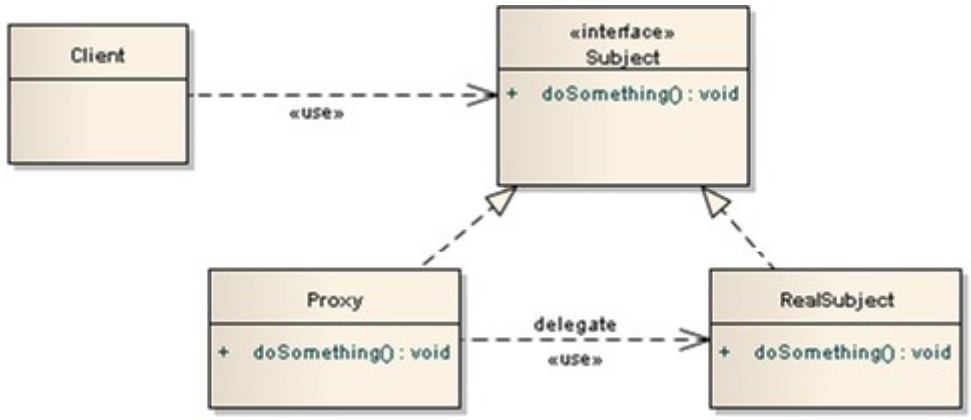
Proxy

Diagrama de clases e interfaces



Intención

Una clase cliente quiere acceder al servicio o funcionalidad que proporciona otra clase (RealSubject). Por algún motivo, como podría ser un mayor control sobre el objeto RealSubject o una mejora en su rendimiento, nos interesa interponer un objeto intermediario entre el objeto Cliente y el RealSubject. Este intermediario es el Proxy, también conocido como Surrogate o Placeholder.

Motivación

Supongamos que estamos diseñando una aplicación que gestiona los stocks de un almacén de expedición de materiales. Los operarios preparan los pedidos de los clientes mediante un terminal móvil que les va indicando paso a paso cómo preparar cada pedido. Por ejemplo, un operario podría visualizar la siguiente pantalla en el transcurso de la preparación de un pedido:

```
                          Operario: 1004 (Carlos García)

Pedido: 2011-NX-1003002   Cliente: 455544 (FatMAX, S.A.)
Ubicación: P04-E026-N08
Cantidad disponible: 6    Cantidad a servir: 5

¿Correcto (s/n)? s
```

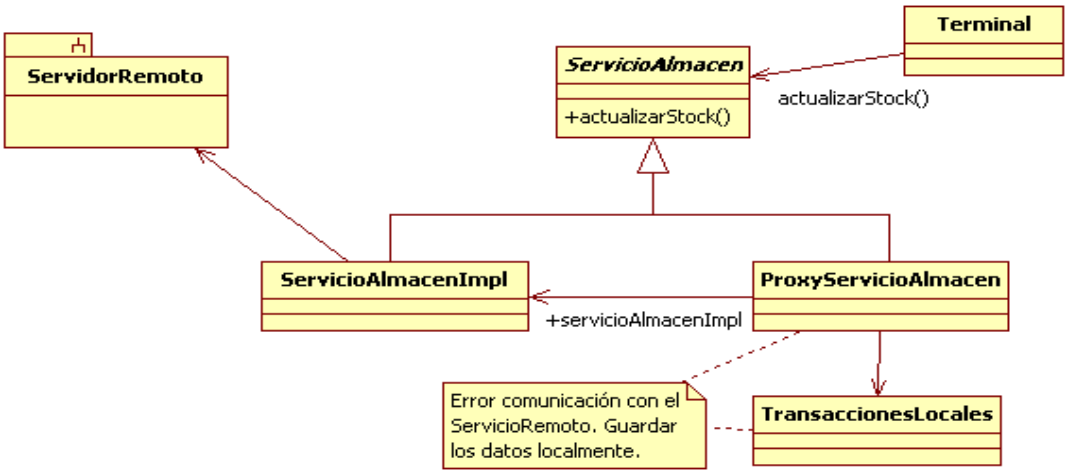
El operario se dirigirá a la ubicación especificada, cogerá 5 artículos, los depositará en su carro de trabajo y confirmará que todo está correcto. A continuación, en el caso de que el pedido tenga otros artículos que servir, aparecerá en la pantalla una nueva ubicación y una nueva cantidad. En caso de que el pedido haya terminado, se comenzará con un nuevo pedido.

> *Para nuestro propósito de estudio del patrón Proxy vamos a seguir el "camino feliz" y no vamos a entrar en qué sucedería si el operario no puede servir las cinco piezas que le indica el sistema porque en la ubicación especificada hubiera sólo cuatro, por ejemplo.*

En cualquier caso, vamos a centrarnos en lo que sucede cada vez que el operario confirma que ha retirado de la ubicación el número de piezas indicado por el programa. La confirmación provoca que el terminal envíe un mensaje a un sistema remoto encargado de actualizar (rebajar) el stock de ese producto (el sistema remoto deduce el producto a partir de su ubicación).

Ahora bien, ¿Qué sucedería si el terminal móvil no pudiera comunicarse con el sistema remoto? Lo adecuado sería no bloquear el terminal a fin de que pudiera seguir trabajando con normalidad. Para ello, se deberían almacenar localmente los datos afectados para poder reenviarlos posteriormente al sistema remoto, una vez que se restablezca la conexión. Aquí es donde el patrón Proxy puede ayudarnos.

La figura siguiente muestra el diseño de clases que resuelve el problema comentado:



Las clases Terminal, TransaccionesLocales y la jerarquía ServicioAlmacen pertenecen

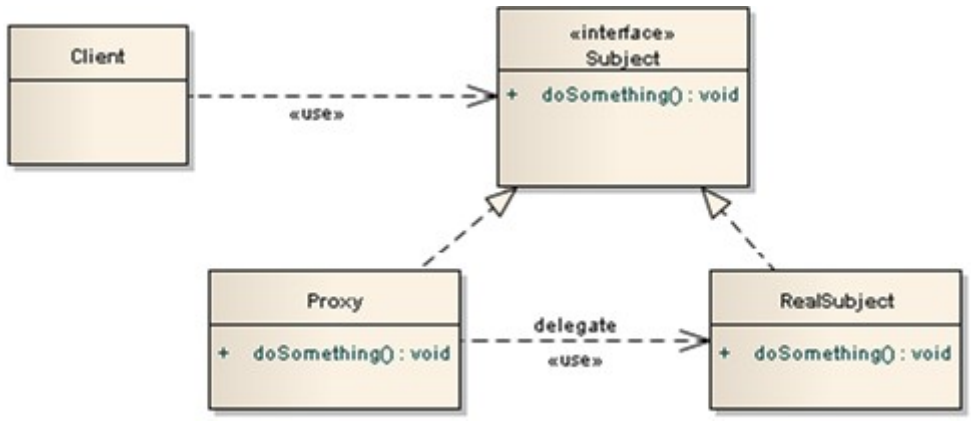
a la aplicación que se ejecuta en el terminal móvil. El subsistema remoto se corresponde a la aplicación que se ejecuta en el servidor de aplicaciones.

El funcionamiento queda como sigue:

La aplicación móvil de terminal se comunicará exclusivamente con ProxyServicioAlmacen, pasándole la información sobre la ubicación del almacén y la cantidad retirada. El proxy delegará la petición en el objeto ServicioAlmacenImpl, encargado de comunicarse con el subsistema ServidorRemoto, responsable de regularizar el stock. Si por cualquier motivo el sistema remoto no responde, ServicioAlmacenImpl retornará un error al proxy, el cual se encargará de invocar a TransaccionesLocales para almacenar la información localmente para su posterior envío.

Implementación

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



Un proxy es simplemente un objeto que implementa la misma interfaz que el objeto al que se quiere acceder en realidad, manteniendo una referencia a él, y controlando su acceso, para mejorarlo de alguna manera (en el ejemplo de los almacenes, la mejora consiste en asegurar la continuidad del servicio a pesar de no estar operativo el subsistema remoto).

Un objeto cliente, como la clase Terminal, no sabe que está referenciando a un proxy, ya que se diseña de manera que piense que está colaborando con el sujeto real.

Las clases participantes en el patrón son las siguientes:

- **Proxy:** Clase que se hace pasar por la clase real. Sostiene una referencia al objeto real (RealSubject), además de tener su misma interfaz. Controla el acceso a dicho objeto real y puede ser el responsable de su creación y borrado. También tiene otras responsabilidades que dependen del tipo de proxy (veremos más sobre esto último).
- **Subject:** Define una interfaz común para el proxy (Proxy) y el objeto real (RealSubject), de tal modo que se puedan usar indistintamente.
- **RealSubject:** Clase del objeto real que el proxy representa.

Aplicabilidad y Ejemplos

El patrón Proxy es adecuado cuando se necesita referenciar a un objeto con mayor versatilidad o sofisticación que la que proporcionaría una (simple) referencia al mismo.

Proxy de Protección. Un proxy para controlar el acceso a un servicio protegido

Debido a que el ejemplo es bastante extenso se ha creado el documento '14-Proxy.Ejemplo aplicacion de Nominas.odt'.

Problemas específicos e implementación

Variantes

Existen diferentes variantes del patrón Proxy. Independientemente de la variante, la estructura de un proxy es siempre la misma; las variantes tienen que ver con lo que hace el proxy una vez que ha sido invocado.

Las variantes más conocidas:

Proxy remoto: permitir llamadas a métodos remotos de manera que resulte simple para el programador de la parte cliente de la aplicación.

Proxy virtual: Mejorar las prestaciones de una aplicación, retrasando el consumo de recursos (CPU, RAM, ...) hasta el momento en que éstos sean realmente necesarios.

Proxy protección: Controlar y mejorar el acceso autorizado a un servicio o recurso.

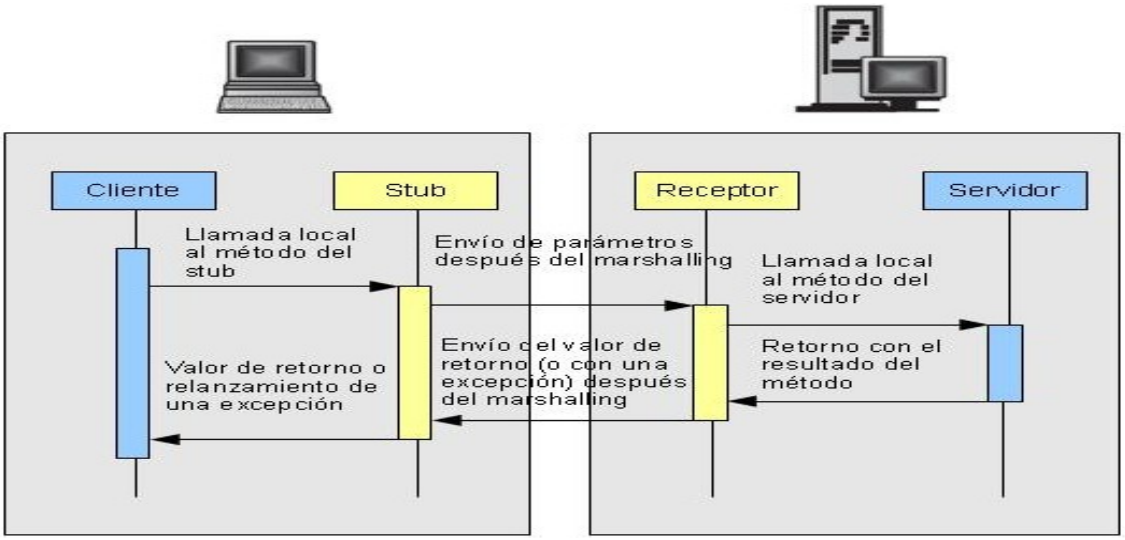
Proxy redirección: Establecer un plan B cuando el plan A falla.

Proxy dinámico: Proxy capaz de sustituir a N RealSubjects.

Proxy remoto

Se trata de un proxy cuya finalidad, dentro de un contexto cliente-servidor, es simplificar la programación de la aplicación del lado del cliente. Por tanto, se quiere realizar una llamada a un método remoto sin tener que ocuparse de los complejos detalles de los protocolos de comunicaciones. Este tipo de proxy oculta el hecho de que un objeto reside en un espacio de direcciones diferente (en otra máquina virtual).

En la tecnología RMI de Java o en CORBA, existe un objeto local del lado del cliente que se denomina stub que sabe cómo comunicarse con el servidor y que aparenta ser el verdadero objeto remoto. La aplicación cliente llama a un método del stub, éste prepara la llamada (marshalling de los parámetros, etc) y envía los datos por la red.



Finalmente se ejecuta el método sobre el objeto remoto (esto ocurre localmente en el servidor). El stub del lado del cliente es un proxy local. Del lado del servidor también hay otro proxy, conocido como skeleton o receptor.

Proxy virtual

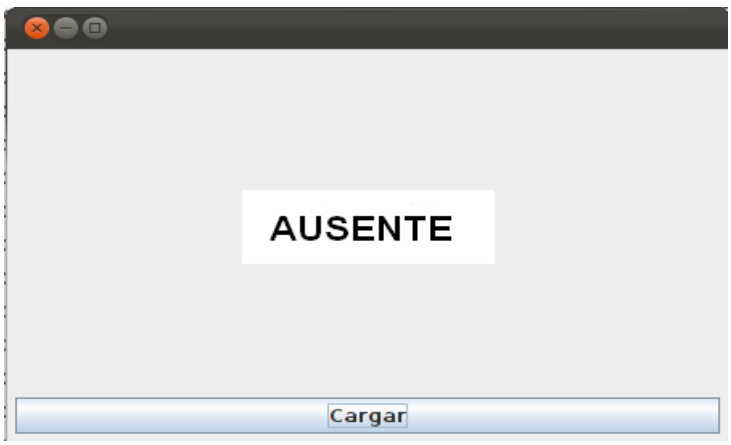
El objetivo de este proxy es mejorar el rendimiento de una aplicación. A veces, una manera de mejorar las prestaciones de un sistema consiste en diferir la creación de objetos computacionalmente costosos hasta que estos sean realmente necesarios para el resto de la aplicación.

Supongamos que tenemos que desarrollar una aplicación (o un módulo de esta) en la que hay que mostrar imágenes de gran tamaño. Este sería un caso típico de una aplicación para una agencia de viajes que muestra a los clientes una serie de hoteles.

Normalmente, las imágenes no tienen que mostrarse justo en el momento en el que el usuario accede a una determinada pantalla, sino cuando éste lo determine, por ejemplo mediante un botón titulado 'mostrar imagen'. Así, continuando con el ejemplo de la agencia de viajes, el usuario del programa (el vendedor) podría encontrarse en una pantalla textual en la que se detallan las características de un hotel; en el momento en que el cliente estuviera interesado en ese hotel y quisiera ver las fotos para decidirse, entonces es cuando se cargarían las imágenes de calidad.

Para evitar la carga de una imagen antes de que ésta sea necesaria, podemos utilizar un proxy (sustituto) para ésta que se encargue de cargarla bajo demanda. En concreto, nuestra estrategia para el programa consistirá en hacer que un objeto proxy muestre inicialmente una imagen muy pequeña indicando que la imagen auténtica está ausente (no cargada). Entonces, cuando se pulse un botón denominado 'cargar', se volverá a mostrar otra imagen, también muy pequeña, denominada 'cargando...' y, simultáneamente, se procederá a la carga de la imagen real, la cual tiene un gran tamaño.

Por tanto, la aplicación nada más iniciar mostrará la imagen de 'ausente'. Esto es muy rápido porque la imagen es pequeña:



Cuando el usuario pulse el botón 'Cargar' se mostrará la imagen 'Cargando...', que también es muy pequeña:

Mientras, en segundo plano, se cargará la imagen de 1,7 Megabytes, en un hilo de ejecución diferente, para no bloquear el programa y permitir que el usuario pueda seguir interactuando con la aplicación (por ejemplo, generar un informe):

Una manera sencilla de mostrar una imagen salvada, por ejemplo, en formato JPEG, es utilizar un objeto de la clase javax.swing.ImageIcon como argumento para una etiqueta (JLabel), ya que las etiquetas son componentes que pueden mostrar imágenes:

ImageIcon icon = new ImageIcon("imagenes/imagen.jpg");

JLabel label = new JLabel(icon);

Sin embargo, en nuestra aplicación queremos pasar a un JLabel un proxy de ImageIcon y no directamente un ImageIcon. Por supuesto, en diferentes momentos del curso del programa, el proxy delegará en ImageIcon, ya que es la clase que realmente implementa el código para tratar con imágenes. Concretamente, al proxy le interesa el objeto Image inherente a ImageIcon.

Por tanto, para configurar un proxy virtual podemos crear una subclase de javax.swing.ImageIcon. A esta subclase la denominaremos en nuestro ejemplo, ProxyImageIcon. Vemos más sobre esto en breve.

Nuestra aplicación consistirá en dos clases: una encargada de soportar la interfaz gráfica (MainGUI) y otra que representa el papel de proxy dentro del patrón Proxy (ProxyImageIcon). Notad que no necesitamos crear clases para otros roles, como Subject o RealSubject, ya que en este ejemplo los proporciona el propio API de Java.

Comencemos por ver código de la clase cliente:

MainGUI.java

```
package estructurales.proxy.virtual.client;


import java.awt.BorderLayout;
import java.awt.EventQueue;


import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JLabel;
```

```
import javax.swing.JButton;
import javax.swing.SwingConstants;

import estructurales.proxy.virtual.ProxyImageIcon;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class MainGUI extends JFrame {

	private static final long serialVersionUID = 1L;

	private JPanel panel = new JPanel();
	private JButton btnCargarImagen = new JButton("Cargar");

	private ProxyImageIcon proxy =
			new
ProxyImageIcon("estructurales/proxy/virtual/recursos/real.jpg");
	private JLabel etiImagen = new JLabel(proxy);

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					new MainGUI().setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
```

```
		});
	}

	/*
	 * Constructor
	 */
	private MainGUI() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 450, 300);

		panel.setBorder(new EmptyBorder(5, 5, 5, 5));
		panel.setLayout(new BorderLayout(0, 0));

		btnCargarImagen.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent evt) {
				cargarImagen();
			}
		});
	btnCargarImagen.setVerticalAlignment(SwingConstants.BOTTOM);

		panel.add("Center", etiImagen);
		panel.add("South", btnCargarImagen);
		setContentPane(panel);
	}

	protected void cargarImagen() {
		proxy.load(this);
		btnCargarImagen.setEnabled(false);
```

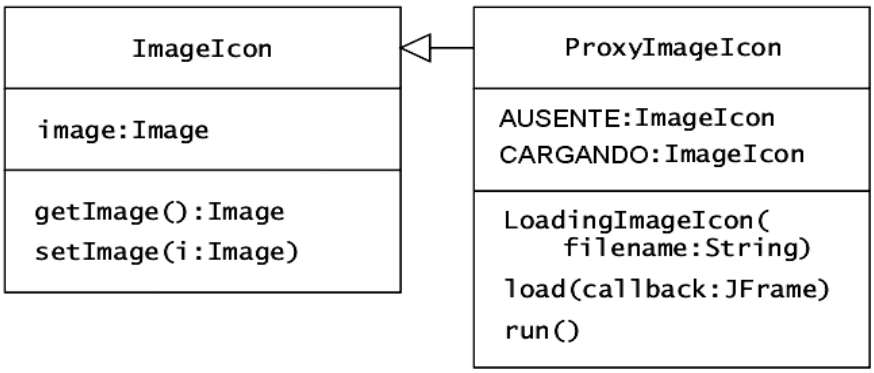
```
        }

}
```

Ahora veamos el código para el proxy virtual, aunque antes, algunos comentarios:

ProxyImageIcon define dos atributos estáticos que contendrán los nombres de los ficheros de imagen 'ausente' y 'cargando...':

```
static final ImageIcon AUSENTE = new ImageIcon(
        ClassLoader.getSystemResource("ruta/ausente.png"));
static final ImageIcon CARGANDO = new ImageIcon(
        ClassLoader.getSystemResource("ruta/cargando.png"));
```

Un objeto ProxyImageIcon puede sustituir a un objeto ImageIcon porque un ProxyImageIcon "es un" ImageIcon.



La clase ProxyImageIcon opera sobre el objeto Image de la clase ImageIcon que sustituye. La anterior figura muestra como la clase ProxyImageIcon tiene dos métodos, a parte del constructor:

- El método load() recibe como argumento el JFrame al que se tendrá que llamar cuando se cargue una imagen. Cuando load() se ejecuta llama a setImage() con la imagen almacenada en CARGANDO como argumento, a continuación repinta el JFrame e inicia un nuevo hilo de ejecución.
- El método run() es el que se ejecuta en el nuevo hilo. Crea un nuevo objeto ImageIcon bajo demanda para cargar la imagen pesada, llama a setImage() con esta imagen y reajusta el JFrame al tamaño de ésta.

Veamos el código:

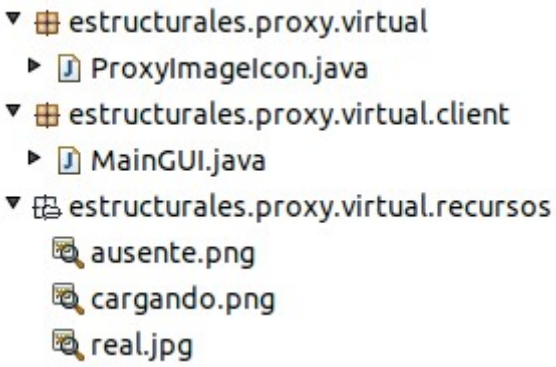
ProxyImageIcon.java

```
package estructurales.proxy.virtual;


import javax.swing.ImageIcon;
import javax.swing.JFrame;


/*
 * Proxy.
 * Esta clase sustituye a un objeto ImageIcon que puede tener tres
 * imágenes: 'ausente', 'cargando' y la imagen de destino (la real)
 */
public class ProxyImageIcon extends ImageIcon implements Runnable {
	private static final long serialVersionUID = 1L;


	static final ImageIcon AUSENTE = new ImageIcon(


ClassLoader.getSystemResource("estructurales/proxy/virtual/recursos/
ausente.png"));


	static final ImageIcon CARGANDO = new ImageIcon(


ClassLoader.getSystemResource("estructurales/proxy/virtual/recursos/
cargando.png"));


	protected String REAL;
	protected JFrame mainGUI;


	/*
```

```
         * Constructor.
         * Carga la imagen 'ausente' y almacena el nombre del fichero
         * de la imagen real para cuando sea necesario cargarla
         */
        public ProxyImageIcon(String nomFicheroImagenReal) {
              // Establecer la imagen que debe mostrar el
              // RealSubject de forma predeterminada
              super(AUSENTE.getImage());
              // Almacenar el nombre del fichero de la imagen real
              this.REAL = nomFicheroImagenReal;
        }

        /*
         * Cargar la imagen real.
         *
         * @param mainGUI: es el JFrame que se debe repintar al
         * cambiar de imagen, tanto de 'ausente' a 'cargando,
         * como de 'cargando' a 'REAL'
         */
        public void load(JFrame mainGUI) {
              this.mainGUI = mainGUI;
              // Establecer la imagen que debe mostrar el RealSubject
              // cuando el usuario ha pulsado el botón de cargar imagen
              setImage(CARGANDO.getImage());
              mainGUI.repaint(); // forzamos el repintado de la ventana
principal

              // Arrancamos en un nuevo hilo la carga de la imagen real
              new Thread(this).start();
```

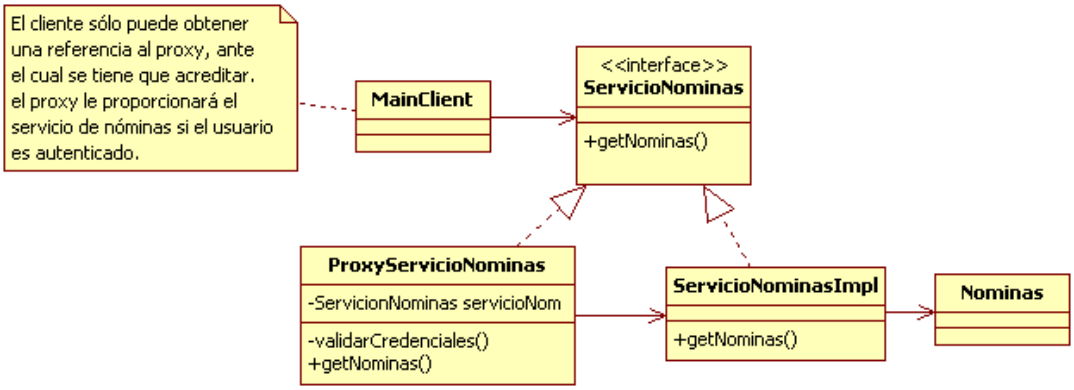
```
        }


        /*
         * Cargar en un hilo separado la imagen real.
         */
        @Override
        public void run() {
              setImage(
                          new ImageIcon(
                                      ClassLoader.getSystemResource(REAL)
                          ).getImage());
              mainGUI.pack();
        }
}
```

La disposición de clases y ficheros de imagen debe quedar como se muestra a continuación:



Proxy de protección

Comprueba que el cliente tiene los permisos necesarios para realizar la petición solicitada. Para ver un ejemplo completo de esta variante podemos consulta el primer ejemplo de la sección *Aplicabilidad y ejemplos*. A modo de recordatorio, se muestra a continuación un diagrama simplificado del diseño seguido en ese ejemplo.



Proxy de Redirección

En el ejemplo de los almacenes mostrado en la sección *Motivación* hemos utilizado la variante conocida como Proxy de Redirección (Redirection Proxy), también conocido como Proxy de Mantenimiento de Servicio (Failover Proxy). Esta variante proporciona un plan de contingencia para cuando las cosas no salen como estaban previstas.

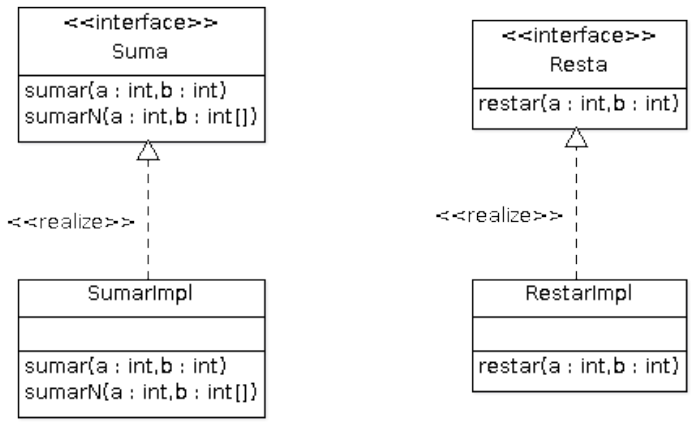
Proxy dinámico

A partir de JSE 1.3, incluido en el API Reflection, Java trae la funcionalidad de crear Dynamic Proxies, es decir objetos que se generan dinámicamente para suplantar el comportamiento de otros. Esto permite disponer de una única clase proxy capaz de monitorizar cualquier tipo de clase, lo cual nos proporciona mucha funcionalidad a partir de un mínimo de código. Esta técnica es ideal para solucionar aspectos transversales, como la seguridad, la auditoria (logs) o las transacciones de bases de datos, entre otros.

No obstante, los proxies dinámicos de Java presentan una limitación importante: toda clase que queramos monitorizar debe implementar una interfaz. Si el RealSubject no implementa una interfaz entonces este tipo de proxies no nos sirve, teniendo que recurrir a librerías de terceros, como CGLIB, ASM, o Javassist.

Veamos un ejemplo:

Consideremos dos clases, SumaImpl y RestaImpl, que definen métodos para realizar operaciones aritméticas elementales.



La clase RestaImpl proporciona una única operación que realiza la resta de dos números. En cambio, la clase SumaImpl proporciona dos métodos para sumar números. El primer método, suma(), nos devuelve la suma de los dos argumentos pasado en la llamada. El otro método, sumarN(), tiene un número variable de parámetros y retorna el sumatorio de todos ellos.

La clase SumaImpl implementa la interfaz Suma y la clase RestaImpl implementa la interfaz Resta.

Supongamos ahora que fuese necesario realizar alguna tarea antes, durante y después de la ejecución del método sumarN() y restar(). Notad que sumarN() corresponde a SumarImpl y que restar() a pertenece a RestarImpl. Asimismo, fijaos que no se dice nada sobre el método sumar(), por lo que no hay que monitorizarlo. La tarea que se tendría que hacer podría tratarse de mostrar información de auditoría por pantalla, del estilo “entrando en el método”, “saliendo del método”, etc.

Según lo comentado en el párrafo anterior, necesitamos monitorizar dos clases: SumaImpl y RestaImpl. Una manera elegante de solventar los requerimientos sería utilizar un proxy basado en los proxies dinámicos de Java, dado que este tipo de objetos pueden monitorizar los métodos de clases arbitrarias. Por tanto, podemos crear una clase proxy que realice el trabajo que queremos. Luego desde una clase cliente utilizaremos el proxy para suplantar a un objeto suma y a un objeto Resta y Java se encarga de que esto funcione.

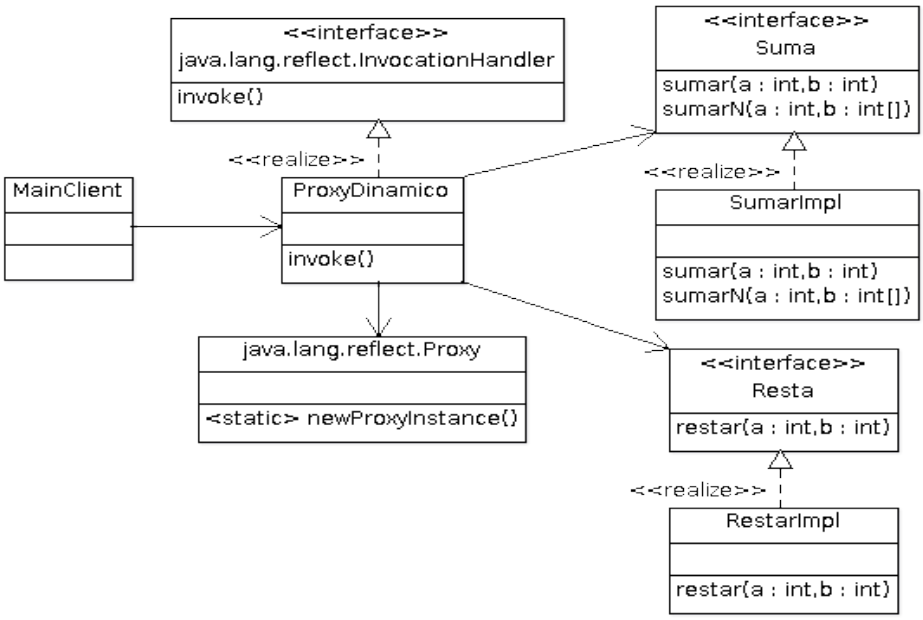
Java no proporciona un mecanismo directo y sencillo para crear un proxy dinámico, sino que nos ofrece la interfaz java.lang.reflect.InvocationHandler y la clase java.lang.reflect.Proxy. No obstante, mediante estos dos artefactos ya podemos crearnos nuestro propio proxy.

La interfaz InvocationHandler define el método invoke(), el cual permite invocar a cualquier método de cualquier clase (que desempeñe el papel de RealSubject dentro del patrón Proxy). Es en este método donde podemos escribir el código que queremos que se ejecute antes o después de la ejecución del método del RealSubject. Además. invoke() retorna el resultado devuelto por el método invocado.

La clase Proxy proporciona el método estático newProxyInstance(). Este método devuelve un proxy de Java, adecuadamente configurado para invocar los métodos del RealSubject.

La idea es que cuando se produzca una llamada sobre un objeto de interés, Java llamará al método invoke() de nuestro proxy para que realicemos las tareas que deseemos. Por nuestra parte nosotros tenemos que llamar desde el cuerpo de invoke() al método del objeto real y devolver el resultado de esta ejecución.

La figura siguiente presenta el diseño de clases de la aplicación de ejemplo que vamos a analizar a continuación:



Veamos el código:

Suma.java

```
package estructurales.proxy.dinamico.java;

/*
 * Interfaz que tiene que implementar el RealSubject
 */
public interface Suma {
	public int sumar(int a, int b);
	public int sumarN(int a, int... b);
}
```

Resta.java

```
package estructurales.proxy.dinamico.java;

/*
 * Interfaz que tiene que implementar un RealSubject
 */
public interface Resta {
	public int restar(int a, int b);
}
```

Ahora vamos con las implementaciones de las interfaces anteriores.

SumaImpl.java

```
package estructurales.proxy.dinamico.java;

/*
 * RealSubject: objeto que implementa la interfaz Suma y
 * que será sustituido (proxeado)
 */
public class SumaImpl implements Suma {
```

```
	@Override
	public int sumarN(int a, int... b) {

		int tmp = 0;
		for (int i=0; i<b.length; i++) {
			tmp += b[i];
		}
		return a + tmp;
	}

	@Override
	public int sumar(int a, int b) {
		return a + b;
	}
}
```

RestaImpl.java

```
package estructurales.proxy.dinamico.java;

/*
 * RealSubject: objeto que implementa la interfaz Resta y
 * que será sustituido (proxeado)
 */
public class RestaImpl implements Resta {

	@Override
	public int restar(int a, int b) {
		return a - b;
	}
```

```
}
```

Veamos a continuación el código para nuestro proxy. Notad que la clase contiene un atributo de tipo Object llamado 'realSubject'. Pero, ¿por qué es de tipo Object? Respuesta:

Una clase que implemente el patrón Proxy siempre contiene un atributo del tipo de la clase que sustituye. La particularidad de este atributo cuando se emplean proxies dinámicos de Java es que puede ser que necesitemos que su tipo sea lo suficientemente flexible como para poder representar en tiempo de ejecución a cualquiera de las clases RealSubject. Este es nuestro caso, ya que necesitamos que pueda corresponderse con un objeto SumaImpl o RestaImpl.

El código está bastante comentado, leedlo con detenimiento:

ProxyDinamico.java

```
package estructurales.proxy.dinamico.java;

import java.lang.reflect.*;
import javax.naming.OperationNotSupportedException;
/*
 * Proxy. Esta clase monitoriza las llamadas efectuadas a los métodos
 * del objeto real.
 * Podemos realizar alguna tarea antes y despues de la ejecucion
 * del método invocado, incluso variar los argumentos enviados al
 * método o modificar el resultado de su ejecucion para que retorne
 * un valor alterado.
 */
public class ProxyDinamico implements InvocationHandler {

	private Object realSubject;

	public ProxyDinamico(Object realSubject) {
		this.realSubject = realSubject;
```

```
	}


	/*
	 * Lo queremos hacer cada vez que se llame a algún método de
alguna clase "RealSubject".
	 */
	@Override
	public Object invoke(Object proxy, Method method, Object[] args)
			throws Throwable
	{
		String metodo = method.getName();
		print("Entrando en el metodo " + metodo);


		// Tenemos control para decidir qué métodos de un objeto
se van a monitorizar
		// en este caso exclusivamente los metodos sumarN() y
restar() pero no suma()
		if (metodo.equals("sumarN") ||
				metodo.equals("restar"))
		{
			// Tenemos acceso a los parametros
			//System.out.println("parametro: " + args[0]);
			mostrarParametros(args);
			// Llamamos al RealSubject
			Object resultado = method.invoke(realSubject, args);


			print("resultado = " + resultado);
			print("Saliendo del metodo " + metodo);


			// Devolvemos el resultado retornado por el
RealSubject aunque
```

```
                // podriamos alterarlo.
                return resultado;
            } else
                throw new OperationNotSupportedException("Metodo no
soportado por el proxy");
      }

      private void mostrarParametros(Object[] args) {
            print("parametro num. 1: " + args[0]);
            if (args.length == 2 &&
args[1].getClass().getSimpleName().equals("int[]"))
            {
                int[] segundoParam = (int[]) args[1];
                for (int i = 0; i < segundoParam.length; i++) {
                      print("parametro num. " + (i+2) + ": " +
segundoParam[i]);
                }
            } else {
                print("parametro num. 2: " + args[1]);
            }

      }

      /*
       * Método de conveniencia para hacer que sea más fácil para las
       * clases cliente obtener un objeto Proxy de Java
       */
      public static Object newProxyInstance(Object realSubject) {

            Class<?> interfaz = realSubject.getClass().getInterfaces()
[0];
```

```
            return Proxy.newProxyInstance(
                  interfaz.getClassLoader(),
                  new Class[] { interfaz },
                  new ProxyDinamico(realSubject)
            );
      }

      private static void print(String texto) {
            System.out.println(texto);
      }
}
```

Comentarios sobre el método newProxyInstance()

Este método encapsula la tediosa manera en la que se debe configurar un proxy dinámico de Java. De esta forma es más fácil para una clases cliente la obtención de un objeto ProxyDinamico correctamente configurado en tiempo de ejecución para invocar a los métodos del RealSubject que corresponda. El código cliente tan sólo tendrá que pasar como argumento una instancia de SumaImpl o de RestaImpl y obtendrá un ProxyDinamico listo para monitorzar las invocaciones sobre un SumaImpl o un RestaImpl, según corresponda.

Para crear el proxy Java necesitamos llamar a la operación Proxy.newProxyInstance, la cual recibe 3 parámetros:

- El cargador de clases de la clase de la que queremos hacer el proxy.
- Un array con la interfaz o interfaces que implementa la clase del objeto cuyos métodos que queremos interceptar.
- El objeto que realizará el trabajo al rededor de la invocación del método o los métodos del RealSubject. Este objeto recibirá la retrollamada por parte del proxy. En nuestro caso este objeto es la propia clase ProxyDinamico.

Por último, veamos el código cliente:

MainClient.java

```
package estructurales.proxy.dinamico.java.client;

import estructurales.proxy.dinamico.java.*;

/*

 * Ejemplo de como se un proxy dinámico de Java puede sustituir a
diferentes objetos.

 * Primero se muestra el caso de utilizar un servicio de manera
normal, sin proxy.

 * A continuación vemos cómo se utiliza el proxy dinámico  para
realizar cierto

 * procesamiento antes, durante y después de la llamada al método del
servicio Suma.

 * Finalmente vemos cómo el mismo proxy se puede utilizar para
realizar cierto

 * procesamiento antes, durante y después de la llamada al método del
servicio Resta.

 */

public class MainClient {


      public static void main(String[] args) {


            print("-->Uso normal del servicio ofrecido por una
clase:");


            Suma objSuma = new SumaImpl();

            int total = 0;

            total = objSuma.sumarN(2 , 3,  4);

            print("La suma es " + total);
```

```
            //
**********************************************************************


            print("\n-->Uso del proxy dinamico para acceder al
servicio de Suma:");

            objSuma = (Suma) ProxyDinamico.newProxyInstance(new
SumaImpl());

            total = objSuma.sumarN(8 , 4, 2, 9);

            print("La suma es " + total);


            //
**********************************************************************


            print("\n-->Uso del mismo tipo de proxy dinamico para
acceder al servicio de Resta:");

            Resta objResta = (Resta)
ProxyDinamico.newProxyInstance(new RestaImpl());


            total = objResta.restar(2 , 2);

            print("La resta es " + total);

      }


      private static void print(String texto) {

            System.out.println(texto);

      }


}
```

Salida:

```
-->Uso normal del servicio ofrecido por una clase:
La suma es 9

-->Uso del proxy dinamico para acceder al servicio de Suma:
Entrando en el metodo sumarN
parametro num. 1: 8
parametro num. 2: 4
parametro num. 3: 2
parametro num. 4: 9
resultado = 23
Saliendo del metodo sumarN
La suma es 23

-->Uso del mismo tipo de proxy dinamico para acceder al servicio de Resta:
Entrando en el metodo restar
parametro num. 1: 2
parametro num. 2: 2
resultado = 0
Saliendo del metodo restar
La resta es 0
```

Ahora bien, ¿y si la clase o las clases cuyos métodos queremos interceptar no implementarán ninguna interfaz? En ese caso, por cierto muy común, tendríamos que recurrir a alguna librería especializada de terceros, por ejemplo CGLIB.

CGLIB es una librería de macros basada en ASM, que es una librería de manipulación de bytecode Java. Dentro de CGLIB se pueden encontrar diversas funciones de utilidad; en nuestro caso vamos a usar una clase llamada net.sf.cglib.proxy.Enhancer.

Con CGLIB no hace falta ninguna interfaz para usar la clase Enhancer. Enhancer requiere que:

- Mediante el método setSuperClass() establezcamos la clase del RealSubject cuyos métodos se tienen que interceptar.
- Con el método setCallBack() fijemos una instancia del objeto que queremos que realice el trabajado de monitorización sobre el RealSubject, esto es, nuestro proxy.

Vamos el código anterior utilizando ahora CGLIB. Notad que ya no necesitamos interfaces para SumaImpl y RestaImpl.

SumaImpl.java

```java
package estructurales.proxy.dinamico.cglib;

/*
 * RealSubject: objeto que será sustituido (proxeado)
 */
public class SumaImpl {

	public int sumarN(int a, int... b) {

		int tmp = 0;
		for (int i=0; i<b.length; i++) {
			tmp += b[i];
		}
		return a + tmp;
	}

	public int sumar(int a, int b) {
		return a + b;
	}
}
```

RestaImpl.java

```java
package estructurales.proxy.dinamico.cglib;

/*
 * RealSubject: objeto que será sustituido (proxeado)
 */
public class RestaImpl {

	public int restar(int a, int b) {
```

```
            return a - b;
      }
}
```

ProxyDinamico.java

```
package estructurales.proxy.dinamico.cglib;

import java.lang.reflect.*;

import javax.naming.OperationNotSupportedException;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

/*
 * Proxy.
 * Esta clase implementa la interfaz MethodInterceptor de CGLIB,
 * lo cual hace que deba implementar el método intercept(). Este
 * método es similar al método invoke() de la interfaz
 * java.lang.reflect.InovationHandler.
 *
 * monitoriza las llamadas efectuadas a los métodos
 * del objeto real.
 * Podemos realizar alguna tarea antes y despues de la ejecucion
 * del método invocado, incluso variar los argumentos enviados al
 * método o modificar el resultado de su ejecucion para que retorne
 * un valor alterado.
 */
```

```
public class ProxyDinamico implements MethodInterceptor {


      public Object intercept(Object realSubject, Method met,
                  Object[] args, MethodProxy methodProxy)
                   throws Throwable
    {
            String metodo = met.getName();
            print("Entrando en el metodo " + metodo);


            // Tenemos control para decidir qué métodos de un objeto
se van a monitorizar
            // en este caso exclusivamente los metodos sumarN() y
restar() pero no suma()
            if (metodo.equals("sumarN") ||
                        metodo.equals("restar"))
            {
                  // Tenemos acceso a los parametros
                  //System.out.println("parametro: " + args[0]);
                  mostrarParametros(args);
                  // Llamamos al RealSubject
                  Object resultado =
methodProxy.invokeSuper(realSubject, args);


                  print("resultado = " + resultado);
                  print("Saliendo del metodo " + metodo);


                  // Devolvemos el resultado retornado por el
RealSubject aunque
                  // podriamos alterarlo.
                  return resultado;
            } else
```

```
                throw new OperationNotSupportedException("Metodo no
soportado por el proxy");


    }


      private void mostrarParametros(Object[] args) {

            print("parametro num. 1: " + args[0]);

            if (args.length == 2 &&
args[1].getClass().getSimpleName().equals("int[]"))

            {

                  int[] segundoParam = (int[]) args[1];

                  for (int i = 0; i < segundoParam.length; i++) {

                        print("parametro num. " + (i+2) + ": " +
segundoParam[i]);

                  }

            } else {

                  print("parametro num. 2: " + args[1]);

            }

      }


      /*

       * Método de conveniencia que proporciona un proxy para la clase
suministrada

       * por parámetro. Notad que se utiliza la clase Enhancer de
CGLIB, a la cual:

       *

       * -Mediante el método setSuperClass() se le establece la clase
del RealSubject

       * cuyos métodos se tienen que interceptar.

       *

       * -Con el método setCallBack() se establece una instancia del
objeto que
```

```
	 * queremos que realice el trabajado de monitorización sobre el
RealSubject,
	 * esto es, nuestro proxy.
	 */
	public static Object createProxy(Class<?> claseRealSubject) {
		Enhancer e = new Enhancer();
		e.setSuperclass(claseRealSubject);
		try {
			e.setCallback(ProxyDinamico.class.newInstance());
		} catch (InstantiationException ex1) {
			ex1.printStackTrace();
		} catch (IllegalAccessException ex1) {
			ex1.printStackTrace();
		}
		return e.create();
	}


	private static void print(String texto) {
		System.out.println(texto);
	}
}
```

MainClient.java

```
package estructurales.proxy.dinamico.cglib.client;

import estructurales.proxy.dinamico.cglib.*;

/*
```

```
 * Ejemplo de uso de proxy dinámico mediante la librería CGLIB. Con
 * CGLIB nuestros RealSubjects no tienen obligatoriamente que
implementar
 * interfaz alguna.
 * Notad que son necesarias las librerias asm.jar y cglib-x.x.x.jar.
 */
public class MainClient {

      public static void main(String[] args) {

            print("-->Uso normal del servicio ofrecido por una
clase:");

            SumaImpl objSuma = new SumaImpl();
            int total = 0;
            total = objSuma.sumarN(2 , 3,  4);
            print("La suma es " + total);

            //
**************************************************************

            print("\n-->Uso del proxy dinamico para acceder al
servicio de Suma:");
            objSuma = (SumaImpl)
ProxyDinamico.createProxy(SumaImpl.class);
            total = objSuma.sumarN(8 , 4, 2, 9);
            print("La suma es " + total);

            //
**************************************************************
```

```
            print("\n-->Uso del mismo tipo de proxy dinamico para
acceder al servicio de Resta:");

            RestaImpl objResta = (RestaImpl)
ProxyDinamico.createProxy(RestaImpl.class);


            total = objResta.restar(2 , 2);

            print("La resta es " + total);

      }


      private static void print(String texto) {

            System.out.println(texto);

      }


}
```

Patrones relacionados

Adapter: un adaptador proporciona una interfaz diferente que la que tiene el objeto al que adapta. Un proxy, en cambio, proporciona la misma interfaz que la del objeto al que sustituye. No obstante, un proxy podría presentar una interfaz ligeramente diferente que la del objeto que sustituye, como podría ser el caso de un proxy de protección que impidiese llevar a cabo una operación para lo que no se estuviera acreditado.

Decorator: A pesar de que un decorador puede tener una implementación parecida a la de un proxy, un decorador tiene un propósito distinto: añadir una o más responsabilidades a un objeto. Por contra, un proxy controla el acceso a un objeto.

Hay que resaltar que el hecho de que la implementación de un proxy se asemeje a la de un decorador varía según la variante de proxy. Por ejemplo, un proxy de protección podría implementarse exactamente como un decorador. Sin embargo, un proxy remoto nunca contendrá directamente una referencia al objeto que sustituye, sino una referencia indirecta, como “ID host y dirección local en el host”. Por otro lado, un proxy virtual comienza teniendo una referencia indirecta, por ejemplo, el nombre del fichero

de una imagen, para pasar en algún momento a tener una referencia directa a la imagen (siguiendo con el ejemplo de imágenes).