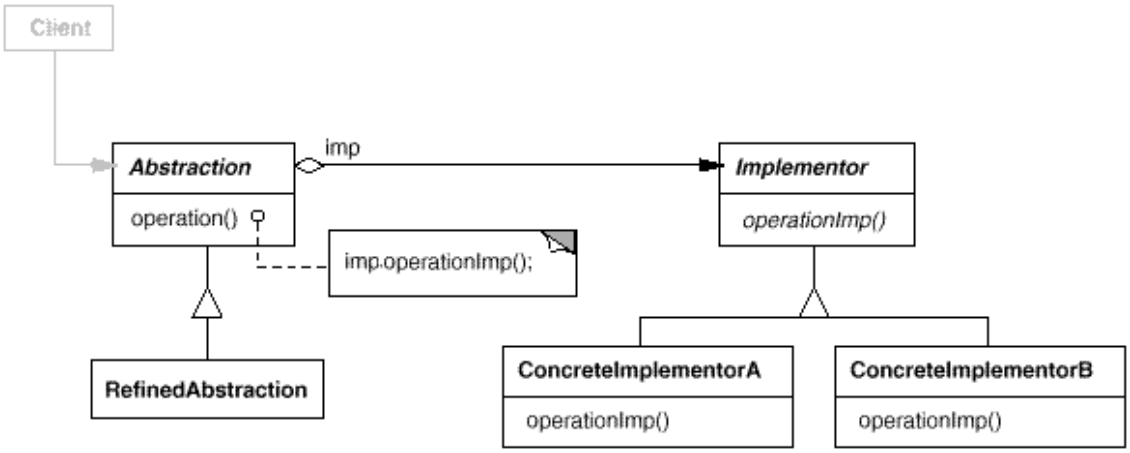
Bridge

Diagrama de clases e interfaces



Intención

La finalidad del patrón de diseño Bridge (Puente), también conocido como Handle/Body, es desacoplar una abstracción de su implementación, de manera que los dos puedan variar independientemente.

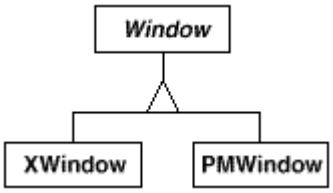
Motivación

Cuando una abstracción puede tener varias implementaciones, la Herencia suele ser la forma más habitual de disponer estas clases. Se crea una clase abstracta para definir la interfaz de la abstracción y se crean varias clases concretas que la implementan. Sin embargo, esta solución no es siempre lo flexible que sería necesario. La Herencia vincula una implementación a una determinada abstracción de manera permanente, lo cual hace que sea difícil modificar, extender y reutilizar abstracciones e implementaciones de forma independiente.

Consideremos el caso del diseño de una librería de componentes gráficos para la creación de interfaces de usuario (GUI). Uno de los requerimientos del cliente es relativo a la portabilidad: la librería tiene que diseñarse de manera abstracta para que pueda utilizarse tanto por programadores que trabajen en un entorno gráfico *X*

Window System como por programadores que lo trabajen en un entorno gráfico IBM Presentation Manager (PM).

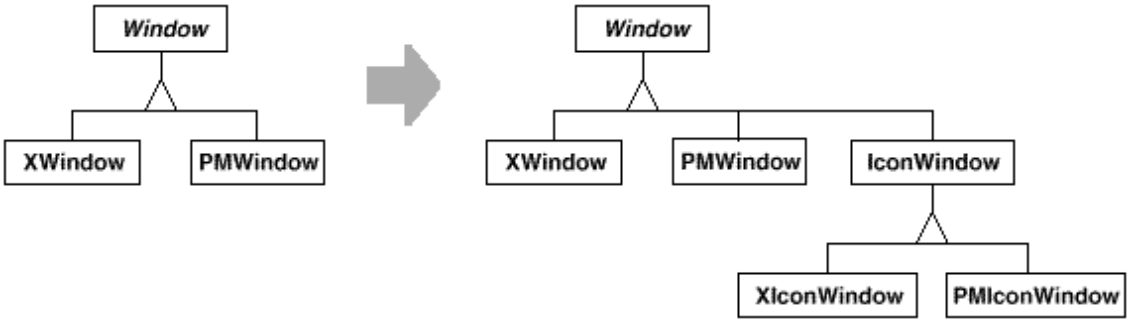
Utilizando la Herencia, podríamos definir una clase abstracta Window y las subclases XWindow y PMWindow, que implementarían la interfaz definida por Window para las respectivas plataformas, tal y como se muestra en la siguiente figura:



No obstante, esta aproximación tiene dos inconvenientes:

Primer problema: proliferación de clases

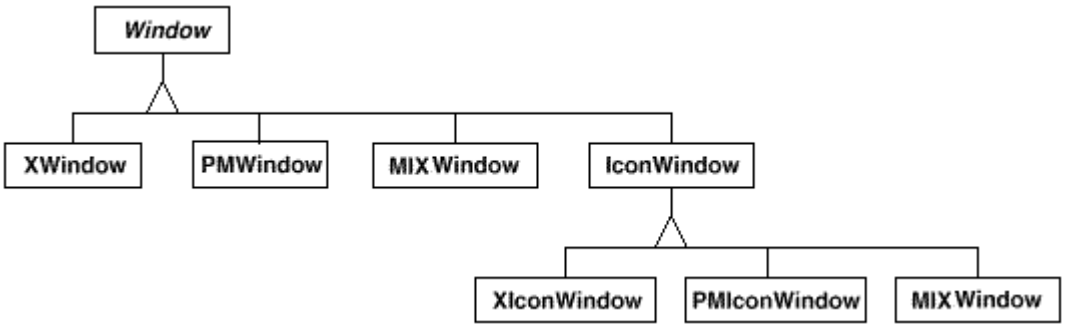
No es conveniente extender la abstracción Window para proporcionar la implementación de diferentes tipos de ventanas o nuevas plataformas. Para entender cuál es el problema, imaginemos una subclase de Window llamada IconWindow, que especializa la abstracción de una ventana para contemplar iconos. Para que la aplicación admita IconWindow en las dos plataformas, tenemos que implementar dos nuevas clases: XIconWindow y PMIconWindow. Esto es lo que ilustra la siguiente figura:



Por tanto:

- Tendremos que crear dos clases de implementación por cada tipo de ventana. Notad que de momento tenemos definidas dos abstracciones de ventanas y ya tenemos que crear cuatro implementaciones.
- En el caso de tener que admitir una tercera plataforma, sería necesario crear una nueva subclase (una nueva implementación) para cada tipo de ventana. La

siguiente figura muestra este caso, donde podemos ver el resultado de introducir una hipotética plataforma llamada MIX:



¿Qué pasa si tenemos tres tipos de ventanas y cuatro plataformas? ¿Qué pasa si tenemos cinco tipos de ventanas, y diez plataformas? El número de clases de implementación que tendríamos que definir es el producto del número de ventanas por el número de plataformas.

Segundo problema: se promueve a que el código de las clases cliente sea dependiente de la plataforma

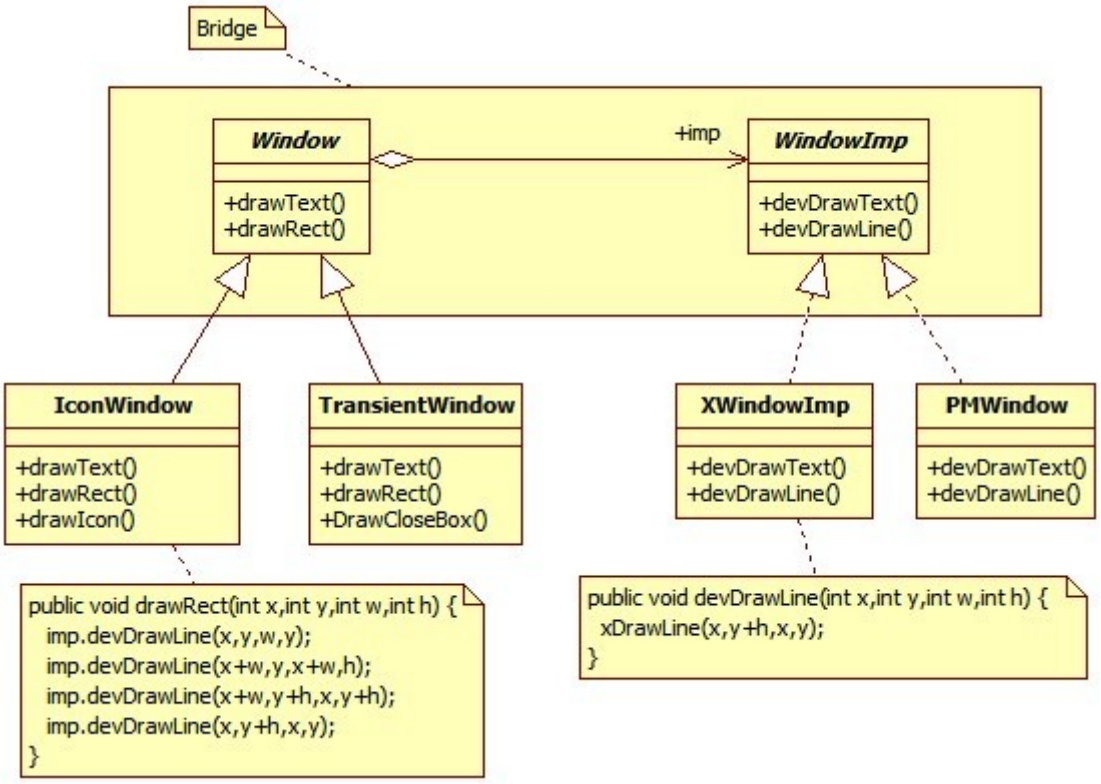
La disposición de clases que estamos comentando hace que las clases cliente sean dependientes de la plataforma. Por ejemplo, crear un objeto XWindow vincula la abstracción Window a la implementación XWindow, lo cual provoca que el código cliente sea dependiente de la implementación XWindow y, consecuentemente, que no pueda portarse a otra plataforma.

Las clases cliente tienen que ser capaces de crear una ventana sin comprometerse a usar una implementación concreta. Solamente la implementación de una ventana debería depender de la plataforma sobre la que se ejecuta la aplicación. Por tanto, el código cliente debería instanciar ventanas sin mencionar plataformas específicas.

Solución

El patrón Bridge soluciona estos problemas separando la abstracción Window y sus implementaciones en dos jerarquías de clases paralelas. Según esto, tendremos una jerarquía de clases para las abstracciones de ventana (Window, IconWindow, TransientWindow) y otra jerarquía para las implementaciones de ventana dependientes de la plataforma, siendo WindowImp la raíz de la jerarquía. La subclase XWindowImp, por ejemplo, proporciona una implementación basada en X Window System.

La figura siguiente ilustra esto:



Las operaciones de las subclases de Window carecen de implementación propia y sencillamente delegan en las operaciones abstractas de la clase (o interfaz) WindowImp. Esto desacopla las abstracciones de ventana de las diferentes implementaciones de las plataformas. No obstante, algunas de las operaciones de las subclases de Window sí tienen implementación porque forman para de la abstracción y no existe contrapartida alguna en la jerarquía de implementación.

Nos referimos a la relación entre Window y WindowImp como un puente (bridge), ya que permite el paso desde la abstracción a la implementación correspondiente, permitiendo que ambas evolucionen de manera independiente.

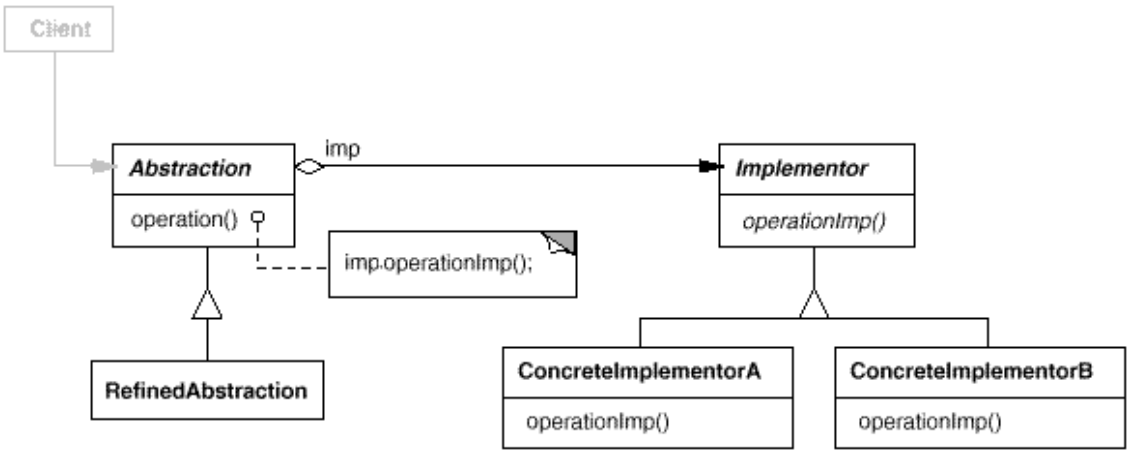
Por ejemplo, si ahora fuese necesario contemplar la plataforma MIX vista anteriormente, dado que se trataría de un problema que afecta exclusivamente a la jerarquía de implementación, tan solo tendríamos que crear la clase MIXWindow, pero no tendría que variar para nada la jerarquía de abstracciones.

Otro ejemplo: si fuese necesario crear un nuevo tipo de ventana, se trataría de una modificación en la jerarquía de abstracciones, por lo que sólo tendríamos que crear una clase, por ejemplo, ApplicationWindow. Obviamente, no tocaríamos para nada la jerarquía de implementación, porque la adición de ApplicationWindow no habría implicado ninguna nueva plataforma. Con estos dos ejemplos se ve que las jerarquías pueden evolucionar de manera independiente.

Por otro lado, el patrón Puente se centra en identificar y desacoplar la abstracción de la implementación. El código cliente sólo trata con la abstracción, pues no quiere hacer frente a los detalles de la plataforma de la que depende. El patrón encapsula esta complejidad detrás de una abstracción "envoltorio".

Implementación

Diagrama de clases para el patrón Bridge:



Descripción de los participantes (los nombres de los paréntesis son las clases comentadas en el ejemplo Window/WindowImp anteriormente comentado):

Abstraction (Window): Clase abstracta que define las operaciones que conforman la abstracción (los métodos que el cliente percibe de la abstracción). Contiene una referencia al objeto Implementor que sirve para delegar las peticiones del código cliente en el Implementor.

RefinedAbstraction (IconWindow, TransientWindow): Subclase de Abstraction que puede ampliar su interfaz definiendo nuevas operaciones.

Implementor (WindowImp): Define la interfaz para la jerarquía de clases de implementación. Esta interfaz no tiene que corresponderse exactamente con la interfaz de la jerarquía de clases de la Abstraction. De hecho, las dos interfaces pueden ser bastante diferentes. Es habitual que la interfaz de Implementor se componga exclusivamente de operaciones primitivas (de bajo nivel) mientras que Abstraction defina operaciones de alto nivel basadas en estas primitivas.

ConcreteImplementor (XWindowImp, PMWindowImp): Clases concretas que implementan a Implementor.

Aplicabilidad y Ejemplos

El uso del patrón Bridge es adecuado en alguna de las siguientes situaciones:

- Cuando se quiere evitar una vinculación permanente entre una abstracción y su implementación. Este es el caso, por ejemplo, en que tiene que ser posible seleccionar o cambiar una implementación en tiempo de ejecución (Por ejemplo, una aplicación que permite, en tiempo de ejecución, conmutar en caso de fallo, desde un RDBMS a un sistema de ficheros para realizar las operaciones de acceso a datos).
- En aplicaciones en que es necesario que tanto las abstracciones como las implementaciones puedan extenderse mediante subclases. En tal situación, el patrón Bridge permite crear dos jerarquías de clases paralelas y hacerlas evolucionar de manera independiente.

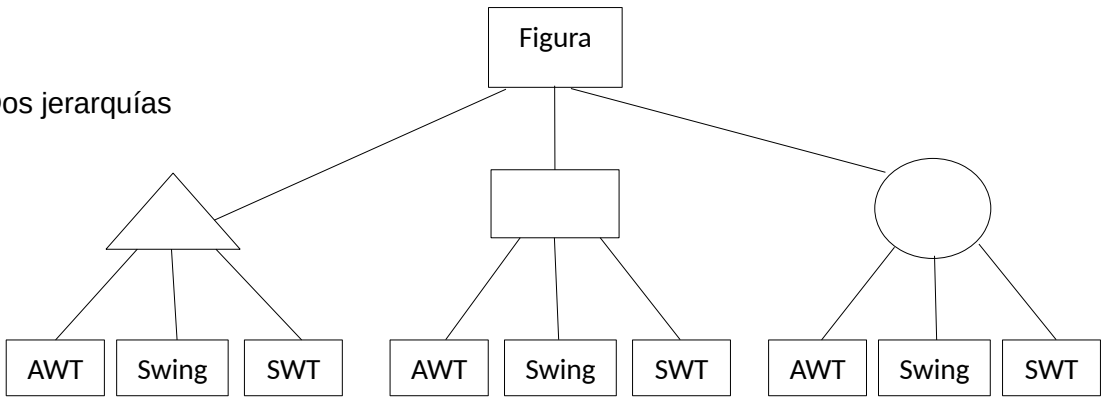
- En situaciones en las que las modificaciones efectuadas en la implementación de una abstracción no deban afectar a las clases cliente, esto es, el código cliente no tenga que ser recompilado.
- Cuando se quiere ocultar completamente al código cliente la implementación de una abstracción.
- Cuando se obtiene un diseño con una proliferación de clases considerable en una única jerarquía de herencia. Tal proliferación es una señal para dividirla en dos jerarquías.

Ejemplo 1: Aplicación que dibuja figuras geométricas con diferentes librerías gráficas

El siguiente ejemplo presenta, salvando las distancias, un caso muy similar al comentado en las páginas anteriores. Por un lado tenemos diferentes figuras geométricas y por otro tenemos diferentes librerías gráficas para representarlas. Además, queremos que las clases cliente se puedan programar sin tener que conocer el tipo concreto de librería gráfica instalada o configurada en el sistema.

Supongamos que la aplicación permite dibujar tres tipos de figuras (triángulos, rectángulos y circunferencias) y utilizar tres tipos de librerías gráficas (AWT, Swing y SWT). En este caso, usando el mecanismo de Herencia de forma convencional –sin aplicar el patrón Puente- necesitaríamos nueve clases, tal y como muestra la siguiente figura:

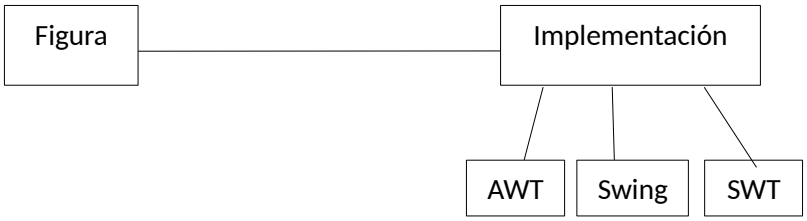
Dos jerarquías



El problema de esta aproximación no es el hecho de tener que crear nueve clases, sino la dificultad de mantener la aplicación ante nuevos requerimientos de figuras y/o de librerías gráficas.

Aplicando el patrón Puente reduciríamos en tres el número de clases totales, ya que tendríamos una jerarquía de tres figuras (las abstracciones o representaciones externas) y una jerarquía de tres implementaciones (las librerías).

La figura siguiente muestra este caso:



Claramente, este diseño es más conveniente que el anterior.

No obstante, por si queda alguna duda, la siguiente tabla muestra un breve escalado con el número de clases terminales requeridas usando Herencia y usando el patrón Puente:

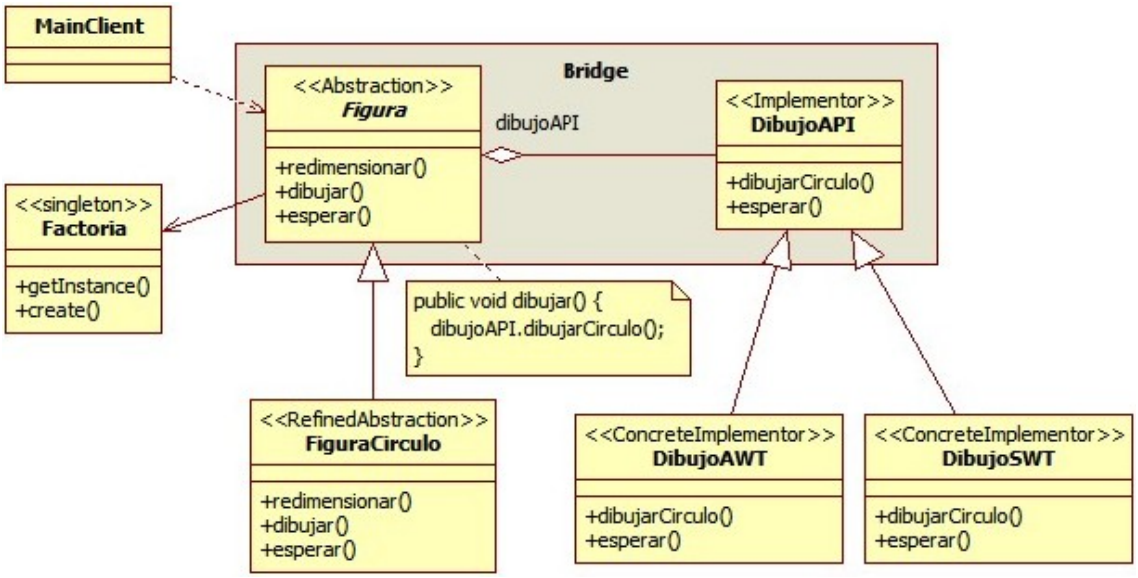
Abstracciones	***Implementaciones***	***Herencia***	***Bridge***	***%Beneficio Bridge***
2	2	4	4	0%
3	2	6	5	16%
3	3	9	6	33%
4	4	16	8	50%
5	4	20	9	55%

En la tabla observamos que para números bajos de abstracciones e implementaciones la diferencia entre usar el patrón o no usarlo aporta un beneficio pequeño o nulo. Lo sorprendente es ver cómo con tan solo un pequeño aumento en el número de abstracciones e implementaciones se produce un crecimiento exponencial en las

clases requeridas en la Herencia y, consiguientemente, un ahorro al utilizar el patrón Bridge.

Código de la aplicación del ejemplo

Para simplificar, se ha limitado el número de figuras a una: las circunferencias, y sólo se utilizan dos tipos de librerías gráficas (AWT y SWT). El siguiente diagrama de clases muestra las entidades que intervienen en la aplicación y sus relaciones.



Comenzamos viendo la jerarquía de las abstracciones: la representación externa que observa el código cliente del sistema.

Factoria.java

Ante la pregunta: ¿Cómo obtiene la clase Figura la instancia adecuada de la subclase de DibujoAPI? Respuesta: Figura podría utilizar el operador *new* para crear *hardcoded* una instancia de alguna de las subclases, aunque lo profesional es utilizar una factoría, que es lo que se ha hecho en esta ocasión.

Esta clase se limita a leer de un fichero de propiedades el nombre de la clase de implementación (DibujoAWT o DibujoSWT) que se utilizará en esta configuración específica (en este terminal) y a continuación crea una instancia y la devuelve a la clase Figura.

```
package estructurales.bridge.dibujoAPI;

import java.io.IOException;
import java.io.InputStream;
```

```
// Factoria implementada como Singleton
public class Factoria {

	private static Factoria factory = new Factoria();

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

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

	public static Factoria getInstance() {
		return factory;
	}

	public DibujoAPI create(String nombrePropiedad) {
		try {
			// La clase se obtiene leyendo del archivo properties
			Class<?> clase =
Class.forName(getClase(nombrePropiedad));
			// Crear una instancia
			return (DibujoAPI) clase.newInstance();
		} catch (ClassNotFoundException e) { // No existe la clase
			e.printStackTrace();
			return null;
		} catch (Exception e) { // No se puede instanciar la clase
			e.printStackTrace();
			return null;
		}
	}

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

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

			// Lectura de propiedad
			String nombreClase =
prop.getProperty(nombrePropiedad, "");
			if (nombreClase.length() > 0)
				return nombreClase;

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

```
	}
}
```

A continuación, el contenido del fichero al que accede la factoría (una única línea). El fichero debe existir en el mismo paquete que la clase factoría.

plataforma.properties

```
plataforma=estructurales.bridge.dibujoAPI.DibujoSWT
```

Notad que la opción alternativa es utilizar la librería AWT:

```
plataforma=estructurales.bridge.dibujoAPI.DibujoAWT
```

Figura.java

Proporciona la representación externa del sistema, es decir, su interfaz es lo único que perciben del sistema las clases cliente. Colabora con la Factoría para obtener la subclase de Implementor que se debe utilizar en este terminal. Una vez que ha obtenido el Implementor, este se hereda en las RefinedAbstraction, por lo que ya será posible que las abstracciones deleguen las responsabilidades en las implementaciones.

```
package estructurales.bridge.dibujoAPI;

/** "Abstraction" */
public abstract class Figura {

	/*
	 * Obtener la implementación mediante una factoría.
	 */
	protected static DibujoAPI dibujoAPI =
		Factoria.getInstance().create("plataforma");

	/*
	 * Metodos que pertenecen a la "Abstraction"
	 */

	// Multiplicar el radio de la circunferencia por
	// el portecentaje indicado por parametro
	public abstract void redimensionar(double pct);

	/*
	 * Metodos que se delegan en el "Implementor"
	 */

	// Dibujar un circulo
	public abstract void dibujar();
```

```
	// Codigo a ejecutar una vez realizado el dibujo,
	// normalmente mostrar un mensaje de finalizacion
	public abstract void esperar();

}
```

FiguraCirculo.java

Esta clase es una implementación de la clase Figura, aunque forma parte de la jerarquía de abstracciones del patrón Bridge. Las clases cliente crean objetos de esta clase para utilizar sus métodos, la mayoría de los cuales delegan en las operaciones correspondientes de la jerarquía de implementación del patrón. Por ejemplo, se muestra en color amarillo como la operación dibujar() se implementa mediante una llamada a la operación dibujarCirculo() del objeto dibujoAPI definido como protegido en la superclase Figura. En el caso de la operación esperar() sucede algo parecido, aunque las operaciones tienen el mismo nombre en ambas jerarquías. Sin embargo, la operación redimensionar es diferente, ya que su implementación forma parte exclusiva de la jerarquía de abstracción, es decir, no tiene contrapartida en la jerarquía de implementación.

```
package estructurales.bridge.dibujoAPI;

/** "Refined Abstraction" */
public class FiguraCirculo extends Figura {
	private int x, y, radio;

	public FiguraCirculo(int x, int y, int radio) {
		this.x = x;
		this.y = y;
		this.radio = radio;
	}

	/*
	 * Metodos que se delegan en el "Implementor".
	 * drawCircle es especifico de la implementacion.
	 */
	public void dibujar() {
		dibujoAPI.dibujarCirculo(x, y, radio);
	}
	public void esperar() {
		dibujoAPI.esperar();
	}

	/*
	 * Metodos que pertenece a la "Abstraction"
	 * resizeByPercentage es especifico de la abstraccion.
	 */
	public void redimensionar(double pct) {
		radio *= pct;
	}
```

```
}
```

Pasemos a ver la jerarquía de las implementaciones: el código que queda oculto y aislado al código cliente y que puede variar y evolucionar de manera independiente a la jerarquía de las abstracciones. Notad que la interfaz presentada en esta jerarquía no tiene porqué parecerse a la de la jerarquía de abstracciones, sencillamente debe existir una correspondencia (un mapeado) pero cualquier otra consideración es arbitraria.

DibujoAPI.java

```
package estructurales.bridge.dibujoAPI;

/** "Implementor" */
interface DibujoAPI {

	static final int FRAME_WIDTH = 450;
	static final int FRAME_HEIGHT = 450;

	public void dibujarCirculo(int x, int y, int radio);
	public void esperar();
}
```

DibujoAWT.java

Subclase que implementa la interfaz anterior haciendo uso de la librería AWT.

```
package estructurales.bridge.dibujoAPI;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;

import javax.swing.JFrame;

/** "ConcreteImplementor" 1/2 */
class DibujoAWT implements DibujoAPI {

	private int x, y, radio;

	/*
	 * Puntero estático al objeto actual: necesario para poder
acceder a
	 * las variables miembro desde el método de callback del JFrame,
que
	 * debe estatico, ya que se comparte por todas las
cicunferencias.
	 */
	private static DibujoAWT objEnCurso;

	private static JFrame marco;
```

```
	private static Font font;

	static {
		marco = new JFrame() {
			private static final long serialVersionUID = 1L;

			@Override
			public void paint(Graphics g) {
				int x = objEnCurso.x;
				int y = objEnCurso.y;
				int radio = objEnCurso.radio;

				g.setColor(Color.BLUE);
				g.fillOval(x, y, radio, radio);

				g.setFont(font);
				g.setColor(Color.BLACK);
				g.drawString("("+x+","+y+") r:" + radio, x-15,
y-15);
			}
		};
		marco.setSize(FRAME_WIDTH, FRAME_HEIGHT);
		marco.setTitle("API de AWT");
		marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		marco.setLocationRelativeTo(null);
		marco.setVisible(true);
		font = new Font("Arial",Font.PLAIN, 10);
	}

	@Override
	public void dibujarCirculo(int x, int y, int radio) {
		objEnCurso = this;
		this.x = x;
		this.y = y;
		this.radio = radio;
		// Actualizamos sin limpiar el fondo, asi evitamos
		// que se borren los circulos anteriores
		marco.update(marco.getGraphics());
		System.out.printf("DibujoPlataformaAWT. Circulo en %d:%d
con radio %d\n",
				x, y, radio);
	}

	@Override
	public void esperar() {
		System.out.println("Dibujo finalizado.");
	}

}
```

DibujoSWT.java

Subclase que implementa la interfaz anterior haciendo uso de la librería SWT.

Nota: Es necesaria que la librería ‘swt.jar’ esté en el build path.

```java
package estructurales.bridge.dibujoAPI;

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

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.PaintEvent;
import org.eclipse.swt.events.PaintListener;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.graphics.GC;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

/** "ConcreteImplementor" 2/2 */
class DibujoSWT implements DibujoAPI {

	/*
	 * Contenedores para almacenar todas las
	 * figuras que envíe el código cliente
	 */
	private List<Integer> x, y, radio;

	/*
	 * Objetos de SWT
	 */
	private Display display;
	private Shell shell;
	private Font font;

	/*
	 * Constructor
	 */
	public DibujoSWT() {
		inicializar();
		crearGUI();
	}

	/*
	 * Este es el metodo que se ejecuta cuando el usario
	 * invoca al metodo dibujar() de la RefinedAbstraction.
	 * Se almacenan los parametros en los array correspondientes
	 * para cuando llegue el momento de pintar las figuras.
	 */
	@Override
	public void dibujarCirculo(int x, int y, int radio) {
		this.x.add(x);
		this.y.add(y);
		this.radio.add(radio);

		System.out.printf(
				"DibujoPlataformaSWT. Circulo en %d:%d con
radio %d\n", x, y,
				radio);
	}

	/*
	 * El usuario llama a esperar() para indicar que ya
	 * ha terminado de dibujar. esperar() invoca a
	 * shell.redraw(), lo que produce que el listener de
```

```
 * pintado llame a dibujarEnPantalla().
 */
@Override
public void esperar() {
	shell.redraw();
	// El codigo siguiente mantiene la GUI en standby,
	// pendiente de cualquier evento que se produzca.
	System.out.println("Dibujo finalizado.");
	while (!shell.isDisposed()) {
		if (!display.readAndDispatch())
			display.sleep();
	}
	display.dispose();
}

/*	METODOS PRIVADOS */

/*
 * Crear los objetos
 */
private void inicializar() {
	display = new Display();
	shell = new Shell(display);
	font = new Font(display,"Arial",8,SWT.NORMAL);

	x = new ArrayList<Integer>();
	y = new ArrayList<Integer>();
	radio = new ArrayList<Integer>();
}

/*
 * Crear la GUI y declarar el listener de pintado
 */
private void crearGUI() {
	shell.setText("API de SWT");
	shell.setLocation(20, 20);
	shell.setSize(FRAME_WIDTH, FRAME_HEIGHT);
	shell.open();
	// Listener para proceder cuando se pinte en la pantalla
	shell.addPaintListener(new PaintListener() {
		public void paintControl(PaintEvent event) {
			GC gc = event.gc;
			dibujarEnPantalla(gc);
			gc.dispose();
		}
	});
}

/*
 * Este metodo recorre todas las figuras enviadas por
 * el usuario y las pinta (muestra).
 */
private void dibujarEnPantalla(GC gc) {
	gc.setAntialias(SWT.ON);
  gc.setFont(font);
  // Para saber el número de figuras se ha tomado 'x'
  // pero hubiera funcionado también con 'y' y con 'radio'.
  for (int i=0; i<x.size(); i++) {
	gc.setForeground(display.getSystemColor(SWT.COLOR_BLACK));
```

```
gc.setBackground(display.getSystemColor(SWT.COLOR_WIDGET_BACKGROUND));
              gc.drawText("("+x.get(i)+","+y.get(i)+") r:"
                        + radio.get(i), x.get(i)-15, y.get(i)-15);


      gc.setBackground(display.getSystemColor(SWT.COLOR_BLUE));
              gc.fillOval(x.get(i), y.get(i), radio.get(i),
radio.get(i));
        }
      }
}
```

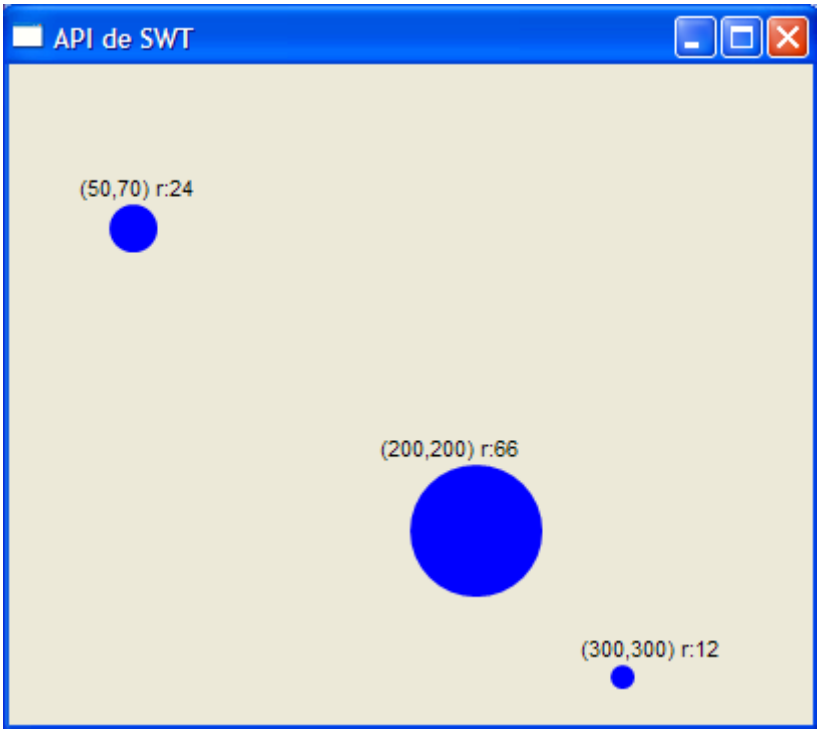
Por último, veamos el código cliente.

MainClient.java

Notad la sencillez del código cliente, que desconoce la plataforma gráfica subyacente.

```
package estructurales.bridge.dibujoAPI.client;

import estructurales.bridge.dibujoAPI.Figura;
import estructurales.bridge.dibujoAPI.FiguraCirculo;

/** "Client" */
public class MainClient {

      private static final long serialVersionUID = 1L;

      public static void main(String[] args) {
            // Creamos tres circulos
            Figura c1 = new FiguraCirculo(200, 200, 22);
            Figura c2 = new FiguraCirculo(300, 300, 12);
            Figura c3 = new FiguraCirculo(50, 70, 24);

            // Aumentamos el radio del segundo circulo
            c1.redimensionar(3);

            // Los agregamos a un array de Figura
            Figura[] figuras = new Figura[] { c1, c2, c3 };

            // Los pintamos en la pantalla
            for (Figura figura : figuras) {
                  figura.dibujar();
            }

            // Esperamos a que el usuario cierre la ventana.
            // Podemos invocar al metodo esperar() sobre cualquier
circulo;
            // aqui lo hacemos sobre el primero
            c1.esperar();
      }

}
```

Salida para el caso de SWT (para AWT es prácticamente la misma salida):



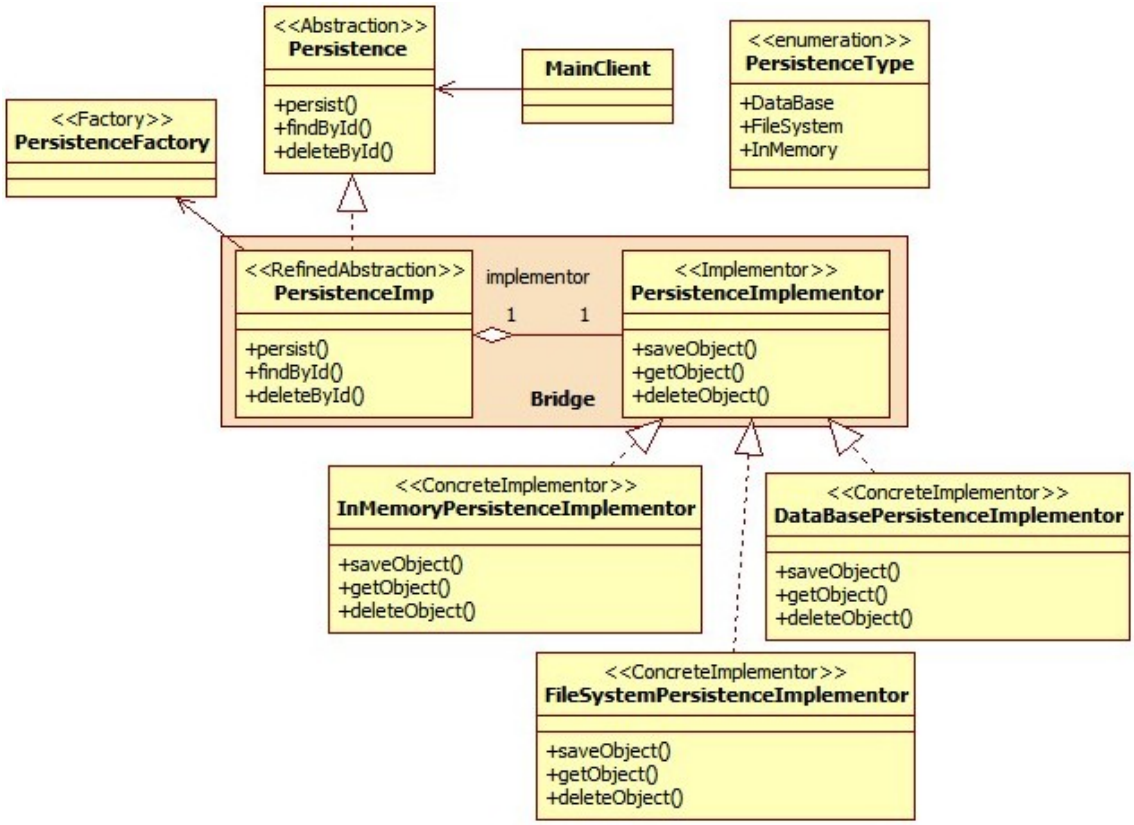
Ejemplo 2 – Diferentes mecanismos de persistencia

En este ejemplo se necesita un objeto que maneje la persistencia de una entidad mediante diferentes mecanismos alternativos:

- Estructura datos en memoria (HashMap): Por su velocidad, es un sistema de persistencia muy adecuado para la fase de desarrollo de una aplicación.
- Ficheros en disco: Es una opción a considerar tanto para la fase de desarrollo como para la entrada en producción de la aplicación, siempre que el número de registros que se tengan que gestionar sea relativamente pequeño. Por otro lado, hay que tener en cuenta que la entidad a persistir deberá poder serializarse.
- Base de datos: Es la opción recomendable cuando la aplicación debe entrar en la fase de producción. No obstante, se puede complementar con el mecanismo de ficheros en disco, de manera que los ficheros en disco actúan de caché local para evitar el acceso a la base de datos en operaciones muy comunes.

Mediante el uso de Generics y el API Reflection podríamos hacer que la jerarquía de implementación operase con cualquier entidad. Esto sería realmente útil. No obstante, para simplificar el ejemplo, la aplicación sólo trabajará con la entidad Alumno, que es una clase que consta de dos campos: el identificador y el nombre del alumno.

De nuevo, como sucedía con el primer ejemplo, este es un caso donde el patrón Puente encaja muy bien, proporcionando un diseño flexible, dinámico que facilitará el mantenimiento de la aplicación. El siguiente diagrama muestra las distintas entidades y las colaboraciones existentes en la aplicación:



Código fuente del ejemplo

Comenzamos viendo la jerarquía de las abstracciones.

PersistenceType.java

Esta enum declara los tres tipos de persistencia que admite la aplicación.

```
package estructurales.bridge.persistencia;

public enum PersistenceType {
      InMemory, FileSystem, DataBase;
}
```

PersistenceFactory.java

Observad que la factoría tiene visibilidad de paquete, pues sólo debe ser utilizada por la RefinedAbstraction.

```
package estructurales.bridge.persistencia;

/*
 * Factoría implementada como Singleton.
 * Crea un objeto de alguna de las subclases de "Implementor"
 */
class PersistenceFactory {

	static final PersistenceFactory instance =
		new PersistenceFactory();

	static PersistenceFactory getInstance() {
		return instance;
	}

	private PersistenceFactory() {

	}

	/*
	 * Crea un objeto de alguna de las subclases de "Implementor".
	 * Notad que no se cachea el tipo de implementor, ya que
	 * interesa poder invocar a create() en cualquier otro momento
	 * en tiempo de ejecucion para que cree otro tipo de implementor
	 */
	PersistenceImplementor create(PersistenceType tipoPersistencia)
{
		if (tipoPersistencia.equals(PersistenceType.InMemory)) {
			return new InMemoryPersistenceImplementor();
		} else if
(tipoPersistencia.equals(PersistenceType.FileSystem)) {
			return new FileSystemPersistenceImplementor();
		} else if
(tipoPersistencia.equals(PersistenceType.DataBase)) {
			return new DataBasePersistenceImplementor();
		} else {
			throw new IllegalArgumentException("El tipo de
persistencia indicado es erroneo: "+tipoPersistencia);
		}
	}

}
```

Persistence.java

Interfaz que declara las operaciones a las que puede acceder el código cliente.

```
package estructurales.bridge.persistencia;

/** "Abstraction" */
public interface Persistence {
```

```
	public String persist(Object object);

	public Object findById(String objectId);

	public void deleteById(String id);

}
```

PersistenceImp.java

Es la subclase que implementa la interfaz Persistence. Notad que tiene dos constructores para que el código cliente pueda elegir la subclase de Implementor con la que trabajar. Observad que delega todas las operaciones en Implementor.

```
package estructurales.bridge.persistencia;

/** "RefinedAbstraction" */
public class PersistenceImp implements Persistence {

	PersistenceImplementor implementor;

	/*
	 * Por defecto utilizamos la implementacion HahMap
	 */
	public PersistenceImp() {
		implementor =


PersistenceFactory.getInstance().create(PersistenceType.InMemory);
	}

	/*
	 * Este constructor permite elegir una de las
	 * implementaciones de persistencia disponibles
	 */
	public PersistenceImp(PersistenceType tipoPersitencia) {
		implementor =

	PersistenceFactory.getInstance().create(tipoPersitencia);
	}

	@Override
	public String persist(Object object) {
		return Long.toString(implementor.saveObject(object));
	}

	@Override
	public Object findById(String objectId) {
		return implementor.getObject(Long.parseLong(objectId));
	}

	@Override
	public void deleteById(String id) {
		implementor.deleteObject(Long.parseLong(id));
	}
```

```
}
```

Veamos ahora la jerarquía de las implementaciones.

PersistenceImplementor.java

```
package estructurales.bridge.persistencia;

/** "Implementor" **/
interface PersistenceImplementor {

	public long saveObject(Object object);

	public Object getObject(long objectId);

	public void deleteObject(long objectId);

}
```

Y ahora las subclases:

InMemoryPersistenceImplementor.java

```
package estructurales.bridge.persistencia;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

import estructurales.bridge.persistencia.client.Alumno;

/** "ConcreteImplementor" **/
public class InMemoryPersistenceImplementor
		implements PersistenceImplementor {

	// Estructura para almacenar los alumnos
	private static Map<Integer, Alumno> alumnosBD;

	/*
	 * Inicializador estatico para crear las estructuras
	 * de datos y algunos registros de ejemplo
	 */
	static {

		// Incializamos la base de datos en memoria
		alumnosBD = new HashMap<Integer, Alumno>();

		// Creamos un Alumno
		Alumno alumno=new Alumno(1000, "Miguel");
```

```
            // Creamos la primera entrada en la base de datos en
memoria
            alumnosBD.put(alumno.getId(), alumno);

            // Creamos otro Alumno
            alumno=new Alumno(1001, "Raquel");
            alumnosBD.put(alumno.getId(), alumno);
      }

      /*
       * Grabar un Alumno, independientemente de si
       * existe o no
       */
      @Override
      public long saveObject(Object object) {
            Alumno alumno = (Alumno) object;
            // Comprobar si existe, en cuyo caso lo borramos
            deleteObject(alumno.getId());
            // Siempre se crea de nuevo
            crearAlumno(alumno);

            // debug
            System.out.println("Listando alumnos...");
            for (Alumno a : listarAlumnos()) {
                  System.out.println(a);
            }

            return alumno.getId();
      }

      /*
       * Recuperar el alumno cuyo ID coincide con el parametro
       */
      @Override
      public Object getObject(long objectId) {
            int intObjectId;
            int IDtope = Integer.MAX_VALUE;
            if (objectId>IDtope) {
                  throw new IllegalArgumentException("El ID no puede
exceder de "+IDtope);
            } else {
                  intObjectId = (int)objectId;
            }
            if (alumnoExiste(intObjectId)) {
                  return alumnosBD.get(intObjectId);

            }
            return null;
      }

      /*
       * Borrar el Alumno cuyo ID coincide con el parametro
       */
      @Override
      public void deleteObject(long objectId) {
            int intObjectId;
            int IDtope = Integer.MAX_VALUE;
            if (objectId>IDtope) {
```

```
                  throw new IllegalArgumentException("El ID no puede
exceder de "+IDtope);
            } else {
                  intObjectId = (int)objectId;
            }
            // Si existe borramos el alumno
            if (alumnoExiste(intObjectId)) {
                  System.out.println("borrado alumno - id = " +
intObjectId);
                  alumnosBD.remove(intObjectId);
            }
      }

      /*
       * Metodo privado que añade una nueva entrada en
       * la base de datos
       */
      private void crearAlumno(Alumno alumno) {
            System.out.println("persistiendo Alumno - nombre = " +
alumno.getNombre());
            alumnosBD.put(alumno.getId(), alumno);
      }

      /*
       * Comprobar si el nombre de un Alumno se
       * encuentra en el Map
       */
      private boolean alumnoExiste(int objectId) {
            return alumnosBD.containsKey(objectId);
      }

      /*
       * Devolver la lista de alumnos
       */
      private List<Alumno> listarAlumnos() {
            List<Alumno> alumnos=new ArrayList<Alumno>();
            for (Entry<Integer, Alumno> e: alumnosBD.entrySet()) {
                  Alumno Alumno=e.getValue();
                  alumnos.add(Alumno);
            }
            Collections.sort(alumnos);
            return alumnos;
      }

}
```

FileSystemPersistenceImplementor.java

```
package estructurales.bridge.persistencia;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
```

```
/** "ConcreteImplementor" */
class FileSystemPersistenceImplementor
            implements PersistenceImplementor {

      @Override
      public long saveObject(Object object) {
            long fileId = System.currentTimeMillis();

            // Crear un fichero cuyo nombre sera la marca de hora
actual
            // EL DIRECTORIO persistence DEBE EXISTIR!!
            File f = new
File("c://persistence/"+Long.toString(fileId));

            // write file to Stream
            writeObjectToFile(f, object);

            return fileId;
      }

      @Override
      public Object getObject(long objectId) {
            File f = new
File("c://persistence/"+Long.toString(objectId));
            return readObjectFromFile(f);
      }

      @Override
      public void deleteObject(long objectId) {
            File f = new
File("c://persistence/"+Long.toString(objectId));
            f.delete();
      }

      /* METODOS PRIVADOS */

      /*
       * Abrir el fichero recibido por parametro,
       * leer el objeto en él almacenado y cerrar
       * el fichero. Finalmente devolver el objeto.
       */
      private Object readObjectFromFile(File f) {
            try {
                  ObjectInputStream in =
                      new ObjectInputStream(
                          new FileInputStream(f));

            Object obj = in.readObject();

            in.close();
            return obj;
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            } catch (ClassNotFoundException e) {
                  e.printStackTrace();
            }
```

```
            return null;
      }

      /*
       * Serializar el objeto y guardarlo en un fichero
       */
      private void writeObjectToFile(File f, Object object) {
            try {
                  ObjectOutputStream out =
                      new ObjectOutputStream(
                          new FileOutputStream(f));
                  out.writeObject(object);
            out.close();
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }

}
```

DataBasePersistenceImplementor.java

```
package estructurales.bridge.persistencia;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import estructurales.bridge.persistencia.client.Alumno;

/** "ConcreteImplementor" **/

/**
 * Si no tenemos en MySQL la BD hacemos lo siguiente:
 * create database dbalumnos;
 * use alumnos;
 * create table alumnos (id int primary key, nombre varchar(20));
 */
class DataBasePersistenceImplementor
            implements PersistenceImplementor {

      // Cadena de conexión
      private final String URLConexion =
            "jdbc:mysql://localhost:3306/dbalumnos?
user=root&password=root";

      public DataBasePersistenceImplementor() {

      }

      /*
       * Grabar un Alumno.
       */
```

```
	@Override
	public long saveObject(Object object) {
		Alumno alumno = (Alumno) object;
		// Si no existe un registro en la BD con ese ID -> INSERT
		if (getObject(alumno.getId()) == null) {
			Connection con = null;
			try {
				con = getConnection();
				PreparedStatement ps =
con.prepareStatement("insert into alumnos"
					+ " (id, nombre) "
					+ " values (?,?) ");
				ps.setInt(1, alumno.getId());
				ps.setString(2, alumno.getNombre());

				if (ps.executeUpdate() == 0)
					throw new RuntimeException("No se ha
podido grabar el alumno");
				ps.close();
			} catch (SQLException e) {
				e.printStackTrace();
			} finally {
				releaseConnection(con);
			}
		// Existe un registro en la BD con ese ID -> UPDATE
		} else {
			actualizarAlumno(alumno);
		}

		return alumno.getId();
	}


	/*
	 * Recuperar el alumno cuyo ID coincide con el parametro
	 */
	@Override
	public Object getObject(long objectId) {
		Connection con = null;
		try {
			con = getConnection();
			PreparedStatement ps = con.prepareStatement("select
id, nombre from alumnos where id=?");
			ps.setInt(1, (int)objectId);
			ResultSet rs = ps.executeQuery();
			Alumno alumno = null;
			if (rs.next()) {

				alumno=new Alumno(rs.getInt(1),
rs.getString(2));
			}
			rs.close();
			ps.close();
			return alumno;
		} catch (SQLException e) {
			for (Throwable t : e) {
				System.err.println("Errores: " + t);
			}
		} finally {
```

```
                releaseConnection(con);
            }
            return null;
        }

        /*
         * Borrar el Alumno cuyo ID coincide con el parametro
         */
        @Override
        public void deleteObject(long objectId) {
            Connection con = null;
            try {
                con = getConnection();
                PreparedStatement ps = con.prepareStatement("delete
from alumnos"
                        + " where id=?) ");
                ps.setInt(1, (int)objectId);

                if (ps.executeUpdate() == 0)
                    throw new RuntimeException("No se ha podido
eliminar el alumno");
                ps.close();
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                releaseConnection(con);
            }
        }

        // METODOS PRIVADOS **********************

        /*
         * Hacer un update en lugar de un insert
         * cuando el alumno ya existe
         */
        private void actualizarAlumno(Alumno alumno) {
            Connection con = null;
            try {
                con = getConnection();
                PreparedStatement ps = con.prepareStatement("update
alumnos"
                        + " set nombre=? where id=?");

                ps.setString(1, alumno.getNombre());
                ps.setInt(2, alumno.getId());

                if (ps.executeUpdate() == 0)
                    throw new RuntimeException("No se ha podido
actualizar el alumno");
                ps.close();
            } catch (SQLException e) {
                e.printStackTrace();
            } finally {
                releaseConnection(con);
            }
        }

        /*
         * OPERACIONES GENERALES
```

```
	 */

	// Obtener la conexión
	private Connection getConnection() throws SQLException {
		return DriverManager.getConnection(URLConexion);
	}

	// Liberar la conexión
	private void releaseConnection(Connection con) {
		if (con != null) {
			try {
				con.close();
				con = null;
			} catch (SQLException e) {
				for (Throwable t : e) {
					System.err.println("Error: " + t);
				}
			}
		}
	}

}
```

Alumno.java

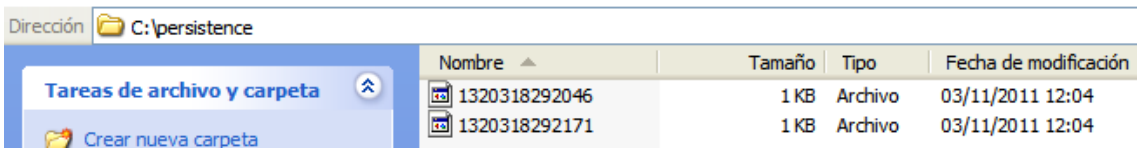
```
package estructurales.bridge.persistencia.client;

import java.io.Serializable;

public class Alumno implements Serializable, Comparable<Alumno> {

	private static final long serialVersionUID = 1L;

	private int id;
	private String nombre;

	public Alumno(int id, String nombre) {
		super();
		this.id = id;
		this.nombre = nombre;
	}

	public int getId() { return id;	}
	public String getNombre() {  return nombre;	}

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + id;
		result = prime * result + ((nombre == null) ? 0 :
nombre.hashCode());
		return result;
```

```
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Alumno other = (Alumno) obj;
		if (id != other.id)
			return false;
		if (nombre == null) {
			if (other.nombre != null)
				return false;
		} else if (!nombre.equals(other.nombre))
			return false;
		return true;
	}

	@Override
	public int compareTo(Alumno otroAlumno) {
		int otroId = otroAlumno.getId();
		String otroNombre = otroAlumno.getNombre();

		if (id == otroId)
			return 0; // iguales

		if (!nombre.equalsIgnoreCase(otroNombre))
			return nombre.compareTo(otroNombre);
		else
			return 0; // iguales
	}

	@Override
	public String toString() {
		return "Alumno [id=" + id + ", nombre=" + nombre + "]";
	}
}
```

Salida:

```
*************************************************
Probando la persistencia InMemory
*************************************************
Alumno creado: ID = 100 Nombre = Juan
persistiendo Alumno - nombre = Juan
Listando alumnos...
Alumno [id=100, nombre=Juan]
Alumno [id=1000, nombre=Miguel]
Alumno [id=1001, nombre=Raquel]
Alumno grabado.
Alumno recuperado: ID = 100 Nombre = Juan
Alumno modificado: ID = 100 Nombre = Eduardo
borrado alumno - id = 100
persistiendo Alumno - nombre = Eduardo
Listando alumnos...
Alumno [id=100, nombre=Eduardo]
Alumno [id=1000, nombre=Miguel]
Alumno [id=1001, nombre=Raquel]
Alumno grabado de nuevo.
*************************************************
Probando la persistencia FileSystem
*************************************************
Alumno creado: ID = 100 Nombre = Juan
Alumno grabado.
Alumno recuperado: ID = 100 Nombre = Juan
Alumno modificado: ID = 100 Nombre = Eduardo
Alumno grabado de nuevo.
*************************************************
Probando la persistencia DataBase
*************************************************
Alumno creado: ID = 100 Nombre = Juan
Alumno grabado.
Alumno recuperado: ID = 100 Nombre = Juan
Alumno modificado: ID = 100 Nombre = Eduardo
Alumno grabado de nuevo.
```

En el directorio ‘persistence’ (en mi caso c:\persistence), si no hemos variado el código de MainClient, tendrán que aparecer dos ficheros, uno correspondiente a la creación de Juan y otro como resultado de modificar este registro cambiando el nombre por Eduardo:



Problemas específicos e implementación

Sólo una clase Implementor

En situaciones en las que sólo tenemos una subclase de Implementor, no es obligatoria la existencia de una clase abstracta o una interfaz. Este es un caso particular del patrón Bridge, donde se tiene una relación uno a uno entre la Abstraction y el Implementor. No obstante, el patrón sigue siendo de utilidad, ya que permite que el Implementor cambie sin que se vean afectadas las clases cliente.

Crear el objeto Implementor correcto

¿Cómo, cuándo y dónde se decide qué subclase de Implementador instanciar cuando hay más de una?

Si la Abstraction sabe de la existencia de las ConcreteImplementor, entonces puede instanciar alguna de ellas en su constructor, dependiendo de algún parámetro recibido. Si, por ejemplo, una clase tipo Collection admite múltiples implementaciones, la decisión sobre qué implementación crear se puede basar en el tamaño de la colección. Así, se podría instanciar una implementación LinkedList cuando la colección es pequeña y una HashTable cuando es grande.

Otro método consiste en elegir una implementación por defecto en el momento que se inicia la aplicación y cambiarla después conforme al uso de la misma. Por ejemplo, si la clase de tipo Collection crece y sobre pasa un determinado umbral, entonces la aplicación cambia dinámicamente a un ConcreteImplementor más adecuado para el tratamiento de muchos elementos.

También es posible delegar completamente en otro objeto la decisión sobre qué subclase de Implementor instanciar. En los ejemplos mostrados anteriormente se ha utilizado una factoría, cuyo único propósito era encapsular el proceso de creación. En el ejemplo de las librerías gráficas, la factoría obtenía el nombre de la subclase de Implementor desde un fichero de propiedades, mientras que en el ejemplo de los diferentes mecanismos de persistencia la factoría obtenía el nombre de la subclase a instanciar mediante un parámetro enviado por el código cliente. La ventaja de utilizar una factoría es que se evita que la Abstraction se acople directamente a las subclases de Implementor.

Patrones relacionados

Una Abstract Factory puede crear y configurar un Bridge.

El patrón Adapter se emplea cuando se necesita que clases no relacionadas puedan colaborar juntas. Habitualmente se aplica en sistemas existentes. En cambio, Bridge se aplica en tiempo de diseño, cuando no existe aún la aplicación, para permitir que abstracciones e implementaciones puedan variar de manera independiente.