.getModel().getRowCount(); i++) {
        String numero_cadena = table.getModel()
            .getValueAt(i, COL_PRECIO).toString();
        String numero = numero_cadena
            .substring(0, numero_cadena.length()-1);
        float valor_celda = Float.parseFloat(numero);
        if (valor_celda >= 2.0)
            decoradorFiltroPrecio.filtrar(i, false);
    }

    // Fijar el decorador de filtrado como el modelo para la tabla
    table.setModel(decoradorFiltroPrecio);

    // Hacer que la tabla quede dentro de un panel con scroll.
    getContentPane().add(

```

...

Del código anterior, tened presente que sólo funcionará la conversión de String a Float para el caso en que el símbolo de la moneda ocupe la última posición del precio. Esto habría que implementarlo mejor.

Salida (notad que no aparecen las filas cuyos productos superaban los 1.99€):

| Decorator que ordena datos | |
|---------------------------------|------------|
| Producto | Precio/Kg. |
| manzana | 1.39€ |
| limon | .89€ |
| naranja | 1.59€ |
| sandía | .79€ |
| mandarina | 1.09€ |
| cereza | 1.49€ |
| lima | 1.29€ |
| pomelo | 1.33€ |
| Inicialmente, datos sin ordenar | |

Problemas específicos e implementación

Identidad

Es importante tener presente que un decorador y la referencia al componente que sostiene (el atributo en el que delega la funcionalidad subyacente) no son lo mismo. El decorador actúa como un envoltorio transparente, pero desde el punto de vista de la identidad de objetos, son tipos diferentes.

Muchos objetos, pequeños y similares

Es habitual que un diseño basado en el patrón Decorador resulte en un sistema compuesto por un numeroso conjunto de pequeños objetos bastante parecidos. Tales objetos diferirán sólo en la manera en que están interconectados, y no en sus clases o en el valor de sus variables. Aunque este tipo de sistemas es sencillo de configurar y mantener por quienes están familiarizados con ellos, suele ser un tanto enrevesado de aprender y depurar. Como ejemplo de esto, pensad en el caso de los streams de entrada y salida del API Java, donde se tienen un número considerable de clases, todas bastante parecidas y que pueden presentarse interconectadas de múltiples formas.

Varios a temas considerar

Cuando se aplica el patrón Decorador es necesario tener en cuenta una serie de aspectos:

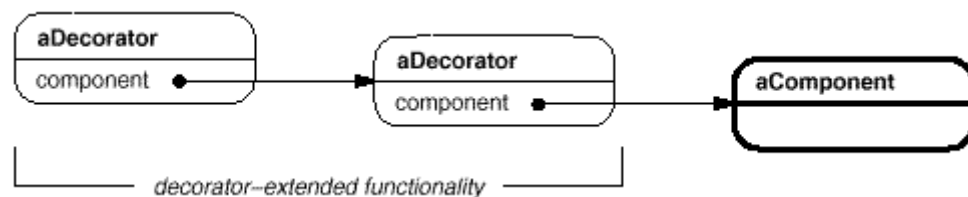
Conformidad a una interfaz: La interfaz de un decorador debe corresponderse a la del componente que decora. Por tanto, las clases ConcreteDecorator deben heredar de una clase común o implementar una misma interfaz.

Omisión de una clase Decorator abstracta: No hay necesidad de definir una clase abstracta Decorator cuando solamente necesitamos añadir una responsabilidad.

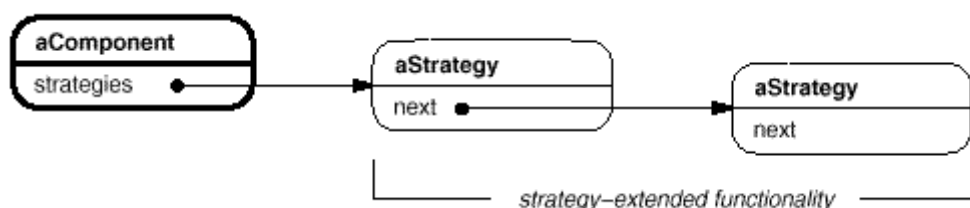
Mantener ligera la clase Component (cuando no es una interfaz): Para asegurar que tienen una misma interfaz, tanto componentes como los decoradores deben descender de una misma clase Component. Por tanto, es importante conservar la clase Component lo más ligera posible, es decir, debe limitarse a definir una interfaz y no a almacenar datos. La implementación debe diferirse a las subclases. En caso contrario, la complejidad de la clase Componente podría resultar en decoradores muy pesados,

de pobre calidad y que no fuesen prácticos de utilizar. Hacer que la clase Component tenga mucha funcionalidad también incrementa la probabilidad de que las subclases concretas estén penalizadas con una funcionalidad que no necesitan.

Cambiar lo superficial de un componente versus cambiar una parte esencial: Tenemos que pensar en un decorador como un envoltorio capaz de modificar el comportamiento de un objeto. No obstante, esa modificación es superficial y transparente para el objeto decorado. Notad como en la siguiente figura el componente nunca es consciente de que es decorado:



En casos en los que se necesite cambiar la parte esencial de un componente, una buena opción es utilizar el patrón Estrategia (Strategy). Este patrón es especialmente interesante cuando la clase Component es intrínsecamente compleja/pesada, por lo que el patrón Decorador resultaría muy costoso de aplicar. Con el patrón Estrategia el componente traspasa parte de su comportamiento a un objeto estrategia. Con esto se consigue alterar y extender el comportamiento de un componente cambiando de objetos estrategia. Por tanto, mediante el patrón Estrategia los propios componentes tienen constancia de que pueden ser extendidos de alguna forma y por ello sostienen referencias a otros objetos estrategia:



Patrones relacionados

Adapter: un decorador es diferente de un adaptador en que el decorador sólo cambia las responsabilidades de un objeto, no su interfaz; un adaptador le dará a un objeto una nueva interfaz.

Composite: Un decorador puede verse como una composición que contiene un único componente. No obstante, un decorador se limita a añadir responsabilidades a un objeto; no está concebido para agregar objetos.

Strategy: Un decorador permite cambiar algún aspecto superficial de un objeto, mientras que una estrategia modifica una parte esencial del objeto. Estas son dos formas alternativas de modificar objetos.