

## **Patrón decorador**

### **Objetivo**

- Proporcionar una práctica que permita reforzar los conceptos vistos.
- Detallar el modelamiento UML realizado para el patrón.

### **Motivación**

Añade funcionalidad adicional a un objeto dinámicamente. Proporciona una alternativa para sub. Clasificar por extensión de funcionalidad.

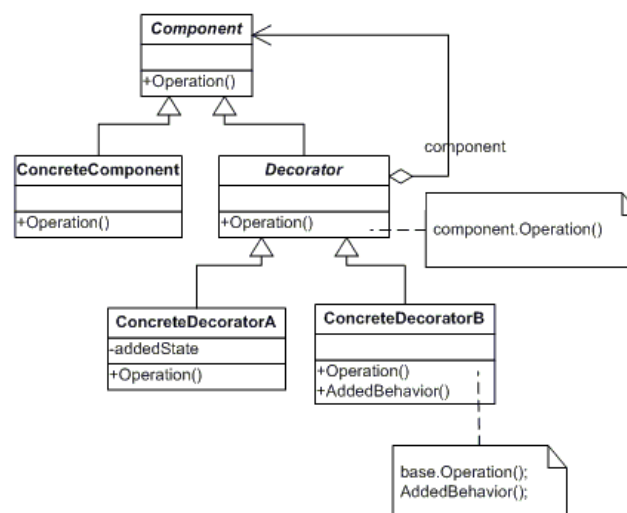
### **Introducción**

El patrón Decorador es usado para extender la funcionalidad de un objeto dinámicamente sin tener que cambiar la clase origen de este objeto o aplicar la herencia. El mecanismo general de funcionamiento del patrón es crear un wrapper (envoltorio) que normalmente conocemos como el Decorador.

### **Características de un decorador**

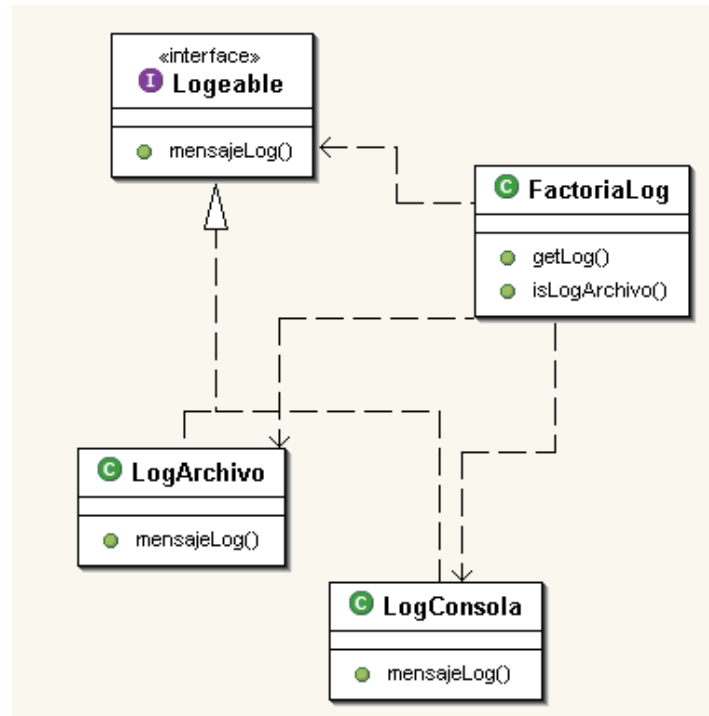
1. El objeto decorador es diseñado para tener la misma interfase que el objeto al cual decora. Esto le permite a un objeto cliente, interactuar de la misma forma en la cual lo realizaba con el objeto original.
2. El decorador debe contener una referencia al objeto decorado.
3. El objeto decorador recibe todas las llamadas desde el cliente, las procesa y luego puede pasarlas al objeto decorado.
4. El objeto decorador normalmente puede adicionar funcionalidad en tiempo de ejecución.

En forma típica, un decorador reemplaza el mecanismo de herencia, teniendo como limite la complejidad de la funcionalidad que se adiciona.



## Escenario

Existe una pequeña aplicación que escribe mensajes de Log, bien sea a un archivo o directamente por consola, se requiere mejorar la aplicación para permitir que los mensajes sean formateados en HTML y posiblemente Cifrados mediante un algoritmo básico de corrimiento de caracteres.



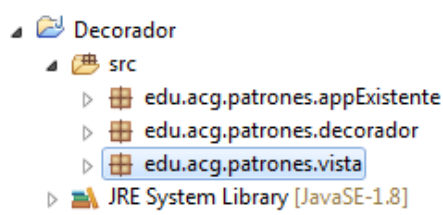
Aplicación de Log simple

Una alternativa para resolver el problema sería heredar de LogArchivo los dos tipos de formato con dos clases como LogArchivoHTML y LogArchivoCifrado, e igualmente podría darse el mismo caso para LogConsola. (Aunque no tendría un sentido práctico en la vida real). Pero lo que finalmente es importante notar es que se requerirían 4 clases, para dotar a esta aplicación de las nuevas características propuestas.

El patrón Decorador puede servir en casos como el que estamos considerando. Particularmente, nos recomienda crear un objeto envoltorio alrededor del objeto actual para extender su funcionalidad mediante composición y no por herencia.

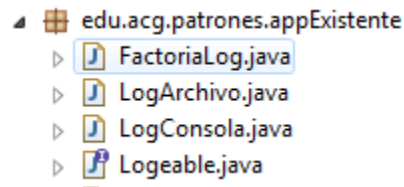
### Tarea 1: Diseño de la aplicación

1. Cree un nuevo proyecto java en Eclipse llamado **Decorador** y dentro cree tres package denominados:
  - edu.acg.patrones.appExistente.
  - edu.acg.patrones.decorador
  - edu.acg.patrones.vista



2. En el paquete denominado edu.acg.patrones.appExistente agregar las siguientes clases.

- FactorialLog
- LogArchivo
- LogConsola
- Logeable



Cada una con el siguiente código:

### 2.1. clase **FactorialLog**

```
package edu.acg.patrones.appExistente;
public class FactorialLog {
    private String VariableLog;
    public FactorialLog(String VariableLog){
        this.VariableLog=VariableLog;
    }
    public boolean isLogArchivo() {
        try {
            if (VariableLog.equalsIgnoreCase("ON") == true)
                return true;
            else
                return false;
        } catch (Exception e) {
            return false;
        }
    }
    //Metodo de Factoria
    public Logeable getLog() {
        if (isLogArchivo()) {
            return new LogArchivo();
        } else {
            return new LogConsola();
        }
    }
    public String getVariableLog() {
        return VariableLog;
    }
    public void setVariableLog(String variableLog) {
        VariableLog = variableLog;
    }
}
```

## 2.2. clase LogArchivo

```
package edu.acg.patrones.appExistente;
import java.io.FileWriter;
import java.io.PrintWriter;
public class LogArchivo implements Logeable {
    FileWriter fichero = null;
    PrintWriter pw = null;
    public String mensajeLog(String msg) {
        StringBuffer mensaje= new StringBuffer();
        mensaje.append("Archivo ");
        try
        {
            fichero = new FileWriter("D:/Log.txt");
            pw = new PrintWriter(fichero);
            pw.println(msg);
            mensaje.append("Generado");
        } catch (Exception e) {
            mensaje.append("No generado");
            e.printStackTrace();
        } finally {
            try {
                // Nuevamente aprovechamos el finally para
                // asegurarnos que se cierra el fichero.
                if (null != fichero)
                    fichero.close();
            } catch (Exception e2) {
                e2.printStackTrace();
                mensaje.append("No generado");
            }
        }
        return mensaje.toString();
    }
}
```



## 2.3. clase LogConsola

```
package edu.acg.patrones.appExistente;
public class LogConsola implements Logeable {
    public String mensajeLog(String msg) {
        return msg;
    }
}
```

## 2.4. clase Logeable

```
package edu.acg.patrones.appExistente;
public interface Logeable {
    public String mensajeLog(String msg);
}
```

3. En el paquete denominado edu.acg.patrones.vista agregar la siguiente clase.
  - VistaExistente

▲  edu.acg.patrones.vista  
▶  VistaExistente.java

Con el siguiente código:

```
package edu.acg.patrones.vista;
import edu.acg.patrones.appExistente.FactoriaLog;
import edu.acg.patrones.appExistente.Logeable;
public class VistaExistente {
    public static void main(String[] args) {
        FactoriaLog factoria = new FactoriaLog("ON");
        Logeable obj_TipoLog = factoria.getLog();
        System.out.println(obj_TipoLog.mensajeLog("Un mensaje de
prueba enviado Log..."));
    }
} // Fin de la clase
```

4. Comprobar el correcto funcionamiento de la aplicación existente:

ON= Escritura del mensaje en el archivo "Log.txt".

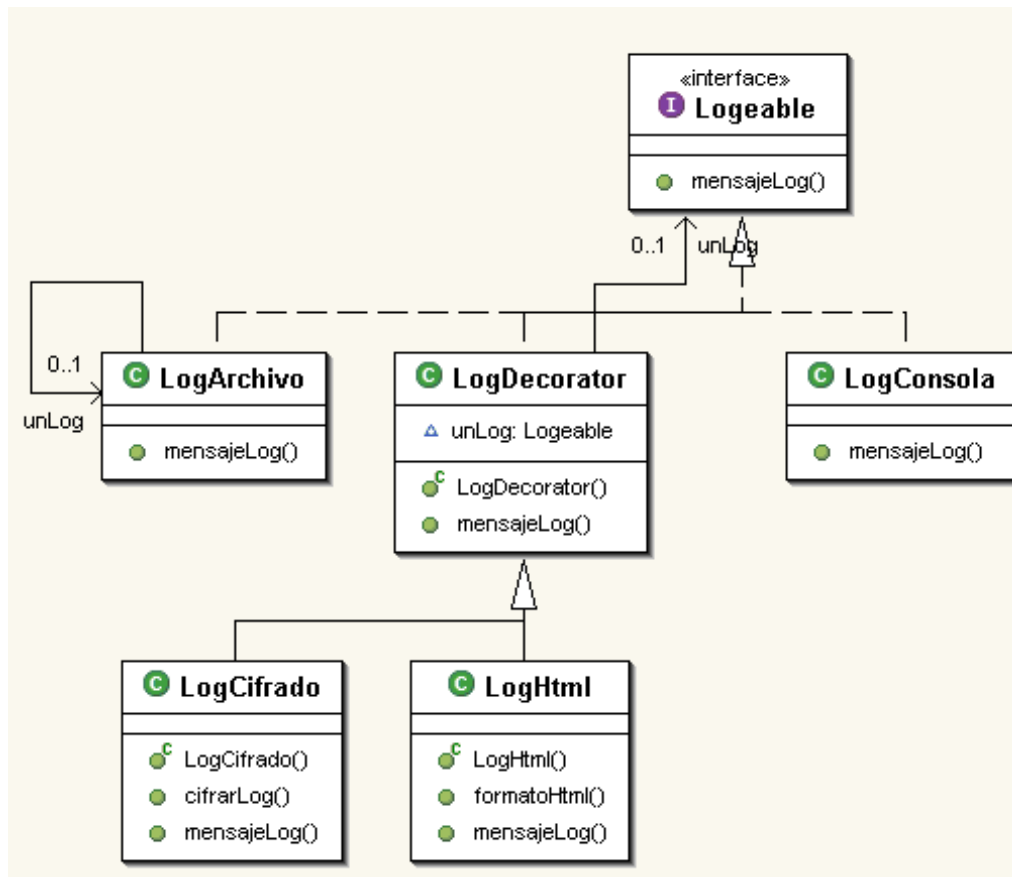
OFF=Mensaje imprimido en consola.

### Decorador

5. Aplicando el patrón Decorador, definiremos un Decorador genérico denominado LogDecorator el cual contiene una referencia a la Interfaz Logeable. Esta referencia tiene como objetivo apuntar al objeto que se esta Decorando. (Para este caso seria LogConsola o LogArchivo). Igualmente, LogDecorator implementa la interfase Logeable y proporciona una implementación básica para el método mensajeLog(), la cual en este caso toma un mensaje y lo redirige hacia el objeto que envuelve.

Finalmente, es importante decir que cada subclase de LogDecorator implementa a su manera el comportamiento definido en mensajeLog para mejorar o decorar su funcionalidad.

En la siguiente figura se muestra el diagrama de clases del ejemplo:



**Ejemplo de aplicación de Decorador**

6. Ahora crearemos los decoradores concretos.

Para el ejemplo, se tienen dos tipos de decoradores, Un Decorador denominado LogCifrado y un decorador LogHtml. LogHtml redefine la implementación del método mensajeLog() para transformar este mensaje en una salida de formato Html.

El decorador LogCifrado a su vez, aplicara al mensaje enviado a uno de los objetos un cifrado simple por desplazamiento de un carácter a la derecha. (Es un cifrado de ejemplo).

Para que este nuevo diseño funcione se requiere entonces modificar la aplicación existente básicamente en las siguientes acciones:

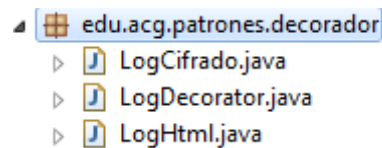
1. Crear una instancia apropiada de la Interfaz Logeable (Metodo Factoria)
2. Crear una instancia del decorador que se desee aplicar (Puede ser parametrizado)
3. Invocar los métodos sobre la instancia del decorador que a su vez van a ser redirigidos a la instancia del objeto envuelto (LogConsola, LogArchivo).

## Tarea 2: Listados de código Decorador

A continuación se detallan los archivos y los cambios a introducir para aplicar el patrón Decorador.

1. En el paquete denominado edu.acg.patrones.decorador agregar las siguientes clases (Propias del patrón decorador).

- LogCifrado
- LogDecorator
- LogHtml



Cada una con el siguiente código:

### 1.1. Clase LogCifrado

```
package edu.acg.patrones.decorador;
import edu.acg.patrones.appExistente.Logeable;
public class LogCifrado extends LogDecorator {
    public LogCifrado(Logeable objLog) {
        /*Recibir por parametro el objeto inicialmente
        instanciado*/
        super(objLog);
    }
    public String mensajeLog(String msgLog) {
        /*Adicionar funcionalidad */
        msgLog = cifrarLog(msgLog);
        /*Enviar el objeto decorado para escribir en el medio
        adecuado */
        return msgLog;
    }
    public String cifrarLog(String msgLog) {
        /*Aplicar un cifrado muy simple, correr los caracteres una
        posición */
        msgLog = msgLog.substring(msgLog.length() - 1) +
            msgLog.substring(0, msgLog.length() - 1);
        return msgLog;
    }
}
// fin de la clase
```

### 1.2. Clase LogHtml

```
package edu.acg.patrones.decorador;
import edu.acg.patrones.appExistente.Logeable;
public class LogHtml extends LogDecorator {
    public LogHtml(Logeable objLogeable) {
        super(objLogeable);
    }
    public String mensajeLog(String msgLog) {
        /*Funcionalidad adicionada por el decorador */
```

```

        msgLog = formatoHtml(msgLog);
        /*Ahora se envia el mensaje al tipo de instancia que se
selecciono */
        return msgLog;
    }
    public String formatoHtml(String msgLog) {
        /*Convertir el log en un formato diferente (HTML). */
        msgLog = "<HTML><HEAD></HEAD><BODY>" + "<b>" + msgLog
+ "</b>" + "</BODY></HTML>";
        return msgLog;
    }
} // fin de la clase

```

### 1.3. Clase LogDecorator

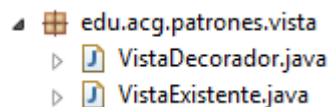
```

package edu.acg.patrones.decorador;
import edu.acg.patrones.appExistente.Logeable;
public class LogDecorator implements Logeable {
    // Se crea una referencia a la interface superior
    Logeable unLog;
    public LogDecorator(Logeable obj_Log) {
        // Se asocia la referencia generica al objeto recibido
        unLog = obj_Log;
    }
    public String mensajeLog(String msgLog) {
        /*Implementacion por defecto que sera redefinida por las
subclases*/
        return unLog.mensajeLog(msgLog);
    }
} // fin de la clase

```

- En el paquete denominado edu.acg.patrones.vista agregar la siguiente clase (Propias del patrón decorador).

- VistaDecorador



Con el siguiente código:

```

package edu.acg.patrones.vista;
import edu.acg.patrones.appExistente.FactorialLog;
import edu.acg.patrones.appExistente.Logeable;
import edu.acg.patrones.decorador.LogCifrado;
import edu.acg.patrones.decorador.LogDecorator;
import edu.acg.patrones.decorador.LogHtml;
public class VistaDecorador {
    public static void main(String[] args) {
        FactorialLog factoria = new FactorialLog("ON");
        Logeable obj_TipoLog = factoria.getLog();
        //Utilizando el objeto existente.
        LogDecorator decoradoImpl= new
LogDecorator(obj_TipoLog);
        System.out.println(decoradoImpl.mensajeLog("Un mensaje
de prueba enviado al Log...."));
    }
}

```



```

        //El objeto decorador proporciona la misma interface
        que los objetos iniciales.
        LogHtml obj_Html = new LogHtml(obj_TipoLog);
        System.out.println(obj_Html.mensajeLog("Un mensaje de
prueba enviado al Log....debe estar en HTML"));
        //El objeto decorador proporciona la misma interface
        que los objetos iniciales.
        LogCifrado obj_Cifrado = new LogCifrado(obj_Html);
        System.out.println(obj_Cifrado.mensajeLog("Un mensaje
de prueba enviado al Log....debe estar cifrado"));
        //Usando en conjunto ambas funcionalidades

        System.out.println(obj_Html.mensajeLog(obj_Cifrado.mensajeLog("U
n mensaje de prueba enviado al Log....debe estar cifrado"))));
    }
}

```

3. Ejecute la aplicación en eclipse y compruebe el correcto funcionamiento del patrón y aplicación.

### **EJERCICIO PROPUESTO**

Amplíe la funcionalidad proporcionada por los dos decoradores mediante un tercer decorador que permita formatear el mensaje en XML y escribirlo tanto en consola como en un archivo.

**Fin de laboratorio de patrón decorador.**