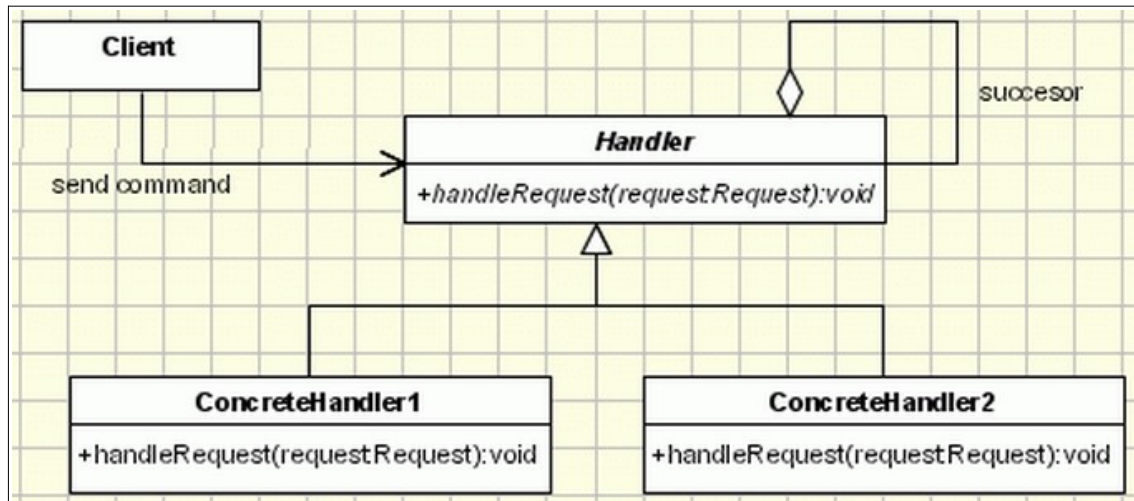


Chain of Responsibility

Diagrama de clases e interfaces

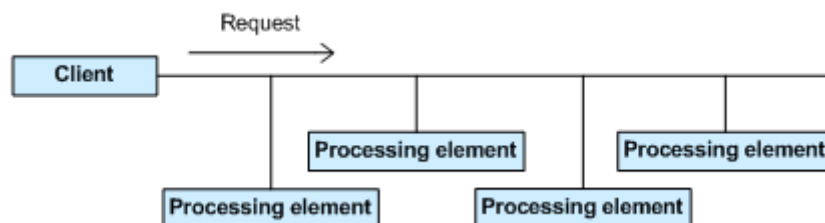


Intención

Chain of Responsibility (CoR) permite establecer dinámicamente una cadena de objetos a través de los cuales se pasa una petición formulada por un objeto emisor. Cualquiera de los objetos receptores puede responder a la petición en función de un criterio establecido.

Motivación

Este patrón permite que un objeto emisor envíe una petición (una solicitud, un mensaje,...) a una serie de objetos receptores que se encadenan secuencialmente (como una lista enlazada), siguiendo un orden determinado.



De esta manera la petición llega al primer objeto en la secuencia, el cual intentará despacharla, o en caso de no poder resolverla, la pasará al siguiente receptor.

Ni el objeto emisor -que inicia la solicitud- ni ninguno de los objetos receptores conoce las características del resto de objetos receptores. Por tanto, nadie sabe nunca qué receptor satisfará la petición en curso.

Además, el patrón no garantiza que las peticiones se despachen exitosamente. Esto quiere decir que puede ocurrir que la petición alcance el final de la cadena y no haya sido procesada convenientemente.

El siguiente código es un ejemplo de una clase cliente (el emisor) que envía una petición a una serie de objetos receptores para su resolución. El ejemplo asume un diseño sin patrón CoR, por lo que el emisor debe adquirir una referencia de todos y cada uno de los objetos manejadores (receptores). Uno a uno, les envía la petición hasta que alguno pueda despacharla:

```
...  
handlers = getHandlers();  
for (int i = 0; i < handlers.length; i++) {  
    handlers[i].handle(request);  
    if (handlers[i].handled()) {  
        break;  
    }  
}
```

En cambio, fíjate cómo difiere el código para una clase cliente donde los objetos receptores implementan el patrón CoR:

```
getChain().handle(request);
```

En este caso podemos asumir que el método `getChain()` retorna una referencia al primer objeto de la cadena, que suele ser lo más habitual. Posteriormente invocamos al método `handle()` del primer objeto de la cadena, pasándole la petición. Esto provoca que los objetos receptores se llamen unos a otros secuencialmente hasta que alguno resuelva la petición (o no, ya que esto es relativo al contexto de la aplicación).

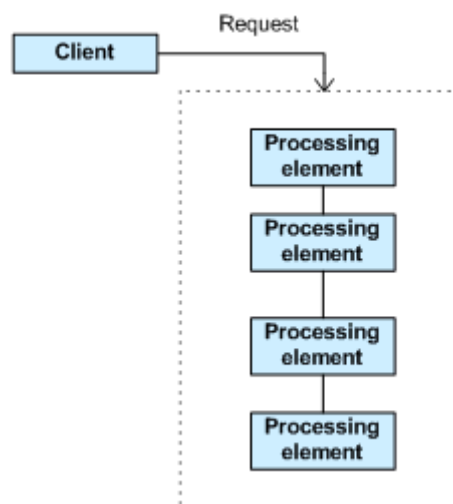
Lo destacable aquí es que el código cliente sólo se acopla al primer eslabón de la cadena y no a toda ella.

Desde el punto de vista del diseño de clases, lo más importante de CoR es que el emisor ve a los objetos receptores como una caja negra, es decir, sólo observa una abstracción a la que se le puede enviar peticiones.

CoR mantiene bajo el acoplamiento del sistema, ya que el emisor tiene suficiente con mantener una referencia al primer receptor de la cadena (al que le enviará la petición) y los objetos receptores tan sólo necesitan sostener una referencia a su siguiente objeto en la cadena.

Lógicamente, para que esto sea posible, los objetos receptores deben presentar una interfaz uniforme, lo cual se resuelve haciendo que implementen una determinada interfaz y/o extiendan una misma clase abstracta.

Recordemos que el bajo acoplamiento siempre es un objetivo a perseguir, ya que al disminuir las relaciones entre objetos se evita que aumente la complejidad de la aplicación, resultando mucho más sencillo su mantenimiento.



No obstante, a pesar de que el emisor puede ignorar los entresijos de la cadena de objetos receptores, el patrón es tan flexible que permite que sea el código cliente quien establezca tanto el número de objetos receptores como la estructura de la cadena. Por tanto, dado que es en tiempo de ejecución cuando se conforma la cadena de receptores (por ejemplo, mediante una clase de servicio ajena al patrón tipo ServicioPeticiones) podemos configurarla de manera diferente para atender distintos tipos de peticiones, según el estado en el que se encuentre el código cliente en cada momento.

Ejemplos de CoR del API de Java

Filtros servlet

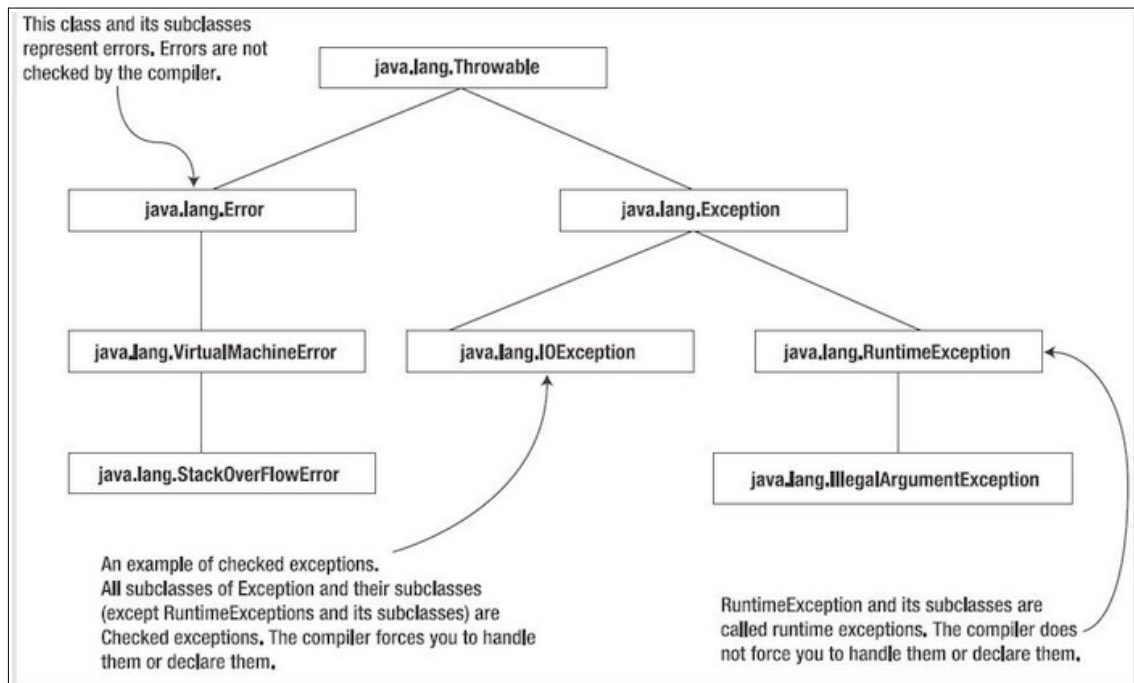
Los filtros servlet son un ejemplo del patrón de diseño Cadena de Responsabilidades. Más adelante veremos un ejemplo (*ejemplo 4*).

Mecanismo de control de excepciones

El sistema de gestión de excepciones de Java es otro ejemplo de diseño CoR. Cuando ocurre un error en tiempo de ejecución se lanza una excepción, la cual recorre la pila de llamadas a métodos en busca de una clase que pueda encargarse de ella.

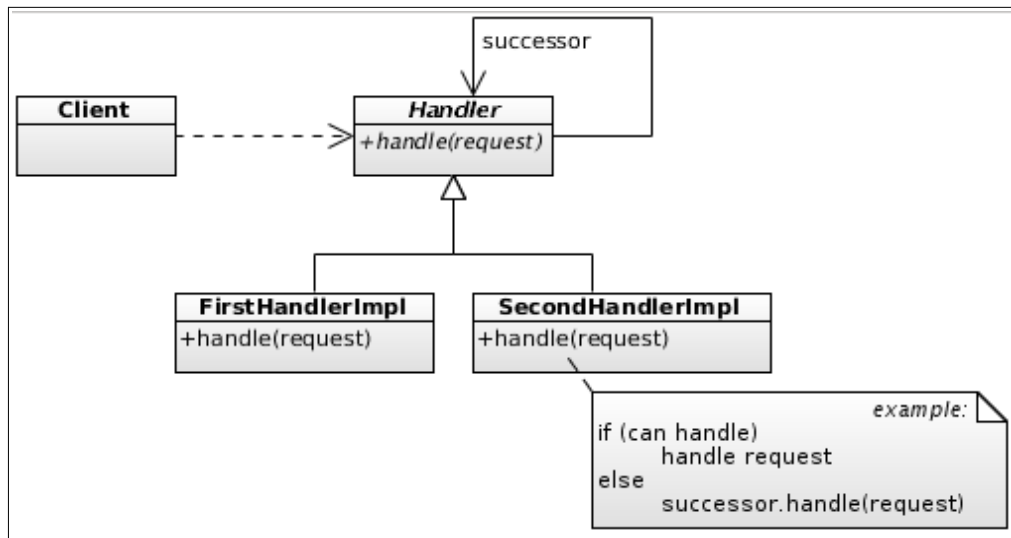
Si no hay ninguna clase apropiada para gestionar el tipo de excepción lanzada, la excepción llega al final de la pila, momento en que el interviene -como último recurso- la súper clase Exception para capturarla.

En otro caso, esto es, si se ha encontrado una clase adecuada para manejar la excepción, entonces ésta es procesada como haya determinado el programador, finalizando la ascensión por la pila de llamadas.



Implementación

El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:

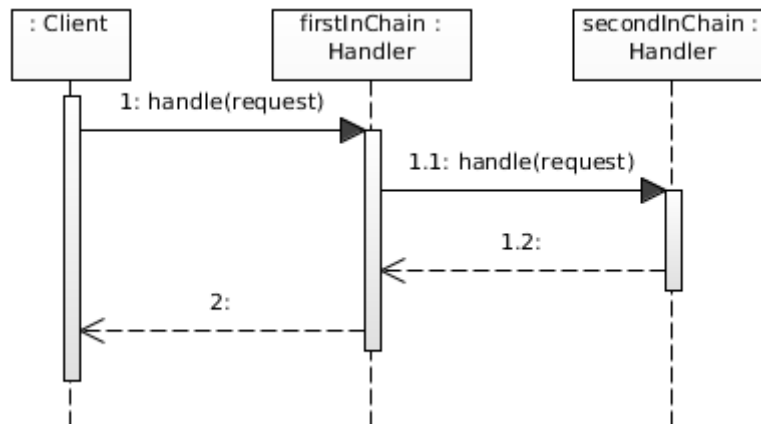


Todos los objetos receptores (**FirstHandlerImpl1**, **FirstHandlerImpl2**,...) implementarán la misma interfaz o extenderán la misma clase abstracta. En ambos casos habrá un método que permita obtener el sucesor en la cadena de receptores, por lo que el paso de la petición por la cadena será lo más flexible y transparente posible.

Las clases participantes en el patrón son las siguientes:

- **Handler:** Normalmente se trata de una clase abstracta que define una interfaz para el tratamiento de las peticiones. Opcionalmente, aunque a menudo, proporciona un método para obtener el objeto sucesor en la cadena. Este método suele ser muy conveniente para sus subclases.
- **ConcreteHandler** (**FirstHandlerImpl1**, **FirstHandlerImpl2**,...): Clase concreta con la implementación necesaria para despachar un determinado tipo de solicitud o una solicitud cuyo estado sea el adecuado para esta clase concreta. Puede sobrescribir la implementación proporcionada por la superclase para la obtención del siguiente objeto en la cadena. Si **ConcreteHandlerX** puede manejar la petición que le llega, lo hace; en caso contrario la reenvía a su sucesor.
- **Client:** Envía peticiones al primer objeto de la cadena (única referencia que conoce).

La siguiente figura muestra el diagrama de secuencias para el patrón CoR:



Aplicabilidad y Ejemplos

El patrón CoR es adecuado cuando:

- Una petición puede ser gestionada por más de un objeto receptor.
- Se desconoce de antemano el objeto receptor adecuado para resolver una petición, por lo que interesa que el objeto manejador se escoja de manera automática, sin que tenga que explicitarlo el objeto emisor.
- Se quiere poder definir dinámicamente (en tiempo de ejecución) el grupo de objetos receptores que formarán la cadena, así como su orden en ella.

Veamos ahora algunos ejemplos.

Ejemplo 1 – Cadena de manejadores de números enteros según su signo

Mediante este sencillo ejemplo podremos ver un caso concreto de implementación de este patrón. Se trata de una clase cliente que genera números enteros aleatorios comprendidos entre el -9 y el 9, ambos incluidos, y los envía a la cadena de objetos manejadores. La cadena está formada por tres clases: una que atiende a números negativos, otra a positivos y la última al número cero. Estas clases se limitan a mostrar el número capturado por pantalla.

Comenzamos por la clase Request, la cual se limita a encapsular un número entero.

Request.java

```
package comportamiento.CoR.basico;

public class Request {

    private int valor;

    public Request(int valor) {

        this.valor = valor;

    }

    public int getValor() { return valor; }

}
```

A continuación, veamos la superclase abstracta que define la interfaz pública para los manejadores de peticiones de nuestro ejemplo. Notad que se define:

- Un atributo denominado 'sucesor' cuyo tipo es el de la propia clase Request. Con esto se consigue mantener una referencia a otro manejador (al sucesor).
- El método handleRequest() se declara como abstracto para obligar a que las subclases proporcionen una implementación.

Handler.java

```
package comportamiento.CoR.basico;

public abstract class Handler {

    protected Handler sucesor;

    public Handler getSucesor() {

        return sucesor;

    }

}
```

```

    public void setSucesor(Handler sucesor) {
        this.sucesor = sucesor;
    }

    public abstract void handleRequest(Request request);
}

```

Ahora es el turno de ver el código de las tres subclases de Handler. Hay que tener en cuenta que estas clases son prácticamente iguales, variando el algoritmo que comprueba si la petición es para ellas. Si la petición no es para ellas la pasan al siguiente manejador en la cadena. Esto lo consiguen obteniendo una referencia a su sucesor (invocando el método `getSucesor()` heredado de `Handler`) y llamando sobre esta referencia al método `handleRequest()`.

ConcreteHandlerNegativos.java

```

package comportamiento.CoR.basico;

public class ConcreteHandlerNegativos extends Handler {
    private String nomClase = this.getClass().getSimpleName();

    public void handleRequest(Request request) {

        // Si la petición puede ser procesada por esta clase
        if (request.getValor() < 0) {
            System.out.println("Valores negativos son manejados
por "
                               + nomClase + ":");

```



```

        System.out.println("\t" + nomClase +
".handleRequest : "

        + request.getValor());
    } else { // Si no, la pasamos al siguiente manejador
        getSucesor().handleRequest(request);
    }
}
}
}

```

ConcreteHandlerPositivos.java

```

package comportamiento.CoR.basico;

```

```

public class ConcreteHandlerPositivos extends Handler {
    private String nomClase = this.getClass().getSimpleName();

    public void handleRequest(Request request)    {

        // Si la peticion puede ser procesada por esta clase
        if (request.getValor() > 0) {
            System.out.println("Valores positivos son manejados
por " +

            nomClase + ":" );

            System.out.println("\t" + nomClase +
".handleRequest : " +

            request.getValor());
        } else { // Si no, la pasamos al siguiente manejador
            getSucesor().handleRequest(request);
        }
    }
}

```

```
}
```

ConcreteHandlerCero.java

```
package comportamiento.CoR.basico;

public class ConcreteHandlerCero extends Handler {
    private String nomClase = this.getClass().getSimpleName();

    public void handleRequest(Request request) {

        // Si la peticion puede ser procesada por esta clase
        if (request.getValor() == 0) {
            System.out.println("Valores cero son manejados por "
                               + nomClase + ":");

            System.out.println("\t" + nomClase +
                               ".handleRequest : "
                               + request.getValor());
        } else { // Si no, la pasamos al siguiente manejador
            getSucesor().handleRequest(request);
        }
    }
}
```

Finalmente, resta por ver el código para la clase cliente.

Esta clase se encarga de:

- Instanciar a los objetos manejadores de peticiones.
- Montar la cadena.

- Enviar las peticiones al primero de los manejadores de la cadena.

MainClient.java

```
package comportamiento.CoR.basico.client;

import java.util.Random;
import comportamiento.CoR.basico.*;

public class MainClient {

    public static void main(String[] args) {

        // Montar la cadena de objetos manejadores
        Handler h1 = new ConcreteHandlerNegativos();
        Handler h2 = new ConcreteHandlerPositivos();
        Handler h3 = new ConcreteHandlerCero();
        h1.setSucesor(h2);
        h2.setSucesor(h3);

        // Enviar 5 peticiones a la cadena
        for (int i = 0; i < 5; i++) {
            h1.handleRequest(new Request(getIntAleatorio()));
        }

    }

    private static int getIntAleatorio() {

        int num = new Random().nextInt(10);

        if (num != 0 && getSignoNegativoAleatorio()) {
            num = -num;
        }

        return num;
    }

}
```

```

private static boolean getSignoNegativoAleatorio() {

    return new Random().nextBoolean();

}
}

```

Salida:

```

<terminated> MainClient (3) [Java Application] /usr/lib/jvm/java-6-openjdk/
Valores negativos son manejados por ConcreteHandlerNegativos:
ConcreteHandlerNegativos.HandleRequest : -2
Valores cero son manejados por ConcreteHandlerCero:
ConcreteHandlerCero.HandleRequest : 0
Valores positivos son manejados por ConcreteHandlerPositivos:
ConcreteHandlerPositivos.HandleRequest : 8
Valores negativos son manejados por ConcreteHandlerNegativos:
ConcreteHandlerNegativos.HandleRequest : -6
Valores positivos son manejados por ConcreteHandlerPositivos:
ConcreteHandlerPositivos.HandleRequest : 7

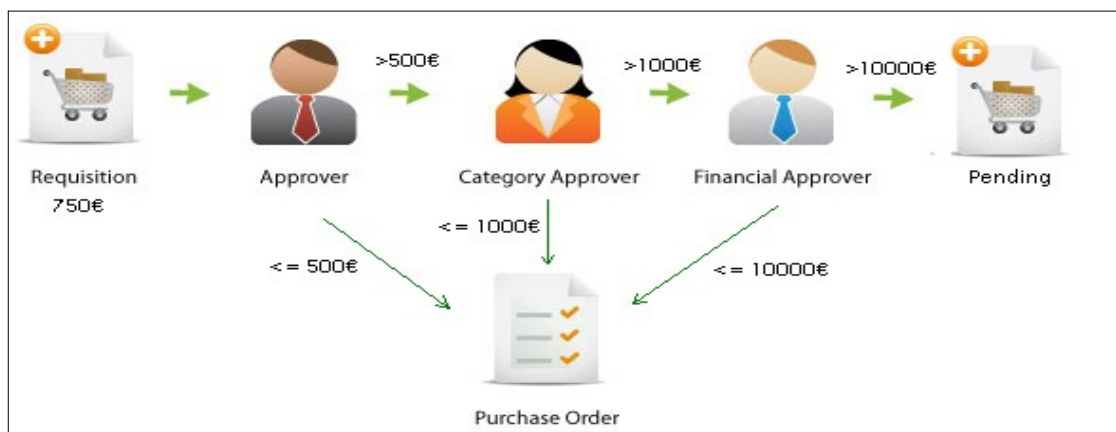
```

Ejemplo 2 – Flujo de aprobación de una petición de compra

Supongamos una empresa cuyo flujo de trabajo en relación a las compras de material, establece que una solicitud de compra puede aprobarse lo antes posible dentro de la cadena de aprobadores, siempre que un aprobador esté en condiciones de autorizar la compra.

Este podría ser el caso que ilustra la siguiente figura, en la que hay una jerarquía de empleados autorizados en la que cada empleado tiene un límite de autorización. Los empleados están ordenados por este límite:

Así, el primer empleado (Approver) puede satisfacer la petición de compra (Requisition) si ésta no es superior a un importe tope (500€) al que está autorizado. En

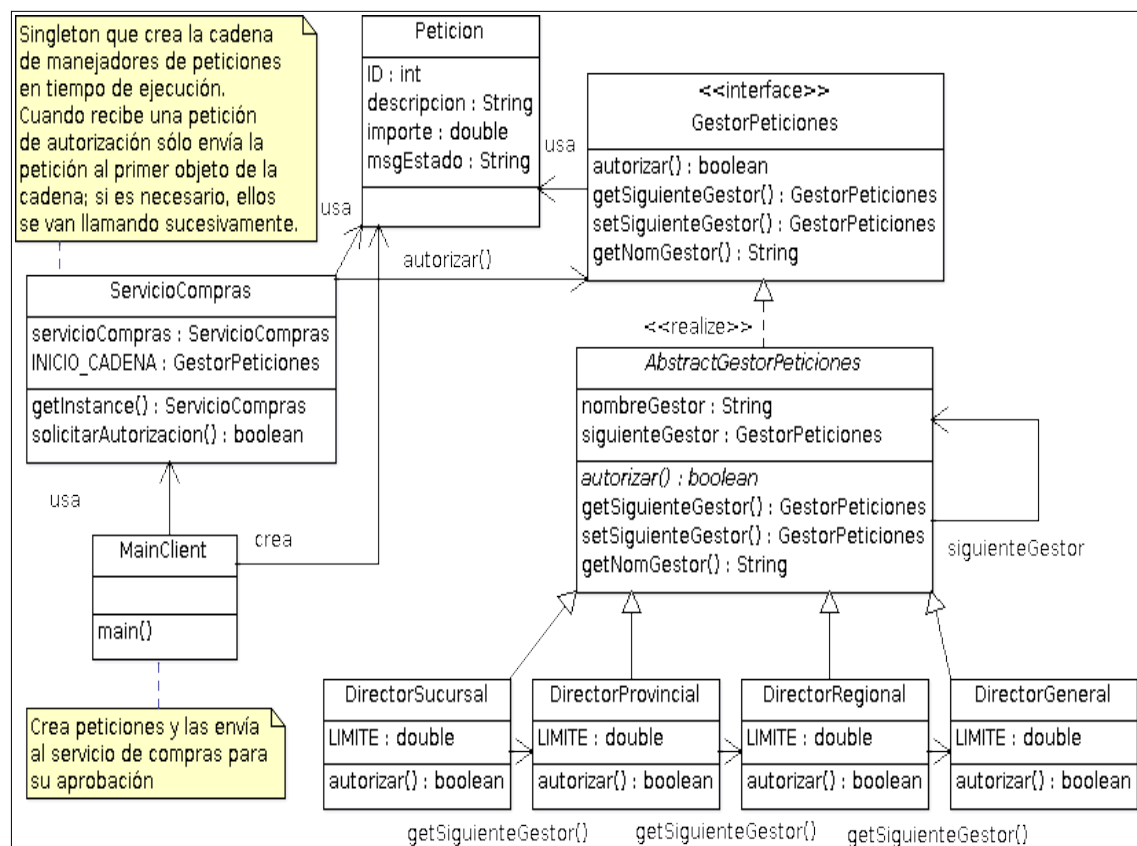


caso contrario, la decisión no está en las manos del primer receptor y tiene que pasar la petición a su sucesor (Category Approver), cuyo tope es superior (1000€). Y así sucesivamente hasta llegar al último de la cadena (Financial Approver).

Ahora bien, ¿qué sucede si el último empleado también tiene un importe tope y la petición lo supera? En este caso la petición atraviesa completamente la cadena y se queda sin aprobar.

No obstante, todo esto es muy relativo, pues está ligado a la lógica de negocio de la aplicación, ya que otro tipo de empresa podría preferir un flujo de aprobación de peticiones de compra sustancialmente diferente, por ejemplo, que las peticiones tuvieran que ser aprobadas por todos y cada uno de los receptores cuyo límite de autorización permitiera aprobar una petición de compra.

El siguiente diagrama de clases muestra las clases e interfaces que intervienen en el ejemplo, así como sus relaciones:



Comenzamos con la clase Peticion, la cual modela el concepto de una petición de compra. Notad que es una clase que carece de comportamiento -exceptuando los triviales getters y setters, en la que lo interesante es el valor de sus atributos:

Peticion.java

```
package comportamiento.CoR.solicitud_compra;

public class Peticion {

    private int ID;
    private String descripcion;
    private double importe;
    private String msgEstado;

    public Peticion(int id, String desc, double imp) {
        ID = id;
        descripcion = desc;
        importe = imp;
    }

    public double getImporte() { return importe; }

    public void setImporte(double importe) {
        this.importe = importe;
    }

    public String getMsgEstado() { return msgEstado; }

    public void setMsgEstado(String msgEstado) {
        this.msgEstado = msgEstado;
    }
}
```

```

    }

    @Override

    public String toString() {

        return msgEstado + "\n" +

            "[ID=" + ID + ", descripcion=" + descripcion + ",
importe=" + importe + "]" + "\n" +

            "*****
*****";

    }

}

```

Continuamos viendo ahora la interfaz que todo manejador de peticiones debe implementar:

GestorPeticiones.java

```

package comportamiento.CoR.solicitud_compra;

public interface GestorPeticiones {

    /** Intentar aprobar una solicitud */
    public boolean autorizar(SolicitudCompra solicitud);

    /** Obtener una referencia al siguiente manejador */
    public GestorPeticiones getSiguienteGestor();

    /** Establecer quien será el siguiente manejador de la cadena.
     * Notad que retorna el tipo GestorPeticiones, lo cual permite
     * que en el código cliente la creación de la cadena de gestores
     * se realice en una sola línea.
    */
}

```

```

        */

        public GestorPeticones setSiguienteGestor(GestorPeticones
gestor);

        /** Obtener el nombre (que no la referencia) del manejador
actual */

        public String getNomGestor();

    }

```

Ahora veamos el código para la superclase abstracta AbstractGestorPeticones. Esta clase:

- Define la operación abstracta autorizar() para obligar a sus subclases a implementarla.
- Proporciona una implementación de conveniencia en varios métodos que son heredados en sus subclases.

AbstractGestorPeticones.java

```

package comportamiento.CoR.solicitud_compra;

public abstract class AbstractGestorPeticones implements
GestorPeticones {

    private GestorPeticones siguienteGestor;

    private String nomGestor;

    public AbstractGestorPeticones(String nombre) {

        nomGestor = nombre;

    }

    @Override

    public abstract boolean autorizar(SolicitudCompra solicitud);

```



```

@Override

public GestorPeticiones getSiguienteGestor() {

    return siguienteGestor;

}

/*
 * Al retornar GestorPeticiones facilitamos la concatenacion de
 * los Gestores
 */
@Override

public GestorPeticiones setSiguienteGestor(GestorPeticiones
gestor) {

    siguienteGestor = gestor;

    return this;

}

@Override

public String getNomGestor() {

    return nomGestor;

}

}

```

Veamos ahora el código para las subclases. Notad que las cuatro subclases del ejemplo presentan la misma estructura:

- Obtener el importe de la petición.
- Comprobar si el importe es inferior al límite permitido.
- En caso afirmativo, aprobar la petición y finalizar el flujo de aprobación retornando *true*. En caso negativo, si hay un siguiente aprobador pasarle a éste

la petición (retornaremos lo que éste nos retorne); si no lo hay, indicar que no se ha podido resolver la petición retornando *false*.

DirectorSucursal.java

```
package comportamiento.CoR.solicitud_compra;
```

```
public class DirectorSucursal extends AbstractGestorPeticiones {
```

```
    private static double LIMITE = 5000;
```

```
    public DirectorSucursal(String nombre) {
```

```
        super(nombre);
```

```
    }
```

```
@Override
```

```
    public boolean autorizar(Peticion solicitud) {
```

```
        double importe = solicitud.getImporte();
```

```
        if (importe <= LIMITE) { // Autorizado
```

```
            String msgEstado = "Director Sucursal '" +  
getNomGestor()
```

```
                + "' ha autorizado la solicitud de  
compra.";
```

```
            solicitud.setMsgEstado(msgEstado);
```

```
            return true;
```

```
        } else {
```

```
            // Reenviar la solicitud al siguiente gestor de  
peticiones
```

```
            GestorPeticiones siguienteGestor =  
getSiguienteGestor();
```

```
            if (siguienteGestor != null) {
```



```

    public boolean autorizar(Peticion solicitud) {
        double importe = solicitud.getImporte();

        if (importe <= LIMITE) {
            String msgEstado = "Director Regional '" +
getNomGestor()
                                + "' ha autorizado la solicitud de
compra.";
            solicitud.setMsgEstado(msgEstado);
            return true;
        } else {
            // Reenviar la solicitud al siguiente gestor de
peticiones

            GestorPeticiones siguienteGestor =
getSiguienteGestor();
            if (siguienteGestor != null) {
                return siguienteGestor.autorizar(solicitud);
            } else {
                return false;
            }
        }
    }
}

```

DirectorGeneral.java

```

package comportamiento.CoR.solicitud_compra;

```

```

public class DirectorGeneral extends AbstractGestorPeticiones {

```

```

private static double LIMITE = 1000000;

public DirectorGeneral(String nombre) {
    super(nombre);
}

@Override
public boolean autorizar(Peticion solicitud) {
    double importe = solicitud.getImporte();

    if (importe <= LIMITE) {
        String msgEstado = "Director General '" +
getNomGestor()
        + "' ha autorizado la solicitud de
        compra.";
        solicitud.setMsgEstado(msgEstado);
        return true;
    } else {
        // Reenviar la solicitud al siguiente gestor de
        peticiones
        GestorPeticiones siguienteGestor =
        getSiguienteGestor();
        if (siguienteGestor != null) { // no entrará en el
        if, ya que es el último de la cadena
            return siguienteGestor.autorizar(solicitud);
        } else {
            String msgEstado = "***Peticion no aprobada.";
            solicitud.setMsgEstado(msgEstado);
            return false;
        }
    }
}

```

```
}  
  
}
```

Llegamos ahora a la clase `ServicioCompras`, la cual no es una clase del patrón CoR, sino una clase cliente del patrón, de conveniencia, empleada en nuestro ejemplo para ocultar a la aplicación cliente “real” la complejidad del patrón.

Lo más interesante de esta clase:

- En su constructor, la creación de la cadena de objetos manejadores de peticiones.
- En su método `solicitarAutorizacion()`, el comienzo del flujo de aprobación.

`ServicioCompras.java`

```
package comportamiento.CoR.solicitud_compra;  
  
public class ServicioCompras {  
  
    private final static ServicioCompras  
        servicioCompras = new ServicioCompras();  
  
    private final GestorPeticiones INICIO_CADENA;  
  
    // Constructor privado, no instanciable  
    private ServicioCompras() {  
        INICIO_CADENA =  
            new DirectorSucursal("Sr. Garcia")  
                .setSiguienteGestor(new  
DirectorProvincial("Sra. Martinez")  
                    .setSiguienteGestor(new  
DirectorRegional("Sr. Perez"))
```

```

        .setSiguienteGestor(new
DirectorGeneral("Sra. Sanchez")
        .setSiguienteGestor(null)));
    }

    public static ServicioCompras getInstance() {
        return servicioCompras;
    }

    public boolean solicitarAutorizacion(Peticion solicitud) {

        return INICIO_CADENA.autorizar(solicitud);
    }
}

```

Por último, veamos el código cliente “real”:

MainClient.java

```

package comportamiento.CoR.solicitud_compra.client;

import comportamiento.CoR.solicitud_compra.ServicioCompras;
import comportamiento.CoR.solicitud_compra.Peticion;

public class MainClient {

    public static void main(String[] args) {

        /*
         * Creamos 3 solicitudes
         */
    }
}

```



```

Peticion[] solicitudes = {
    new Peticion(1000, "Ordenadores laboratorio", 25000),
    new Peticion(1001, "Viaje feria Alemania", 25800),
    new Peticion(1002, "Nuevo almacen BCN", 1000001)
};

/*
 * Obtenemos una instancia del servicio de compras y
pedimos
 * la aprobación de cada una de la solicitudes de compra
 */
ServicioCompras servicio = ServicioCompras.getInstance();

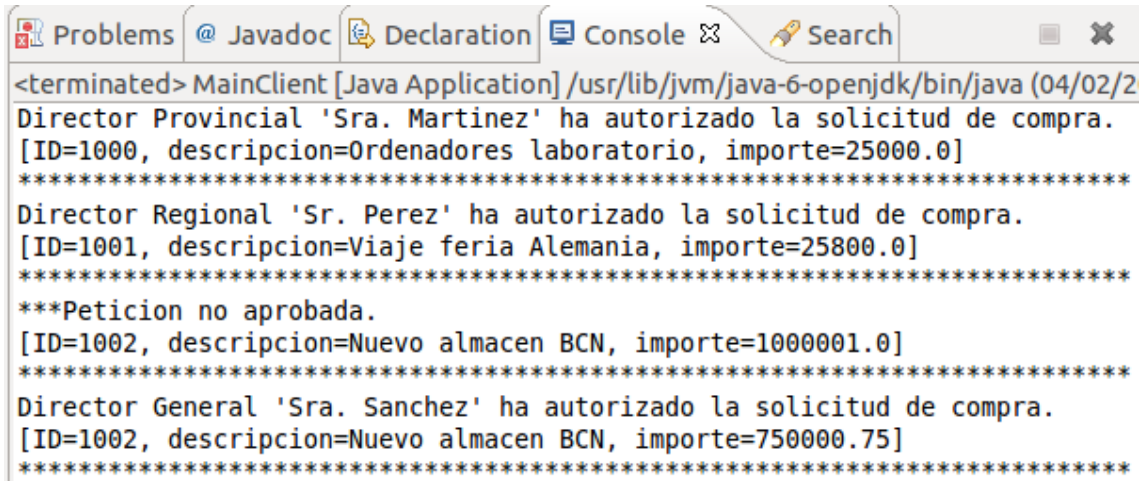
for (Peticion solicitud : solicitudes) {
    boolean aprobada =
servicio.solicitarAutorizacion(solicitud);
    System.out.println(solicitud);
    if (!aprobada) {
        //Simulamos que hemos conseguido un descuento y
        //la volvemos a enviar al flujo de aprobación
        double importe =
getDescuento(solicitud.getImporte());
        solicitud.setImporte(importe);
        servicio.solicitarAutorizacion(solicitud);
        System.out.println(solicitud);
    }
}

private static double getDescuento(double importe) {
    return importe*0.75;
}

```

```
}  
  
}
```

Salida:



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, Console, and Search. The Console tab is active, displaying the output of a Java application. The output text is as follows:

```
<terminated> MainClient [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (04/02/2  
Director Provincial 'Sra. Martinez' ha autorizado la solicitud de compra.  
[ID=1000, descripcion=Ordenadores laboratorio, importe=25000.0]  
*****  
Director Regional 'Sr. Perez' ha autorizado la solicitud de compra.  
[ID=1001, descripcion=Viaje feria Alemania, importe=25800.0]  
*****  
***Peticion no aprobada.  
[ID=1002, descripcion=Nuevo almacen BCN, importe=1000001.0]  
*****  
Director General 'Sra. Sanchez' ha autorizado la solicitud de compra.  
[ID=1002, descripcion=Nuevo almacen BCN, importe=750000.75]  
*****
```

Ejemplo 3 – Framework de logs

Es habitual hacer que una aplicación genere mensajes que sirvan de retroalimentación tanto a los programadores como a los administradores y usuarios del programa.

Los mensajes pueden mostrarse por pantalla, guardarse en un archivo o incluso enviarse por correo electrónico o enviarse a otro programa remoto (como en el caso de Chainsaw que viene con Java).

Naturaleza de los mensajes

Los mensajes suelen informar de diferentes aspectos de la aplicación. Veamos un par de enfoques:

- Mientras un programador desarrolla o prueba una funcionalidad, quiere que la aplicación sea lo más explícita posible sobre lo que va haciendo en cada momento de su ejecución. Debido a esto, el programador tiene que escribir los mensajes allá donde quiera que aparezcan. En cambio, una vez dada por buena esa funcionalidad ya no se necesitan todos esos mensajes y se tienen que eliminar.
- Hay mensajes que se quieren conservar una vez desarrollada la aplicación y puesta en el entorno de producción. Por ejemplo, un administrador de la

aplicación querría que ésta llevase un registro en disco del número de intentos fallidos de un usuario al introducir las credenciales durante el acceso a algún recurso protegido.

Opciones para generar mensajes

Cuando estamos desarrollando, para mostrar los mensajes en nuestras aplicaciones, siempre podemos utilizar la familia de métodos `System.out.println()`. Sin embargo, esto no es práctico, ya que es muy poco flexible y difícil de mantener.

Es poco flexible porque no da margen de maniobra. Por ejemplo, con `System.out.print()` no podemos indicar un filtro para que en el entorno de desarrollo (mientras elaboramos el programa) se muestren todo tipo de mensajes, esto es, tanto los mensajes leves, como los moderados, importantes y graves, mientras que en el entorno de producción sólo se muestren los mensajes importantes y graves.

Por otro lado, `System.out.print()` es costoso de mantener porque cuando la aplicación ya está lista para entrar en producción, dado que no queremos que aparezcan los mensajes que se mostraban en desarrollo, tenemos que recorrer todo el código localizando y eliminando aquellos mensajes que sólo tenían sentido en el entorno de desarrollo. Esto requiere mucho esfuerzo y con frecuencia quedan mensajes sin eliminar que le serán mostrados al usuario sin ninguna lógica aparente. Además si hemos eliminado (o comentado) todos los `System.out.print()` porque ya no eran necesarios y sin embargo ahora los volvemos a necesitar, entonces tenemos que deshacer todo lo hecho a este respecto. Por todo esto se dice que `System.out.print()` es poco flexible y costoso de mantener.

Mejor alternativa

Un framework de log nos da la solución a las necesidades comentadas, proporcionando un medio sencillo y flexible de gestionar los mensajes de la aplicación. Veamos algunas de las características que implementaremos en nuestro ejemplo:

Características que presentará nuestro framework

Nivel de log

Tendremos varios tipos (niveles) de mensajes para permitir al código cliente afinar respecto a qué se quiere expresar en cada momento.

Veamos estos niveles:

- TRAZA: Es el menos importante y se suele usar en la fase de desarrollo, para que el programador sepa por donde va pasando el programa. Es el típico `System.out.println("el programa pasa por esta linea de código")`.
- DEBUG: Es para información útil para depurar, como algún resultado parcial, el valor de alguna variable, etc.
- INFO: Es para cosas normales en la aplicación que pueden tener cierto interés para mostrar en el log. Por ejemplo, se establece una conexión con base de datos, se conecta un cliente al servidor, un usuario entra en sesión, etc.
- AVISO: Para pequeños fallos de los que la aplicación se puede recuperar fácilmente, por ejemplo, un usuario introduce una contraseña errónea, un fichero de configuración no existe y la aplicación toma entonces la configuración por defecto que se establezca en el código, etc.
- ERROR: Para errores importantes, pero que no obligan a terminar la aplicación. Por ejemplo, no es posible conectar a la base de datos, por lo que se utilizará un sistema de persistencia basado en ficheros en disco. También porque existen otras funcionalidades de la aplicación que sí pueden seguir ofreciéndose, aun sin base de datos.
- FATAL: Para errores que obligan a terminar la aplicación, por ejemplo, se ha terminado la memoria disponible.

Lo mejor de disponer de diferentes niveles de log es que el código cliente puede decidir dinámicamente a partir de qué nivel de log quiere que se muestren los mensajes. Por ejemplo, una vez puesta la aplicación en el entorno de producción ya no tendrá sentido mostrar los mensajes con nivel TRAZA y DEBUG, pues son propios de la fase de desarrollo de la aplicación.

Nosotros no lo haremos pero lo habitual es disponer de un fichero de configuración externo a la aplicación para modificar estos valores sin tener que recompilar el código Java. Así, cuando se carga el programa se lee el fichero y dinámicamente se establece la configuración de log para esa ejecución.

Sencilla activación/desactivación del sistema de log

Mediante una única instrucción el código cliente puede desactivar o reactivar la generación de mensajes.

Tipo de salida: pantalla, fichero, ambos

Proporcionamos tres posibilidades como destino para los mensajes: sólo la consola, sólo un fichero en disco o ambos.

Formato de los mensajes

No ofrecemos la posibilidad de que el código cliente pueda personalizar el formato de los mensajes. Sencillamente antepondremos la fecha/hora con precisión de milisegundos al mensaje pasado como argumento.

Varias configuraciones

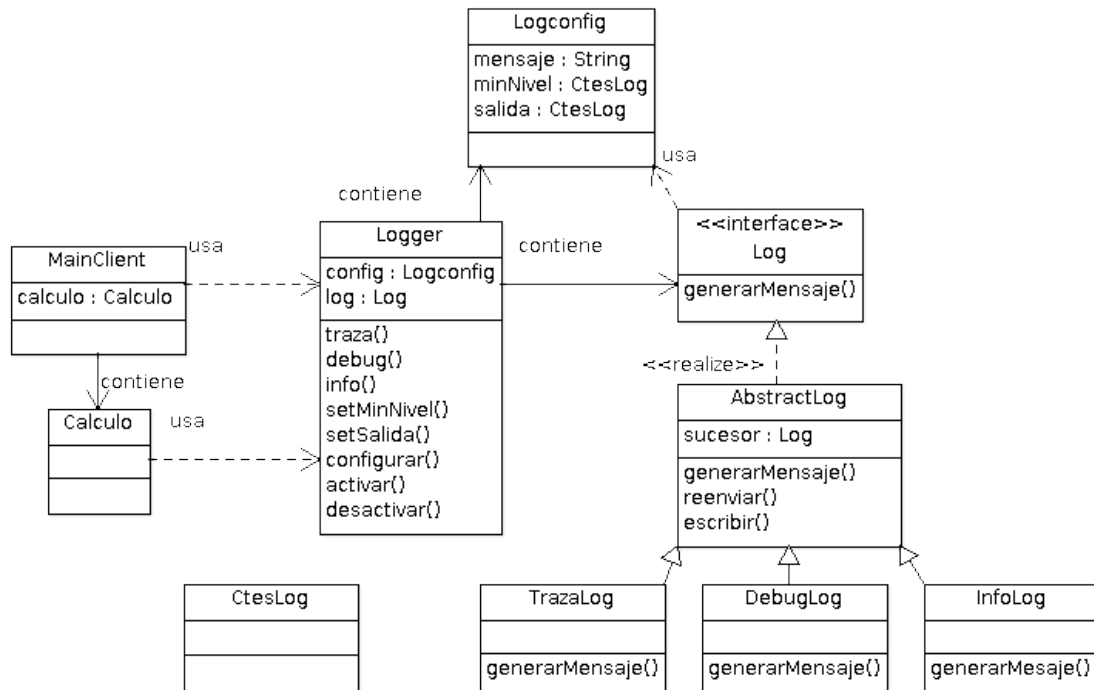
El framework debe permitir que cada clase configure el sistema de log de una manera diferente. Por ejemplo, dadas dos clases cliente, la clase A y la clase B, debería ser posible que A configurase el framework para generar mensajes sólo en pantalla y cuando el tipo (nivel) de log sea AVISO, ERROR y FATAL. En cambio, B tendría que poder generar mensajes sólo en fichero para cualquier nivel de log.

Nota: Limitaremos a un único fichero para todos los mensajes, el cual será compartido por todas las clases.

Después de la explicación anterior veamos cómo el patrón CoR nos puede ayudar a diseñar el framework de log.

La siguiente figura muestra el diagrama de clases de la aplicación.

Importante: Notad que para no ser repetitivos no se han representado todas las subclases de AbstractLog.



Breve explicación sobre las clases del diagrama

La aplicación presenta un arquitectura dividida en tres niveles. De derecha a izquierda tenemos:

- La jerarquía de clases que constituye propiamente el framework de log. Estas clases implementan el patrón CoR para procesar en cadena los mensajes (peticiones) enviados por las clases cliente. Como vemos, hay una interfaz denominada Log que define un único método denominado generarMensaje(). Esta interfaz permite usar toda la potencia del framework a la vez que evita que se revelen los detalles de implementación del mismo. Notad que la clase Logger contiene una referencia de esta interfaz para comunicarse con el framework. No obstante, por simplicidad, la clase Logger se acopla con las subclases de la jerarquía en el momento que crea la cadena de receptores (cuando hace los “new” de cada una). Una solución más sofisticada utilizará un contenedor de beans para utiliza Inyección de dependencias.

- Una clase denominada Logger que actúa de wrapper (envoltorio) o decorador del sistema de log para el código cliente. Es una clase de conveniencia, luego no es imprescindible, que facilita el uso del framework de log a las clases cliente, además de desacoplarlo totalmente. Logger crea un objeto de la clase LogConfig por cada clase cliente que quiere utilizar el sistema de log. LogConfig no define comportamiento alguno, sino que se limita únicamente a contener datos. Se trata de la petición que envía el código cliente con la intención de que alguna clase receptora -implementando el patrón CoR- la despache.
- MainClient y Calculo son clases cliente que utilizan el framework de log para generar mensajes. MainClient contiene un método main() y una referencia de tipo Calculo. Decidimos arbitrariamente que MainClient sólo generará mensajes por consola mientras Calculo sólo lo hará en el fichero.

Comenzamos por ver el código para la enum CtesLog, que define las constantes a utilizar, tanto por parte del framework como por parte del código cliente.

CtesLog.java

```
package comportamiento.CoR.logfrwk.core;
```

```
public enum CtesLog {
    TRAZA, DEBUG, INFO, AVISO, ERROR, FATAL, // niveles de log
    PANTALLA, FICHERO, PANTALLA_Y_FICHERO;    // tipos de salida
}
```

A continuación, veamos el código para la clase LogConfig. Esta clase, como se comentó anteriormente, se trata de una simple encapsulación de datos relativos a la petición que el código cliente envía al framework. La clase contendrá el mensaje propiamente, así como la configuración que en ese momento desea el usuario. Por ejemplo, mostrar el mensaje en fichero, sólo aquellos con nivel de error y fatal.

Revisando el diagrama de clases, se puede apreciar que cada clase cliente crea una instancia de la clase Logger (MainClient crea una y Calculo crea otra) y que por cada

Logger creado se crea una instancia de LogConfig. Esto permite disponer de una configuración individual por cada clase cliente, lo cual hace posible, por ejemplo, que una clase cliente desactive el sistema de log y que otras no.

Notad que los atributos de LogConfig se inicializan con valores predeterminados. Esto permite que el usuario no deba configurar nada para poder emplear el sistema de logs.

LogConfig.java

```
package comportamiento.CoR.logfrwk.core;

import static comportamiento.CoR.logfrwk.core.CtesLog.*;

public class LogConfig {

    private CtesLog nivelMin = TRAZA;
    private CtesLog salida = PANTALLA;
    private boolean logActivo = true;

    private String msg;

    public CtesLog getNivelMin() { return nivelMin;}
    public CtesLog getSalida() { return salida;    }
    public boolean isLogActivo() { return logActivo; }

    public String getMsg() {return msg; }

    public void setNivelMin(CtesLog nivelMin) {
        this.nivelMin = nivelMin;
    }
}
```



```

    public void setSalida(CtesLog salida) {
        this.salida = salida;
    }

    public void setLogActivo(boolean log_activo) {
        this.logActivo = log_activo;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }
}

```

Veamos ahora la jerarquía del framework, cuyo pilar básico es la implementación del patrón Chain of Responsibility.

Cómo se comentó anteriormente, el acceso al framework se lleva a cabo a través de una interfaz denominada Log, la cual únicamente define un método llamado generarMensaje().

Log.java

```

package comportamiento.CoR.logfrwk.core;

```

```

public interface Log {

```

```

    /**
     * Método abstracto a implementar en cada subclase.
     * Toda subclase comprobará si el nivel de log recibido por
parámetro

```

```

        * es el adecuado para ella, en cuyo caso imprimirá el mensaje.
En
        * caso contrario llamará al método reenviar() para pasar la
peticion
        * a la siguiente clase de la cadena
        */

    public abstract void generarMensaje(CtesLog nivel, LogConfig
config);

}

```

Ahora es el turno de la superclase abstracta AbstractLog. Esta clase, junto con sus subclases, es fundamental en la implementación del framework. En particular, AbstractLog tiene las siguientes responsabilidades:

- Contiene un atributo de tipo Log para apuntar al siguiente objeto receptor en la cadena. Esta referencia se asigna en el constructor.
- Define (pero no implementa) el método abstracto generarMensaje() para que las subclases lo implementen.
- Implementa el método reenviar(), proporcionando una implementación por defecto para las subclases, consistente en pasar la petición al siguiente objeto en la cadena, siempre que haya un objeto al que pasársela. Las subclases son libres de redefinir o extender esta implementación, aunque en nuestro caso esto no será necesario.
- Implementa el método escribir(), proporcionando una implementación por defecto para las subclases, consistente en comprobar los valores del objeto LogConfig para determinar qué se debe escribir (el mensaje) y cómo hay que hacerlo (¿pantalla?¿fichero?¿ambos?). Las subclases son libres de redefinir o extender esta implementación, aunque en nuestro caso esto no será necesario.
- Implementa otros métodos privados, destacando el método escribirEnFichero(), encargado de escribir en disco los mensajes que así lo requieran.

Sobre la generación del fichero: por simplicidad, el fichero tiene un nombre fijo, consistente en el nombre completamente cualificado del paquete donde reside el framework más la palabra “log”. El fichero aparecerá dentro del proyecto. Esto se hace así para simplificar su consulta dentro del Eclipse, por lo que en una aplicación real emplearíamos un sistema más flexible.

Veamos ahora las subclases que implementan AbstractLog. Dado que todas son iguales, variando el nivel de log específico de cada una, sólo mostraremos la primera. Notad la simplicidad de este tipo de clases, lo cual es normal ya que la mayoría del código se hereda de AbstractLog. A destacar:

- En el constructor es obligado pasar el sucesor al constructor de AbstractLog. Esto es fundamental para la correcta construcción de la cadena de receptores.
- Implementación del método abstracto generarMensaje(), definido en la interfaz Log. El código para este método es prácticamente igual en todas las subclases: comprobar si el mensaje puede ser despachado por la clase; en caso afirmativo, personalizarlo, manipulando el objeto LogConfig, y pasarlo como argumento al método escribir() heredado de AbstractLog. Con lo que la petición se habrá resuelto aquí. En caso contrario, reenviar la petición al siguiente receptor.

TrazaLog.java

```
package comportamiento.CoR.logfrwk.core;
```

```
import static comportamiento.CoR.logfrwk.core.CtesLog.*;
```

```
public class TrazaLog extends AbstractLog {
```

```
    public TrazaLog(Log sucesor) {
```

```
        super(sucesor);
```

```
    }
```

```
@Override
```

```

    public void generarMensaje(CtesLog nivel, LogConfig config) {
        if (procedeMostrar(nivel, config.getNivelMin())) {
            String msg = "[TRAZA] " + config.getMsg();
            config.setMsg(msg);
            escribir(config);
        } else {
            reenviar(nivel, config);
        }
    }

    private boolean procedeMostrar(CtesLog nivel, CtesLog nivelMin) {
        return nivel == TRAZA &&
            nivel.ordinal() >= nivelMin.ordinal();
    }
}

```

Nota: para crear el resto de clases tenemos que sustituir las dos apariciones de “TRAZA” por lo que corresponda. Al final del código se muestra una imagen del proyecto para facilitar esta tarea.

Las clases mostradas hasta el momento son la parte nuclear del framework. Sin embargo, para facilitar el uso del mismo a las clases cliente, es una buena opción proporcionar una clase “amiga” tanto del framework como del código cliente. Esta clase envolverá la funcionalidad del sistema de log ofreciendo una serie de métodos sencillos de utilizar al código cliente y ocultándolo completamente.

Ha de quedar claro que esta clase wrapper no forma parte del core del framework, y aunque no se beneficia de él se puede considerar como una clase cliente.

Logger.java

```

package comportamiento.CoR.logfrwk.wrapper;

```

```

/*

```

- * Logger es una clase envoltorio que facilita el uso del
- * framework de log a las clases cliente, ocultando los
- * detalles de implementacion y proporcionando una serie
- * de métodos fáciles de utilizar.
- *
- * Por cada clase cliente que quiera usar una configuracion
- * distinta de log, se tiene que crear una instancia de Logger.
- * A su vez, con cada ejecucion del constructor de logger se
- * crea una instancia del framework de log.
- *
- * Notad que su constructor se encarga de crear la cadena
- * objetos receptores de peticiones de log.
- */

```
import static comportamiento.CoR.logfrwk.core.CtesLog.*;
```

```
import comportamiento.CoR.logfrwk.core.*;
```

```
public class Logger {
```

```
    // Referencia a la instancia del framework
```

```
    private Log log;
```

```
    // Nombre de la clase que llama. Util para mostrarlo
```

```
    // junto a cada mensaje de log
```

```
    private String llamante;
```

```
    // Configuracion de log asociada a esta clase cliente
```

```
    private LogConfig config;
```

```
    // Constructor
```

```

public Logger(Class<?> claseLlamante) {

    /*
     * Creacion de la cadena de manejadores de log.
     *
     * Notad que al ultimo de la cadena le decimos
     * que su sucesor es null para marcar el final.
     * Ordenamos la cadena según nos interesa: el nivel
     * menor es TRAZA y el mayor FATAL
     */
    log =
        new TrazaLog(
            new DebugLog(
                new InfoLog(
                    new AvisoLog(
                        new ErrorLog(
                            new
FatalLog(null))));

        llamante = claseLlamante.getSimpleName();
        config = new LogConfig(); // Configuracion de Log
predeterminada
    }

    // ***** NIVEL DE LOG *****

    public void traza(String msg) {
        config.setMsg "[" + llamante + "] : " + msg);
        log.generarMensaje(TRAZA, config);
    }

```

```

public void debug(String msg) {
    config.setMsg "[" + llamante + "]" : " + msg);
    log.generarMensaje(DEBUG, config);
}

public void info(String msg) {
    config.setMsg "[" + llamante + "]" : " + msg);
    log.generarMensaje(INFO, config);
}

public void aviso(String msg) {
    config.setMsg "[" + llamante + "]" : " + msg);
    log.generarMensaje(AVISO, config);
}

public void error(String msg) {
    config.setMsg "[" + llamante + "]" : " + msg);
    log.generarMensaje(ERROR, config);
}

public void fatal(String msg) {
    config.setMsg "[" + llamante + "]" : " + msg);
    log.generarMensaje(FATAL, config);
}

// ***** CONFIGURACION FINA *****

public void setMinNivel(CtesLog minNivel) {

```

```

        config.setNivelMin(minNivel);
    }

    public void setSalida(CtesLog salida) {
        config.setSalida(salida);
    }

    public void activar() {
        config.setLogActivo(true);
    }

    public void desactivar() {
        config.setLogActivo(false);
    }

    // ***** CONFIGURACION GRUESA *****

    public void configurar(CtesLog minNivel, CtesLog salida) {
        setMinNivel(minNivel);
        setSalida(salida);
    }

}

```

Y por fin llegamos a las clases cliente. En nuestro ejemplo tendremos dos clases, una conteniendo el método main() y otra que será un objeto subordinado a la primera. Comenzamos por la que contiene el main(). Notad que se muestran algunos ejemplos alternativos entre comentarios sobre cómo configurar el sistema de log. Experimentad con ellos o con otros propios.

MainClient.java

```
package comportamiento.CoR.logfrwk.client;
```

```
import comportamiento.CoR.logfrwk.wrapper.Logger;
```

```
import static comportamiento.CoR.logfrwk.core.CtesLog.*;
```

```
public class MainClient {
```

```
    private static final Logger log = new Logger(MainClient.class);
```

```
    private Calculo calculo = new Calculo();
```

```
    public MainClient() {
```

```
        /*
```

```
        * Si no establecemos ninguna configuracion el framework  
de log
```

```
        * adopta una por defecto, consistente en mostrar mensajes  
por
```

```
        * pantalla a partir del menor nivel de log (TRAZA)
```

```
        */
```

```
        //ejemplo1ConfigurarLog();
```

```
        //ejemplo2ConfigurarLog();
```

```
        //ejemplo3ConfigurarLog();
```

```
        log.traza("llamando a metodoSuma());
```

```
        calculo.sumar(Integer.MAX_VALUE, Integer.MAX_VALUE);
```

```
        log.traza("llamando a metodoResta());
```

```
        calculo.restar(2.00, 1.10);
```

```

        log.traza("llamando a metodoBucle()");
        metodoBucle();
    }

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

    private void metodoBucle() {
        // Idem a double i = 1.0 / 0.0;
        double infinito = Double.POSITIVE_INFINITY;

        while (infinito + 1 == infinito + 2) {
            log.avisó("Imposible que se muestre esto, ¿o no?");
            break;
        }
    }

    /*
     * En este ejemplo se genera un fichero de log ademas de mostrar
     * los mensajes en pantalla
     */
    private void ejemplo1ConfigurarLog() {

        /*
         * -primer param: nivel de log a partir del cual hay que
         * mostrar mensajes
         * -segundo param: sólo fichero o fichero y pantalla

```

```

        */

        log.configurar(TRAZA, PANTALLA_Y_FICHERO);

        // Las dos siguientes líneas son equivalentes a la anterior:
        //log.setMinNivel(TRAZA);
        //log.setSalida(PANTALLA_Y_FICHERO);

    }

    /**
     * En este ejemplo sólo se muestran los mensajes por pantalla y
     * con un nivel de log mayor o igual a AVISO
     */
    private void ejemplo2ConfigurarLog() {
        log.setMinNivel(AVISO);
    }

    /**
     * En este ejemplo se desactiva todo el sistema de log, por lo
     * que no se mostrará ningún mensaje
     */
    private void ejemplo3ConfigurarLog() {
        log.desactivar();

        // Si queremos activar los logs
        // log.activar();
    }

}

```

Veamos por último la clase Calculo:

Calculo.java

```
package comportamiento.CoR.logfrwk.client;

import comportamiento.CoR.logfrwk.wrapper.Logger;

import static comportamiento.CoR.logfrwk.core.CtesLog.*;

public class Calculo {

    private static final Logger log = new Logger(Calculo.class);

    public Calculo() {
        log.configurar(TRAZA, FICHERO);
    }

    public double sumar(int sumando1, int sumando2) {

        log.traza("Primer argumento pasado al metodo sumar(): " +
sumando1);

        log.traza("Segundo argumento pasado al metodo sumar(): " +
sumando2);

        long totalSuma = sumando1 + sumando2;

        if ((sumando1 > 0 && sumando2 > Integer.MAX_VALUE -
sumando1) ||
            (sumando1 < 0 && sumando2 < Integer.MIN_VALUE -
sumando1))
        {
```

```

        log.error("Se ha producido desbordamiento al
realizar la suma! Resultado: " + totalSuma);

        log.info("Haciendo un cast a long para evitar el
desbordamiento...");

        totalSuma = (long) sumando1 + sumando2;
    }

    log.traza("La suma es: " + totalSuma);

    return totalSuma;
}

/*
 * Los ordenadores no pueden representar todos los numeros
reales.
 * La logica de punto flotante es particularmente mala para
 * los calculos monetarios, ya que es imposible representar 0.1
 * o cualquier potencia negativa de 10 (1/10, 1/100, etc...)
 * exactamente como un fraccion de longitud fija.
 * Para calculos monetarios o similares es mejor utilizar int,
 * long, o la clase BigDecimal. Podriamos hacer lo siguiente:
 *
 * String strMinuendo = String.valueOf(minuendo);
 * String strSustraendo = String.valueOf(sustraendo);
 * // Lo mejor es usar String's para el constructor
 * BigDecimal BDminuendo = new BigDecimal(strMinuendo);
 * BigDecimal BDsustraendo = new BigDecimal(strSustraendo);
 *
 * return BDminuendo.subtract(BDsustraendo).doubleValue();

```

```

    */

    public double restar(double minuendo, double sustraendo) {

        log.traza("Primer argumento pasado al metodo restar(): " +
minuendo);

        log.traza("Segundo argumento pasado al metodo restar(): "
+ sustraendo);

        double totalResta = minuendo - sustraendo;

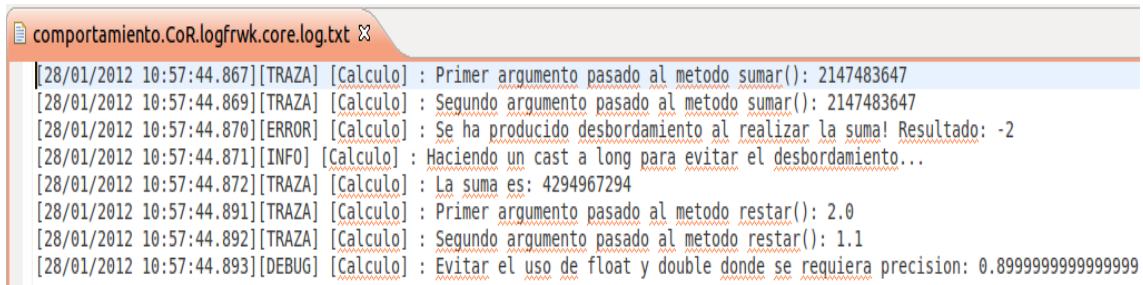
        log.debug("Evitar el uso de float y double donde " +
                "se requiera precision: " + totalResta);

        return totalResta;

    }
}

```

Salida (en fichero):

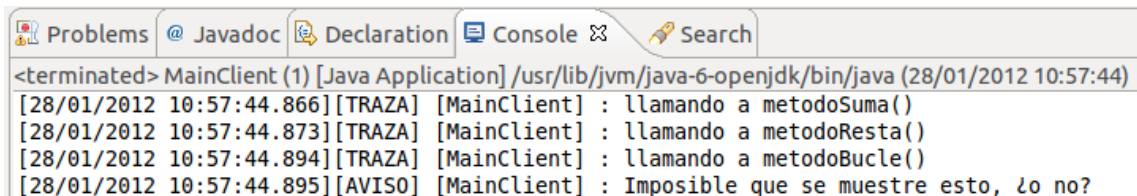


```

comportamiento.CoR.logfrwk.core.log.txt
[28/01/2012 10:57:44.867][TRAZA] [Calculo] : Primer argumento pasado al metodo sumar(): 2147483647
[28/01/2012 10:57:44.869][TRAZA] [Calculo] : Segundo argumento pasado al metodo sumar(): 2147483647
[28/01/2012 10:57:44.870][ERROR] [Calculo] : Se ha producido desbordamiento al realizar la suma! Resultado: -2
[28/01/2012 10:57:44.871][INFO] [Calculo] : Haciendo un cast a long para evitar el desbordamiento...
[28/01/2012 10:57:44.872][TRAZA] [Calculo] : La suma es: 4294967294
[28/01/2012 10:57:44.891][TRAZA] [Calculo] : Primer argumento pasado al metodo restar(): 2.0
[28/01/2012 10:57:44.892][TRAZA] [Calculo] : Segundo argumento pasado al metodo restar(): 1.1
[28/01/2012 10:57:44.893][DEBUG] [Calculo] : Evitar el uso de float y double donde se requiera precision: 0.8999999999999999

```

Salida (en consola):

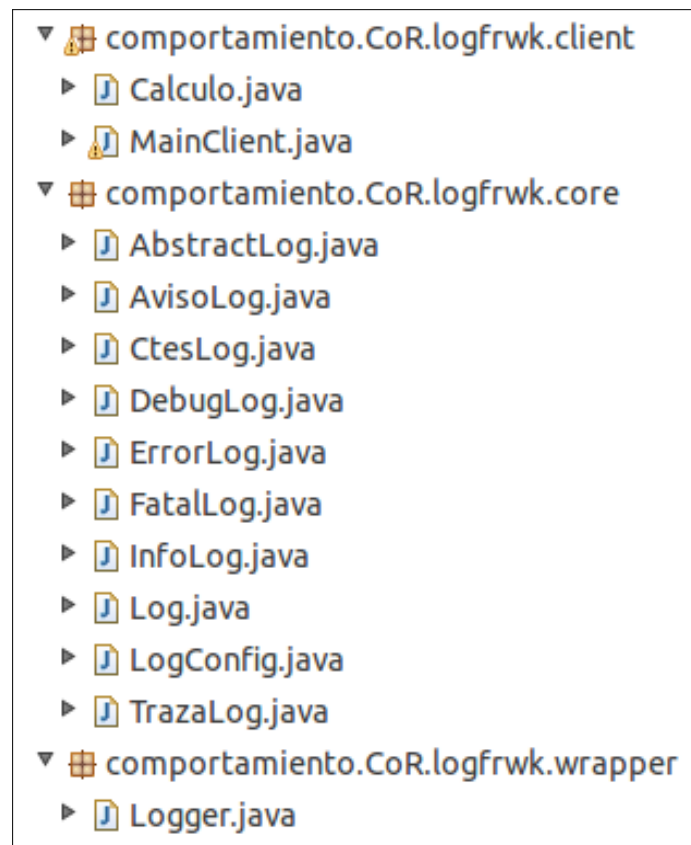


```

Problems  Javadoc  Declaration  Console  Search
<terminated> MainClient (1) [Java Application] /usr/lib/jvm/java-6-openjdk/bin/java (28/01/2012 10:57:44)
[28/01/2012 10:57:44.866][TRAZA] [MainClient] : llamando a metodoSuma()
[28/01/2012 10:57:44.873][TRAZA] [MainClient] : llamando a metodoResta()
[28/01/2012 10:57:44.894][TRAZA] [MainClient] : llamando a metodoBucle()
[28/01/2012 10:57:44.895][AVISO] [MainClient] : Imposible que se muestre esto, ¿o no?

```

La siguiente figura muestra la disposición de clases y paquetes:



Ejemplo 4 – Filtros Servlet

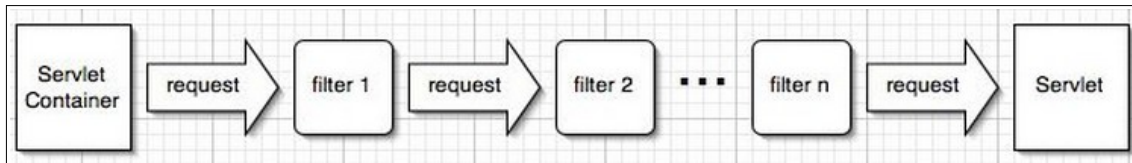
Los filtros del API Servlet son un ejemplo del patrón Chain on Responsibility. Estos filtros se pueden encadenar para realizar tareas de preprocesamiento sobre la petición (request) y tareas de postprocesamiento sobre la respuesta (response).

Algunos ejemplos típicos de filtros:

- Seguridad. Bloqueo de peticiones al servidor web en función de la identidad del usuario.
- Auditoria y registro (logging) de las interacciones de los usuarios de una aplicación web.
- Conversión de imágenes.
- Compresión de datos para acelerar las descargas.
- Cifrado de datos.

- Transformaciones XSL/T de contenido XML para más de un tipo de cliente (dispositivo físico).
- Y en general, cualquier comportamiento que pueda considerarse “infraestructura”, en lugar de “lógica” específica para un componente web.

La figura siguiente muestra el esquema típico del uso de filtros servlet:



El ejemplo

El ejemplo que veremos a continuación consiste en un simple servlet que genera un documento HTML para el navegador, es decir no delega en ninguna página JSP o HTML, sino que escribe directamente un texto en la respuesta (response). Así, el usuario introducirá el URL del Servlet y esperará obtener una página HTML con el texto generado por el servlet.

La gracia es que el texto no será el que hayamos escrito originalmente en el servlet, sino (ligeramente) modificado por un filtro. Este filtro busca una cadena de texto como patrón y la reemplaza por otro texto. Tanto la cadena a buscar como el texto que la reemplazará son parámetros que se configuran en el descriptor de despliegue (web.xml).

Además, tendremos otro filtro para llevar una auditoría de las URI's que solicitan los usuarios y el tiempo que nuestro servidor tarda en responderlas. A decir verdad, este filtro no es muy interesante cuando sólo tenemos un componente en el servidor al que hacer peticiones (como es nuestro caso), sin embargo es muy práctico en fase de desarrollo y depuración cuando tenemos varios componentes (varios servlets, páginas JSP, HTML, etc).

Para crear un filtro servlet tenemos que hacer tres cosas:

- Crear un servlet (creando una clase que extienda a la clase `javax.servlet.http.HttpServlet`).

- Crear una clase que implemente la interfaz javax.servlet.Filter.
- Asociar el filtro al servlet.

Veamos el código necesario para estos pasos. Asumiendo Eclipse como nuestro entorno de desarrollo y Tomcat como contenedor de servlets, lo primero será crear un proyecto web dinámico mediante el asistente de proyectos.

Primero creamos el servlet. Notad que:

- Por sencillez, sólo contemplamos que atienda a peticiones GET.
- Obtiene del parámetro 'response' un objeto PrintWriter para poder generar contenido de salida para el navegador.

ServletMensaje.java

```
package org.xyz.servlets;

import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletMensaje extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, java.io.IOException
    {
        PrintWriter out = response.getWriter();
        out.println("Alguien voló sobre el nido del cuco");
    }
}
```

Ahora es el turno de la creación de los dos filtros de nuestro programa. Comenzamos por el filtro que realiza la auditoría, ya que es el más simple de los dos. No obstante, veamos algunos aspectos a destacar, primero en relación al funcionamiento de los filtros en general y después sobre este filtro en particular:

En el método `init()` tenemos la posibilidad de obtener un objeto `FilterConfig`. Aprovechamos esto para obtener una referencia al contexto servlet (`ServletContext`), que es un espacio de almacenamiento con visibilidad global para una aplicación web.

En nuestro caso, la intención de obtener la referencia al objeto `FilterConfig` es poder utilizar el sistema de log que nos proporciona el contenedor de servlets (Tomcat es nuestro caso). De esta manera conseguimos mostrar por pantalla el recurso web solicitado por el usuario (la URI de nuestro servlet) y el tiempo que le ha llevado al servidor despachar esa petición (esta información también queda almacenada en el fichero de log de Tomcat).

En el método `doFilter` podemos realizar tareas cuando una petición HTTP se dirige al servlet y también cuando la respuesta HTTP vuelve hacia él (siendo poco ortodoxo, los filtros son una cadena que podríamos considerar de ida y vuelta).

La manera de separar estas tareas es invocando a `chain.doFilter()`, así, el código que aparece antes de `chain.doFilter()` se ejecuta en el instante de la petición HTTP mientras que el código que aparece después se ejecuta en la respuesta HTTP. No obstante, hemos de tener en cuenta que no es obligatorio realizar la llamada a `chain.doFilter()`. Omitirla implica que la petición no se pasará a ningún otro filtro de la cadena, por lo que será dirigida directamente al servlet o a cualquier otro componente al que estuviese dirigida la petición. También es importante saber que el orden de encadenamiento de los filtros lo determina su orden de aparición en el descriptor de despliegue (`web.xml`).

En nuestro caso, lo que hacemos en la petición HTTP es obtener y almacenar el instante en que ésta llega al filtro y la URI que ha sido solicitada. A continuación, pasamos la petición HTTP al siguiente filtro en la cadena, que será el filtro de buscar y reemplazar texto que veremos en breve. Cuando el servlet genera la respuesta HTTP para el navegador, ésta llegará de nuevo al filtro de buscar y reemplazar y posteriormente volverá de nuevo al filtro de auditoría, ejecutándose entonces las

líneas de debajo de la llamada `chain.doFilter()`, las cuales se encargan de obtener el instante en que llega la respuesta HTTP y escribir en el log de Tomcat.

FiltroAuditoria.java

```
package org.xyz.filtros;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;

/*
 * Registra en la consola el tiempo que tarda en completarse cada
petición.
 * También muestra el recurso al que se ha accedido.
 */
public class FiltroAuditoria implements Filter {

    private FilterConfig config = null;

    public void init(FilterConfig config) throws ServletException {
        this.config = config;
    }

    public void destroy() {
        config = null;
    }

    public void doFilter(ServletRequest request,
        ServletResponse response,
        FilterChain chain) throws IOException,
        ServletException
```

```

{
    long ini = System.currentTimeMillis();

    String nombre = ((HttpServletRequest)
request).getRequestURI();

    // Pasa al siguiente filtro en la cadena
    chain.doFilter(request, response);

    // El resto de código siguiente se realiza a la vuelta
(response)
    long fin = System.currentTimeMillis();

    config.getServletContext().log(nombre + ": " + (fin - ini)
+ "ms");
}

}

```

Veamos ahora el otro filtro, encargado de modificar la respuesta HTTP que genera el servlet. Después de la explicación anterior sobre el filtro de auditoría, podemos pasar directamente a ver lo que sucede en el método `doFilter()`:

A diferencia del filtro anterior, algunos filtros necesitan alterar la petición (para añadir algún atributo, por ejemplo) y/o la respuesta HTTP. La forma de hacer esto es pasar al servlet un objeto que haga de sustituto del objeto `ServletResponse`. Este sustituto evita que el servlet cierre la respuesta HTTP auténtica una vez realizado su trabajo y así el filtro la pueda modificar. El decorador tiene que sobrescribir los métodos `getWriter()` o `getOutputStream()` para poder devolver el objeto `ServletResponse` original al servlet. Por tanto, esta tarea requiere el uso del patrón Decorador.

Este es el caso de nuestro filtro que, como se comentó anteriormente, realiza una búsqueda de texto en la respuesta HTTP y lleva a cabo la sustitución del mismo. Para ello:

1. A partir de la respuesta HTTP crea una instancia de la clase DecoradorRespuesta. Veremos esta clase posteriormente, por ahora nos es suficiente con saber que se trata de una clase que sigue el patrón de diseño Decorador y que se instancia a partir del objeto ServletResponse (promocionado a HttpServletResponse).
2. Pasa la instancia de DecoradorRespuesta al servlet, en lugar de la respuesta original. Esto lo hace pasando la petición HTTP al siguiente objeto en la cadena, que, al no haber ninguno, llega al servlet.
3. Cuando el servlet termina de escribir en la respuesta decorada, ésta llega de nuevo al filtro y es entonces cuando, una vez obtenidos los parámetros del descriptor de despliegue, tiene lugar el proceso de búsqueda y sustitución del texto. Para obtener el contenido de la respuesta decorada utilizamos el método toString() del decorador. A continuación hacemos la sustitución y la escribimos de nuevo en la respuesta.

FiltroBuscarReemplazar.java

```
package org.xyz.filtros;

import javax.servlet.*;
import javax.servlet.http.*;
import org.xyz.utiles.DecoradorRespuesta;

public class FiltroBuscarReemplazar implements Filter {

    private FilterConfig config;

    public void init(FilterConfig config) {
        this.config = config;
    }
}
```

```

    }

    public void destroy() {
        config = null;
    }

    public void doFilter(ServletRequest request,
                        ServletResponse response, FilterChain chain)
                        throws java.io.IOException,
java.servlet.ServletException
    {
        // Instanciamos el decorador de la respuesta
        DecoradorRespuesta decorador =
new DecoradorRespuesta((HttpServletResponse)
response);

        // Pasar al siguiente filtro
        chain.doFilter(request, decorador);

        // Esto se hace a la vuelta (en la respuesta)
        String strResp = decorador.toString();

        String busqueda = config.getInitParameter("buscar");
        String reemplazar = config.getInitParameter("reemplazar");

        if (busqueda == null || reemplazar == null) {
            config.getServletContext().log("Parametros de
configuracion no establecidos en web.xml");
            return;
        }
    }

```

```

        // Posicion de comienzo la busqueda
        int index = strResp.indexOf(busqueda);

        if (index != -1) { // Si está

            // Cadena justo antes de la palabra buscada
            String strPrevio = strResp.substring(0, index);

            // Cadena justo despues
            String strPosterior =
                strResp.substring(index +
busqueda.length());

            response.getWriter()
                .print(strPrevio + reemplazar + strPosterior);
        }
    }
}

```

Veamos el código para el decorador. Esta clase es una extensión de una clase que nos proporciona el propio API Servlet: `HttpServletResponseWrapper`, la cual nos libera del trabajo de tener que crear una clase base para decorar respuestas HTTP, lo que supondría tener que implementar la interfaz `ServletResponse` y tener que escribir todos sus métodos.

Dado que `HttpServletResponseWrapper` implementa `ServletResponse`, podemos pasar un objeto de nuestra clase `DecoradorRespuesta` a cualquier método que espere un objeto `ServletResponse`. Esta es la razón por la que en la llamada `chain.doFilter()`

podemos pasar `chain.doFilter(request, decorador)` en lugar de `chain.doFilter(request, response)`.

DecoradorRespuesta.java

```
package org.xyz.utiles;
```

```
import java.io.*;
```

```
import javax.servlet.http.*;
```

```
public class DecoradorRespuesta extends HttpServletResponseWrapper {
```

```
    private StringWriter writer = new StringWriter();
```

```
    public DecoradorRespuesta(HttpServletResponse response) {
```

```
        super(response);
```

```
    }
```

```
@Override
```

```
    public PrintWriter getWriter() {
```

```
        return new PrintWriter(writer);
```

```
    }
```

```
@Override
```

```
    public String toString() { return writer.toString(); }
```

```
}
```

Finalmente, veamos el código para el descriptor de despliegue:

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
version="2.5">

    <display-name>Filtros_ChainOfResponsibility</display-name>

    <welcome-file-list>

        <welcome-file>index.html</welcome-file>

    </welcome-file-list>

    <servlet>

        <servlet-name>servletMensaje</servlet-name>

        <servlet-class>org.xyz.servlets.ServletMensaje</servlet-class>

    </servlet>

    <servlet-mapping>

        <servlet-name>servletMensaje</servlet-name>

        <url-pattern>/servletMensaje</url-pattern>

    </servlet-mapping>

    <filter>

        <filter-name>filtroBuscarReemplazar</filter-name>

        <filter-class>org.xyz.filtros.FiltroBuscarReemplazar</filter-
class>

        <init-param>

            <param-name>buscar</param-name>

            <param-value>sobre</param-value>

        </init-param>

        <init-param>

            <param-name>reemplazar</param-name>
```

```
<param-value>bajo</param-value>
```

```
</init-param>
```

```
</filter>
```

```
<filter-mapping>
```

```
  <filter-name>filtroBuscarReemplazar</filter-name>
```

```
  <servlet-name>servletMensaje</servlet-name>
```

```
</filter-mapping>
```

```
<filter>
```

```
  <filter-name>filtroAuditoria</filter-name>
```

```
  <filter-class>org.xyz.filtros.FiltroAuditoria</filter-class>
```

```
</filter>
```

```
<filter-mapping>
```

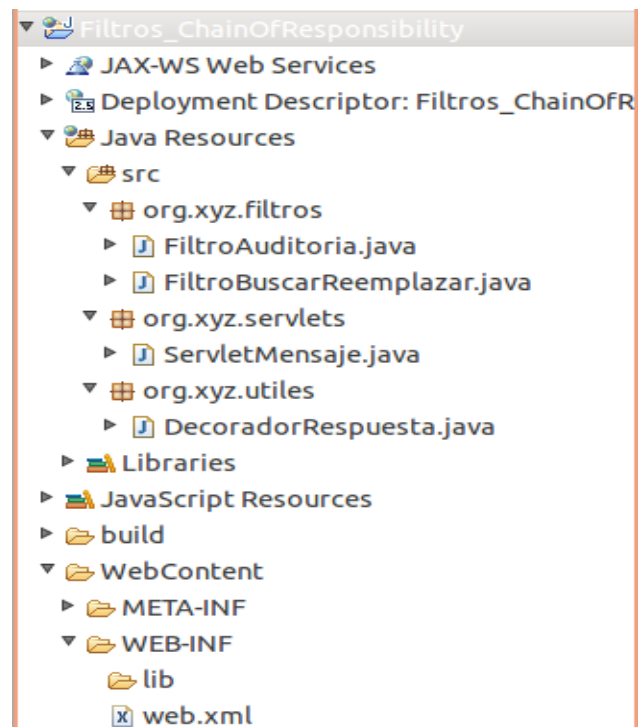
```
  <filter-name>filtroAuditoria</filter-name>
```

```
  <servlet-name>servletMensaje</servlet-name>
```

```
</filter-mapping>
```

```
</web-app>
```

El proyecto tendrá la siguiente disposición de archivos y paquetes:



Salida:



Notad que se ha reemplazado en tiempo de ejecución la palabra 'sobre' por 'bajo'.

Problemas específicos e implementación

Representación de las peticiones

En una aplicación real cada manejador representa un subsistema (contabilidad, compras, etc) y cada subsistema puede manejar peticiones específicas para él o peticiones que son comunes para varios manejadores. Este es un punto clave a tener en cuenta a la hora de implementar el patrón CoR.

Una buena decisión de diseño consistirá en crear una superclase abstracta para la petición, por ejemplo una clase denominada Request, que implementará el comportamiento predeterminado para las diferentes subclases que pueda tener. De esta manera, si necesitamos añadir a la aplicación un nuevo manejador que necesite un tipo específico de petición, todo lo que tenemos que hacer es extender de la clase base Request.

No obstante, una petición modelada como una jerarquía de clases no es la única opción. Supongamos un caso en que una petición contuviera tanta información que no fuese adecuado tenerla encapsulada en un objeto para enviarla como una petición. En este supuesto y asumiendo que los datos de la petición están almacenados en una base de datos, podría ser factible que la petición consistiera únicamente en un identificador a partir del cual los manejadores pudieran recuperar los datos relativos a la petición desde la base de datos.

Peticiones no despachadas

El patrón no garantiza que todas las peticiones sean resueltas. Una petición puede acabar sin ser atendida adecuadamente a causa de que ningún manejador sea el apropiado para satisfacerla. No obstante, cuando esto sucede por un fallo en la implementación de alguno manejador, entonces sí que podemos tener un problema. La moraleja de esto es que se debe poner un cuidado especial en el diseño de las peticiones y manejadores que pueden aparecer en nuestra aplicación.

Cadena rota

Para entender esta variante del patrón utilizaremos la siguiente nomenclatura:

- Handler: Superclase abstracta que define la interfaz para los objetos que manejan peticiones.
- ConcreteHandler: Subclases que implementan a Handler.
- handlerRequest(): Método definido abstracto en Handler e implementado por las subclases ConcreteHandler con la lógica para procesar las peticiones.
- getSuccesor(): Método implementado en Handler y heredado en las subclases ConcreteHandler que retorna una referencia al siguiente manejador de peticiones de la cadena.

Puede ocurrir que en la implementación del método handlerRequest() de alguna de las subclases de Handler olvidemos llamar al método que pasa la petición al siguiente objeto en la cadena:

```
getSuccesor().handlerRequest();
```

En este caso la cadena queda rota, finalizando en esa subclase el flujo de procesamiento de la petición y, consecuentemente, con bastantes probabilidades de no haberla despachado.

Para evitar esta situación existe una variante del patrón que consiste en realizar unas sencillas modificaciones en la superclase Handler y en las subclases ConcretHandler. De hecho, esta variante que vamos a ver debe ser la opción recomendada, a pesar de no ser el diseño original del patrón. La razón es que presenta un diseño más robusto y flexible que el original.

Tomaremos como base el código del primer ejemplo para ilustrar esta refactorización:

Paso 1 - En la clase Handler creamos el método handleRequestImpl() con la siguiente signatura:

```
protected abstract boolean handleRequestImpl(Request request);
```

Notad que este nuevo método:

- Se define como protegido, por lo que el código cliente no lo puede invocar.
- Se define como abstract, por lo que las subclases tendrán que proporcionar una implementación.
- Retorna un booleano, por lo que la implementación en las subclases retornará true o false.

Paso 2 - También en la clase Handler, cambiamos drásticamente la signatura del método handleRequest(), dejando de ser abstract y por tanto, necesitando una implementación, y pasando a ser un método final, por lo que ahora no se podrá sobrescribir en las subclases:

```
public final void handleRequest(Request request) {  
  
    boolean peticionDespachada = this.handleRequestImpl(request);  
  
    if (sucesor != null && !peticionDespachada)  
  
    {  
  
        sucesor.handleRequest(request);  
  
    }  
  
}
```

Notad que hemos movido a la clase base la implementación que las subclases hacían para pasar la petición al siguiente objeto manejador.

Paso 3 - En las subclases ConcreteHandler creamos el método HandlerRequestImpl(), ya que en el paso 1 lo hemos definido como abstracto en Handler. Este método debe tener la misma implementación que el método HandlerRequest() de cada subclase, con la salvedad siguiente:

- El bloque del 'if' retornará *true*
- El bloque del 'else' retornará *false* en lugar de implementar el envío de la petición al siguiente objeto manejador.

```
public boolean handleRequestImpl(Request request) {  
  
    // Si la petición puede ser procesada por esta clase  
  
    if (request.getValor() < 0) {  
  
        System.out.println("Valores negativos son manejados por "  
            + nomClase + ":");  
  
        System.out.println("\t" + nomClase + ".HandleRequest : "  
            + request.getValor());  
  
        return true;  
  
    } else { // Si no, la pasamos al siguiente manejador  
  
        return false;  
  
    }  
  
}
```

Paso 4 - En las subclases ConcreteHandler, eliminamos el método HandleRequest().

En el código cliente no tenemos que hacer nada, ya que no se ve afectado por la refactorización que hemos llevado a cabo. No obstante, sí tenemos que observar que ahora se invoca el método HandleRequest() implementado en Handler, por tanto heredado en cada subclase, el cual se encarga de ejecutar al método HandlerImpl(),

que sí es específico de cada subclase. Para que no quede ninguna duda, a continuación se proporciona el código de la refactorización:

Handler.java

```
package comportamiento.CoR.basico.variante;

public abstract class Handler {
    protected Handler sucesor;

    public Handler getSucesor() {
        return sucesor;
    }

    public void setSucesor(Handler sucesor) {
        this.sucesor = sucesor;
    }

    //public abstract void handleRequest(Request request);
    protected abstract boolean handleRequestImpl(Request request);

    // método final -> no se puede sobrescribir en las subclases
    public final void handleRequest(Request request) {
        boolean peticionDespachada =
            this.handleRequestImpl(request);
        if (sucesor != null && !peticionDespachada)
        {
            sucesor.handleRequest(request);
        }
    }
}
```



```
}
```

Respecto a las subclases, se muestra sólo el caso de la clase manejadora de número negativos, ya que para las otras es prácticamente lo mismo:

ConcreteHandlerNegativos.java

```
package comportamiento.CoR.basico.variante;
```

```
public class ConcreteHandlerNegativos extends Handler {  
    private String nomClase = this.getClass().getSimpleName();  
  
    //public void handleRequest(Request request) {  
    public boolean handleRequestImpl(Request request) {  
  
        // Si la peticion puede ser procesada por esta clase  
        if (request.getValor() < 0) {  
            System.out.println("Valores negativos son manejados  
por "  
                                + nomClase + ":");  
            System.out.println("\t" + nomClase +  
".HandleRequest : "  
                                + request.getValor());  
            return true;  
        } else { // Si no, la pasamos al siguiente manejador  
            //getSucesor().handleRequest(request);  
            return false;  
        }  
    }  
}
```

La refactorización vista, no sólo elimina el problema de la cadena rota, sino que incrementa el grado de flexibilidad del patrón CoR. Por ejemplo, si debido a un cambio de requerimientos necesitáramos que la petición atravesara todos los objetos de la

cadena, independientemente de su valor o estado, tan sólo tendríamos que hacer la siguiente modificación en la clase Handler:

// método final -> no se puede sobrescribir en las subclases

```
/*public final void handleRequest(Request request) {  
  
    boolean peticionDespachada = this.handleRequestImpl(request);  
  
    if (sucesor != null && !peticionDespachada)  
  
    {  
  
        sucesor.handleRequest(request);  
  
    }  
  
}*/  
  
public final void handleRequest(Request request) {  
  
    this.handleRequestImpl(request);  
  
    if (sucesor != null)  
  
    {  
  
        sucesor.handleRequest(request);  
  
    }  
  
}
```

Este supuesto requerimiento no tiene mucho sentido para el ejemplo de los números, pero pensemos otros casos en los que esta última modificación podría ser interesante, por ejemplo:

- Una aplicación que necesita notificar a sus hilos el cierre del programa: todos los hilos del programa deben ver tal notificación para finalizarse.
- Subasta de algún bien: la petición que llega contiene un precio y todos los objetos de la cadena deben verlo.

Patrones relacionados

- CoR envía la petición de un emisor a través de una cadena de potenciales receptores manteniendo un bajo acoplamiento entre el emisor y los receptores. Command, Mediator y Observer son patrones que también se centran en desacoplar a emisores de receptores. Sin embargo, como veremos, cada uno tiene su propia intención.
- CoR puede utilizar el patrón Command para representar las peticiones.
- Chain of Responsibility se utiliza a menudo con el patrón Composite. En este caso, el padre del componente puede actuar como su sucesor. Por ejemplo, en una aplicación de gráficos, supongamos que un objeto línea recibe una petición para retornar el área. Dado que la línea no dispone de superficie, pasará la petición a su objeto sucesor en la cadena (asumamos para este caso un Rectángulo) que además resulta ser su padre (Rectángulo tendrá cuatro objetos Línea hijas).