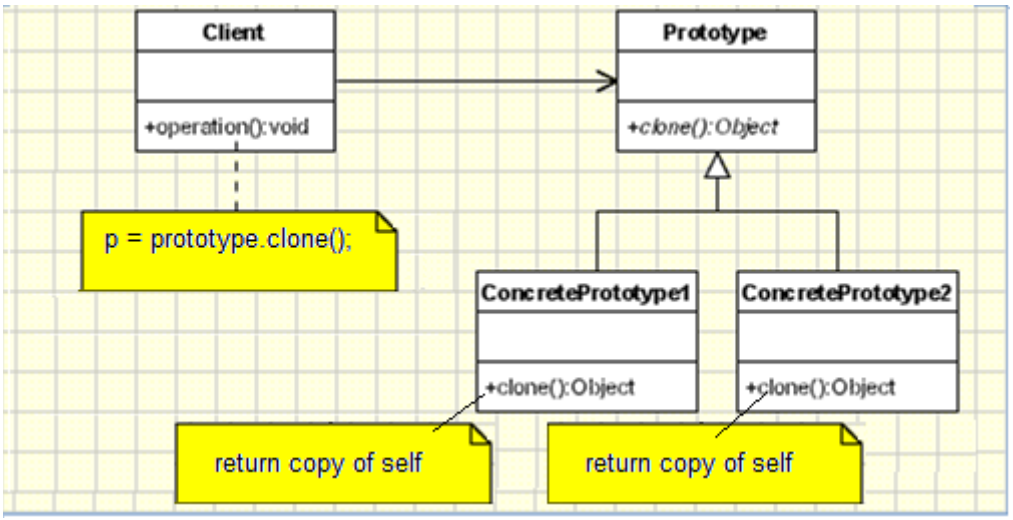
# Prototype

## Diagrama de clases e interfaces




## Intención

Este patrón tiene como finalidad obtener un nuevo objeto mediante la clonación de otro objeto ya existente, conocido como prototipo. El prototipo sirve de molde para crear rápidamente nuevos objetos con las mismas características que el molde.

Una clase cliente necesita un nuevo objeto y puede desconocer tanto la clase del objeto como los detalles de su creación, por lo que el objeto se obtiene a través de la operación de una interfaz que permite obtener una copia. Las clases concretas de prototipos desarrollan esta operación para crear copias específicas.

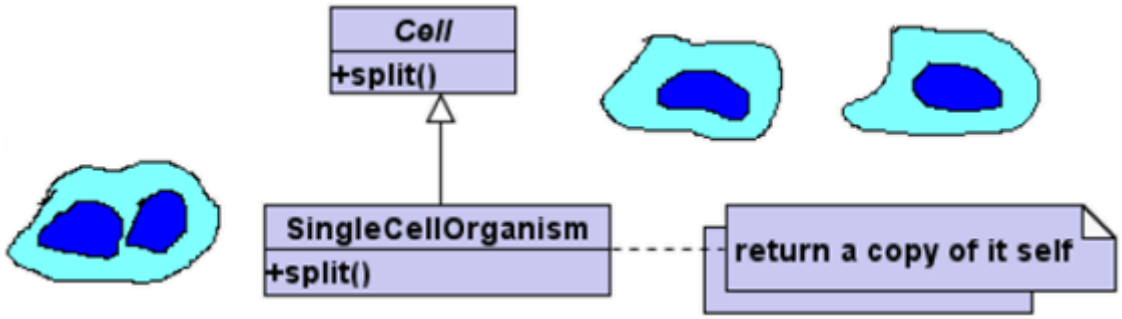
## Motivación

Mediante el patrón Protoype cuando queremos obtener un nuevo objeto (producto) en lugar de crearlo de cero lo que hacemos es recuperarlo a partir de un conjunto de objetos similares prefabricados. De este conjunto de objetos seleccionaremos el que mejor se adecue a nuestras necesidades. Es frecuente que el objeto recuperado se tenga que configurar sensiblemente para adaptarlo a los requerimientos específicos del momento y del contexto. Protoype resulta particularmente interesante cuando es costoso computacionalmente crear desde cero instancias de esa clase.

Ejemplo de la vida real

El siguiente ejemplo nos ayudará a entender el concepto de *prototipo* en un caso de la vida real. Refleja el proceso de la división celular (mitosis) y cómo el concepto de clonación se adapta a la programación.

Cuando un ser humano se ve afectado por un virus, por ejemplo la gripe, las células T comienzan a combatirlo y simultáneamente otras células tratan de reconocer al virus. En caso de ser reconocido, la glándula tiroides comienza a producir células B, que son glóbulos blancos especializados que contendrán la información adecuada (anticuerpos) sobre cómo atacar y eliminar este virus del cuerpo humano. La acción de replicar las células se conoce como clonación.



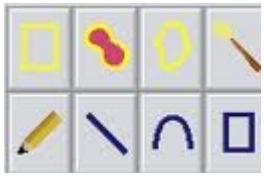

Al mismo tiempo que las células B combaten al virus, el cuerpo humano sigue generando nuevas replicas de estas células. El proceso de crear de cero nuevas células B es muy complejo y lento, por lo que nuestro cuerpo lleva a cabo una estrategia mucho más efectiva: hacer que las células B se dupliquen ellas mismas en tiempo de "ejecución".

Ejemplos en programación

Encontramos ejemplos clásicos de este patrón en:

- Los editores de texto, donde las características de copiar y pegar mejoran la productividad del usuario. El objeto copiado contiene el mismo valor que el original y le servirá al usuario como punto de partida para, seguramente, alterarlo y que termine siendo un objeto diferente al original.

- Los editores gráficos, donde podemos crear mediante una barra de herramientas figuras (arco, rectángulo, línea, etc).

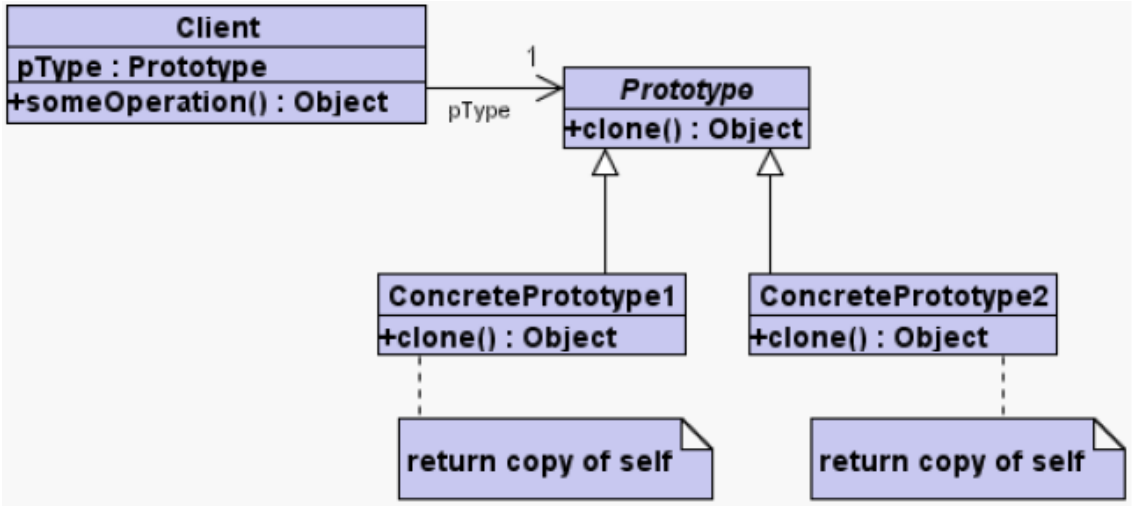


Cada vez que seleccionamos una figura de la barra de herramientas sabemos que tendrá una forma precisa (círculo, rectángulo, etc), por lo que claramente se trata de copias de prototipos. Estos objetos gráficos pertenecerán a una jerarquía cuyas clases derivadas implementarán el mecanismo de clonación.

Más adelante desarrollaremos el caso del editor gráfico.

## Implementación

El siguiente diagrama muestra los participantes en el patrón:



Descripción de los participantes:

**Prototype:** Esta interfaz o clase abstracta declara un método clone() que devuelve un objeto clonado. Esta interfaz o clase abstracta oculta los ConcretePrototype al Client.

**ConcretePrototype:** Estas clases implementan la interfaz Prototype o extienden la clase abstracta Prototype mediante la definición del método clone(). Cuando se llama a su método clone() devuelve una copia de sí misma.

**Client:** Inicializa la acción de clonado solicitando a Prototype que le proporcione un objeto ConcretePrototype, aunque Client desconoce la clase real de ConcretePrototype. El objeto se obtiene mediante el método clone() definido en cada una de las ConcretePrototype.

El proceso de clonado necesita que las diferentes instancias de las clases producto, esto es, los objetos de ConcretePrototype, estén disponibles de alguna manera a las clases cliente. Cuando sea necesario, el cliente solicitará un nuevo objeto a la clase (o interfaz) Prototype, por ejemplo, mediante un token como 'elipse' o 'rectangulo'. La clase Prototype utilizará la clase ConcretePrototype adecuada para servir el tipo de objeto solicitado. ConcretePrototype utilizará su método clone() para crear una nueva instancia de sí misma.

## Aplicabilidad y Ejemplos

Veamos algunos ejemplos en los que sería apropiado el uso de este patrón.

Ejemplo - Editor gráfico

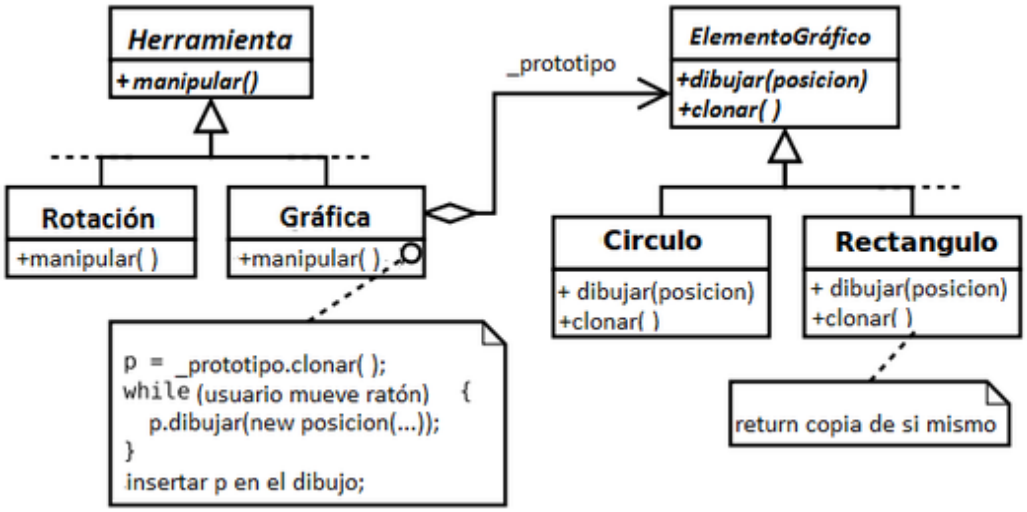
Continuando con el caso de los editores gráficos introducido anteriormente, supongamos que creamos un framework orientado a gráficos. El framework dispone de una serie de servicios:

- Gestionar documentos (nuevo, abrir, guardar)
- Herramientas que realizan operaciones sobre los elementos gráficos (rotar, desplazar, etc)

Un usuario puede utilizar nuestro framework y crear cualquier tipo de subclase de la interfaz ElementoGrafico que se ajuste a su lógica de negocio. Por ejemplo, podría crear subclases del tipo Tornillo, Tuerca, etc. en un contexto de mecánica. En cambio, en un contexto de arquitectura podría crear ladrillos, tabiques, etc. Por lo tanto, nuestro framework desconoce las clases de los usuarios, pues se trata de clases que corresponden a una aplicación particular de cada usuario.

Analicemos la siguiente figura:

La jerarquía de Herramienta forma parte del framework, mientras que la jerarquía de ElementoGrafico, exceptuando la propia interfaz ElementoGrafico, pertenece a la aplicación del usuario:




En la figura vemos una clase llamada Grafica. Esta clase es la que permite crear figuras concretas al usuario, como elipses, rectángulos, etc. Por tanto, se ejecuta cuando el usuario pulsa el botón 'Elipse', 'Figura', etc. Ahora bien, esta clase suscita dos temas importantes:

- Cada vez que el usuario pulsa sobre una herramienta se debe obtener la misma figura. Normalmente esto lo solventamos creando un nuevo objeto. No obstante, en lugar de crear cada vez un objeto partiendo de cero sería más eficiente tener ya preparada (cacheada) una figura y cuando el usuario quiera una nueva, obtener un duplicado de la cacheada.

- Grafica dispone de un método manipular(), que supongamos permite seleccionar una figura y moverla a otra posición (drag and drop). El proceso de

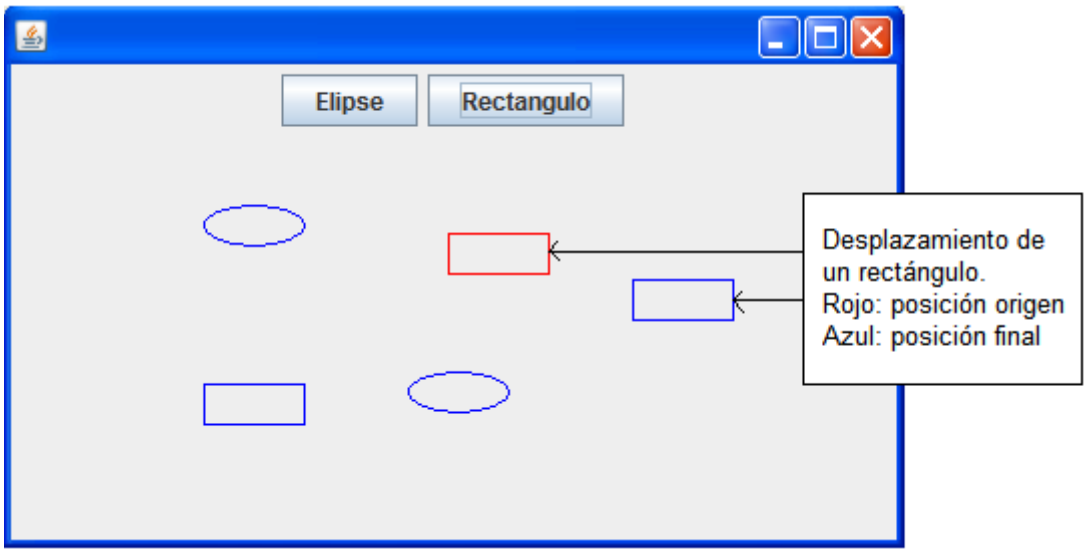
arrastrar una figura (desplazando el ratón con el botón izquierdo pulsado) y soltarla en otra posición diferente a la original, requiere crear una nueva figura, ya que el usuario ha de observar durante el desplazamiento la figura original en la posición original y la nueva figura (que sustituirá a la original) desplazándose conforme los movimientos del ratón. Una vez que el usuario suelte la nueva figura en la posición deseada, habrá que eliminar la figura original, pues en caso contrario tendríamos dos figuras. Ahora bien, tanto la clase Grafica como la clase Rotacion ¿cómo pueden crear instancias de las subclases de ElementoGrafico si desconocen el tipo exacto de tales clases?

La solución a las dos cuestiones planteadas las podemos resolver aplicando el patrón Protoype. Prototype obtiene duplicados de otros objetos y también permite obtener nuevos objetos sin conocer su clase ni ningún detalle sobre la manera de crearlos. En este sentido se parece mucho a la familia de patrones Factory, aunque los factory no disponen de un conjunto de objetos prefabricados, sino que se limitan a crear una instancia y devolverla.

## Código para el editor gráfico

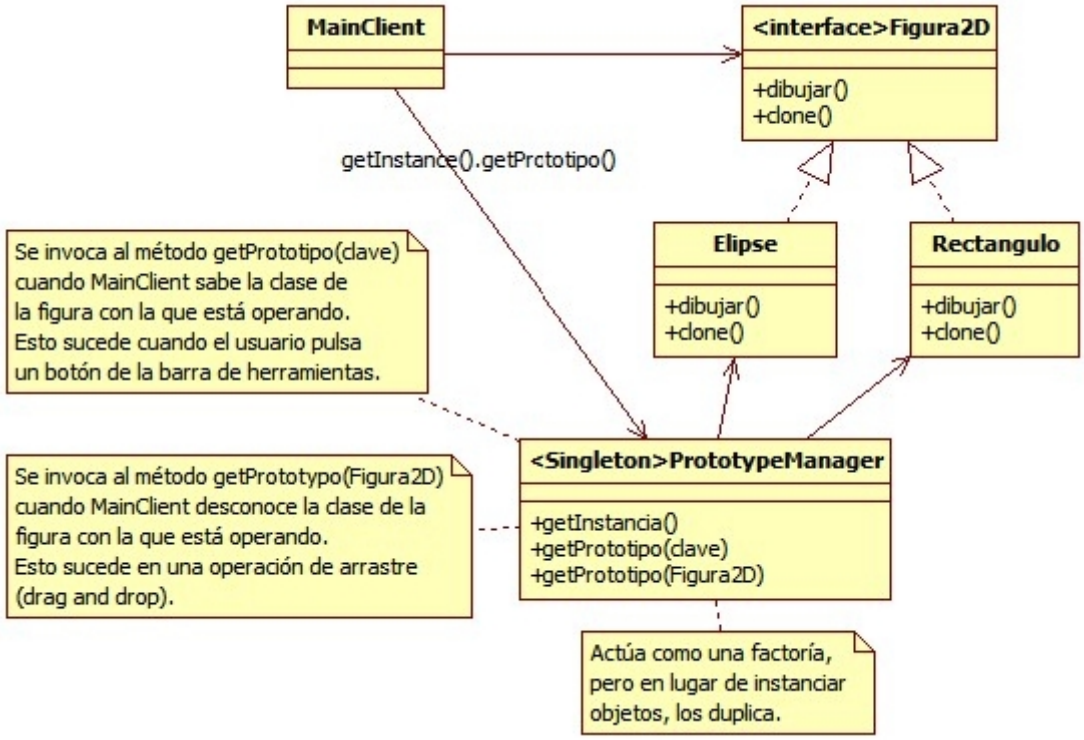
Se trata de un editor gráfico muy simplista que únicamente permite crear elipses y rectángulos, aunque es muy sencillo extender el tipo de figuras que podrían aparecer. Como operaciones, sólo se proporciona el movimiento de las figuras de un punto a otro con el ratón (drag and drop).

La figura siguiente muestra el aspecto que presentará la aplicación: Un botón por cada tipo de figura que podemos crear. Para arrastrar una figura sólo tenemos situarnos encima de ella, presionar el botón izquierdo del ratón y movernos hasta la posición deseada. Durante la transición del movimiento la figura original se muestra de color rojo y cuando soltemos el botón desaparecerá.




Como veis no se proporcionan servicios para la gestión de los dibujos: grabar, cargar, etc. Es mejor así para no perder el foco sobre lo que nos interesa en estos momentos: entender el patrón Prototype. Se deja como ejercicio la construcción de una aplicación de dibujo más elaborada.

Diagrama de clases



Notad la importancia de la clase PrototypeManager en el diagrama. Más adelante se comenta detalladamente su función. Sin más dilación pasemos a ver el código de la aplicación.

*Importante: leed con detenimiento los comentarios que aparecen en el código.*

Comenzamos viendo la jerarquía de Prototype.

Figura2D.java

En el patrón tiene el rol *Prototype*. Se trata de la interfaz que define las operaciones que comparten las subclases que la implementan. Desde el punto de vista del patrón objeto de estudio, el método clone() es la operación más importante.

```
package creacionales.prototype;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.geom.RectangularShape;

/*
 * En el patron adopta el papel 'Prototype'
 */
public interface Figura2D {

	public static final short ELIPSE = 1;
	public static final short RECTANGULO = 2;

	// Metodo que pinta una figura mediante
	// el contexto grafico Graphics2D
	public void dibujar(Graphics2D g2);

	// Retorna un objeto del tipo java.awt.geom.RectangularShape
	// que toda clase ConcretePrototype tiene como atributo
	public RectangularShape getFigura();

	// Metodo para la duplicacion de instancias
	public Figura2D clone();

	// metodo para establecer el color de una figura
	public void setColor(Color color);

}
```

Ahora veamos las figuras concretas. En el patrón tiene el rol *ConcretePrototype*, por lo que su objetivo es duplicarse a sí mismas y devolver la copia a quien la solicite.

Estas clases son formas que el usuario puede seleccionar de una barra de herramientas para que se dibujen en el editor gráfico.

Es importante advertir que no se utiliza el mecanismo de clonado proporcionado por Java mediante la interfaz Clone, sino que la duplicación se lleva a cabo mediante un constructor de copia, esto es, el método clone() crea un nueva instancia e inicializa sus atributos con los valores de la instancia actual. Se ha optado por este diseño a causa de los numerosos problemas que presenta el mecanismo de clonado estándar de Java.

Especial atención merece la copia del atributo Ellipse2D, cuyo tipo es del API java.awt.geom y el objeto que al fin y al cabo aparece dibujado en el editor. Este atributo es un objeto, por lo también se tiene que crear una nueva instancia de él para el objeto clonado, pues en caso contrario tendríamos a dos objetos Elipse apuntando al mismo atributo de tipo Ellipse2D. Este tipo de copia se conoce como copia profunda, en contraposición a la copia superficial (en la última sección de documento se explican las diferencias entre estas dos modalidades de copia).

Elipse.java

```
package creacionales.prototype;

import java.awt.geom.Ellipse2D;
import java.awt.geom.RectangularShape;
import java.awt.Graphics2D;
import java.awt.Color;

/*
 * En el patron adopta el papel 'ConcretePrototype'
 */
class Elipse implements Figura2D {

	Ellipse2D.Double elipse;
	private Color color;

	Elipse(int x, int y, int ancho, int alto, Color color) {

		elipse = new Ellipse2D.Double(x, y, ancho, alto);
		this.color = color;
	}

	/*
	 * Constructor de copia
	 */
	Elipse(Elipse e) {
		elipse = (Ellipse2D.Double) e.getFigura().clone();
		elipse.x = e.getFigura().getX();
		elipse.y = e.getFigura().getY();
		elipse.width = e.getFigura().getWidth();
		elipse.height = e.getFigura().getHeight();
		color = e.color;
	}
```

```
	public void dibujar(Graphics2D g2) {
		g2.setColor(color);
		g2.draw(elipse);
	}

	@Override
	public RectangularShape getFigura() {
		return elipse;
	}

	@Override
	public Figura2D clone() {
		return new Elipse(this);
	}

	@Override
	public void setColor(Color color) {
		this.color = color;
	}

}
```

Rectangulo.java

```
package creacionales.prototype;

import java.awt.geom.Rectangle2D;
import java.awt.geom.RectangularShape;
import java.awt.Graphics2D;
import java.awt.Color;

/*
 * En el patron adopta el papel 'ConcretePrototype'
 */
class Rectangulo implements Figura2D {

	private Rectangle2D.Double rectangulo;
	private Color color;

	Rectangulo(int x, int y, int ancho, int alto, Color color) {

		this.rectangulo = new Rectangle2D.Double(x, y, ancho,
alto);
		this.color = color;
	}

	/*
	 * Constructor de copia
	 */
	Rectangulo(Rectangulo r) {
		this.rectangulo = (Rectangle2D.Double)
r.getFigura().clone();
		this.rectangulo.x = r.getFigura().getX();
		this.rectangulo.y = r.getFigura().getY();
		this.rectangulo.width = r.getFigura().getWidth();
		this.rectangulo.height = r.getFigura().getHeight();
```

```
            color = r.color;
        }

        public void dibujar(Graphics2D g2) {
            g2.setColor(color);
            g2.draw(rectangulo);
        }

        @Override
        public RectangularShape getFigura() {
            return rectangulo;
        }

        @Override
        public Figura2D clone() {
            return new Rectangulo(this);
        }

        @Override
        public void setColor(Color color) {
            this.color = color;
        }
}
```

Una vez vista la jerarquía de prototipos, pasamos a ver una clase factoría (y singleton) que quizá nos sorprenda, ya que por lo explicado hasta el momento no se podría haber deducido fácilmente su utilidad. A esta clase le hemos llamado PrototypeManager y no aparece en el diagrama de clases estándar del patrón Prototype (aunque sí el del editor gráfico), pues se trata de una clase de conveniencia que puede interesar en algunos casos y no en otros. La función de PrototypeManager no es otra que crear una instancia de cada ConcretePrototype, almacenarlas en un HashMap (cachearlas) y utilizarlas de molde para obtener duplicados según le soliciten las clases cliente.

Cada vez que un usuario selecciona una elipse o un rectángulo de la barra de herramientas, se está solicitando a PrototypeManager la obtención de una nueva instancia de ese objeto, la cual se obtiene por clonación. Notad que la clase cliente que hace la solicitud pasa un token para informar sobre objeto quiere (Figura2D.ELIPSE o Figura2D.RECTANGULO).

Cada vez que un usuario realiza una operación de arrastre (drag and drop) está solicitando a PrototypeManager una nueva figura de las mismas características que la que debe arrastrar. En este caso la clase cliente que hace la solicitud desconoce el tipo exacto de la clase con la que está operando, por lo que le dice “pásame un duplicado de este objeto”.

PrototypeManager.java

```
package creacionales.prototype;

import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

/*
 * Factoria de Prototipos, implementada como un Singleton.
 *
 * Define operaciones públicas para obtener:
 * -la instancia Singleton.
 * -un duplicado de algun ConcretePrototype en funcion de
 * un token que se utiliza como clave del HashMap o de un
 * valor de un subtipo de Figura2D que permitirá localizar
 * la entrada correspondiente en el HashMap.
 *
 * La primera vez que invoca a PrototypeManager se deben
 * crear las ConcretePrototype, por lo que sera un poco
 * lento, quizas. En cambio el resto de  llamadas seran
 * muy rapidas porque las ConcretePrototype se obtienen
 * del HashMap
 */
public class PrototypeManager {
	private static Map<Short, Figura2D> prototipos;
	private static PrototypeManager instancia =
		new PrototypeManager();

	// Retornar la instancia Singleton
	public static PrototypeManager getInstancia() {
		return instancia;
	}

	/*
	 * Constructor privado.
	 * Se instancia el HashMap y se añade una instancia
	 * de cada ConcretePrototype.
	 *
	 * Estas instancias no se llegan a representar en el
	 * editor grafico, ya que tan solo sirven del molde
	 * para obtener duplicados.
	 */
	private PrototypeManager() {
		prototipos = new HashMap<Short, Figura2D>();
		prototipos.put(Figura2D.ELIPSE,
				new Elipse(100, 100, 50, 20, Color.BLUE));
		prototipos.put(Figura2D.RECTANGULO,
				new Rectangulo(100, 400, 50, 20, Color.BLUE));
	}

	// Retornar un duplicado del ConcretePrototype
	// a partir de una clave
	public Figura2D getPrototipo(short clave) {
		return prototipos.get(clave).clone();
	}
```

```
	// Retornar un duplicado del ConcretePrototype
	// a partir de un subtipo de Figura2D
	public Figura2D getPrototipo(Figura2D valor) {

		// Recorrer el mapa hasta encontrar una entrada cuya clase
		// coincida con la clase del objeto pasado por parametro
		for (Entry<Short, Figura2D> e : prototipos.entrySet()) {
			// Obtener la clase de la entrada actual
			Class<?> claseProto = e.getValue().getClass();
			// ¿Coincide con la clase del parametro?
			if (claseProto.equals(valor.getClass())) {

				// Devolver el objeto asociado a la clave de la
entrada actual
				return prototipos.get(e.getKey()).clone();
			}
		}
		throw new RuntimeException("PropertyManager no ha podido
recuperar un prototipo a partir del objeto especificado por
parametro.");
	}
}
```

Y llegamos a la clase cliente: el editor gráfico.

Es importante observar en esta clase los dos casos en los que se obtienen duplicados de Elipse y de Rectangulo a través de ProtoypeManager:

- Cuando el editor sabe la figura que necesita, esto es, cuando el usuario pulsa sobre un botón en concreto de la barra de herramientas.
- Cuando el editor desconoce con qué clase está trabajando en un momento dado. Este caso se produce cuando el usuario realiza una operación de arrastre (drag and drop).

MainClient.java

```
package creacionales.prototype.client;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import java.awt.geom.Point2D;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
```

```java
import creacionales.prototype.Figura2D;
import creacionales.prototype.PrototypeManager;

import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Color;
import java.awt.Point;
import java.util.LinkedList;
import java.util.List;

/*
 * Editor Grafico muy simple que ilustra el funcionamiento del
 * patron Prototype. Adopta el papel 'Client' en el patrón.
 *
 * Contiene una barra de herramientas que permite añadir al
 * editor elipses y rectangulos. La unica operacion que
 * podemos hacer con estas figuras es moverlas.
 *
 * Notad que esta clase no trabaja con ninguna clase concreta,
 * sólo con el tipo interfaz Figura2D.
 */
public class MainClient extends JFrame {

	private static final long serialVersionUID = 1L;
	private static final int FRAME_WIDTH = 450, FRAME_HEIGHT = 450;

	private Figura2D figuraSeleccionada, figuraEnTransito;
	private Point2D inicioDrag;
	private JButton btnRectangulo, btnElipse;

	// Lista para almacenar las figuras creadas por el usuario
	private static List<Figura2D> figuras =
		new LinkedList<Figura2D>();

	public static List<Figura2D> getFiguras() {
		return figuras;
	}

	/*
	 * Metodo main
	 */
	public static void main(String[] args) {
		SwingUtilities.invokeLater(new Runnable() {
			public void run() {
				MainClient inst = new MainClient();
				inst.setLocationRelativeTo(null);
				inst.setVisible(true);
			}
		});
	}

	/*
	 * Constructor
	 */
	public MainClient() {
		super();
		initGUI();
		inicializarListeners(); // Manejadores de eventos de raton
	}
```

```
private void initGUI() {
	/*
	 * Creamos la barra de herramientas y la posicionamos en el
	 * formulario. Se crean los manejadores de eventos de boton
	 * para crear figuras
	 */
	crearBarraHerramientas();

	setSize(FRAME_WIDTH, FRAME_HEIGHT);
	setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

/*
 * Metodo invocado por la MVJ para pintar graficos.
 * Al arrancar la aplicacion la MVJ llama una vez
 * a este metodo. Nosotros somos responsables de
 * llamarlo mediante repaint() cuando necesitemos
 * reflejar algún cambio en el escenario.
 */
public void paint(Graphics g) {
	super.paint(g);
	Graphics2D g2 = (Graphics2D) g;

	// Si hay figuras que pintar...que se pinten
	if (!figuras.isEmpty()) {
		for (Figura2D figura : figuras) {
			if (figura != null) {
				figura.dibujar(g2);
			}
		}
	}
}

/*
 * Crear los botones de la barra de herramientas
 * y configurar el comportamiento de los botones.
 *
 * Notad que se llama a la factoria PrototypeManager
 * para obtener una instancia de cada ConcretePrototype.
 * Estas instancias quedan cacheadas en un HashMap. Las
 * utilizamos como moldes para crear duplicados. De esta
 * manera si, por ejemplo, un usurio añade 3 elipses cada
 * una de las figuras será un objeto independiente (con
 * su propia direccion de memoria) del objeto real
 * almancenado en el HashMap.
 */
private void crearBarraHerramientas() {
	JPanel barra = new JPanel();
	// Boton para crear elipses
	btnElipse = new JButton("Elipse");
	btnElipse.addActionListener(new ActionListener() {
		@Override
		public void actionPerformed(ActionEvent e) {

			Figura2D elipse =
				PrototypeManager.getInstancia()
					.getPrototipo(Figura2D.ELIPSE);
			// añadir la figura a la lista
```

```
                        figuras.add(elipse);
                        repaint(); // mostrar los cambios al usuario

                  }
            });

            // Boton para crear Rectangulos
            btnRectangulo = new JButton("Rectangulo");
            btnRectangulo.addActionListener(new ActionListener() {
                  @Override
                  public void actionPerformed(ActionEvent event) {
                        Figura2D rectangulo =
                              PrototypeManager.getInstancia()
                                    .getPrototipo(Figura2D.RECTANGULO);
                        figuras.add(rectangulo);
                        repaint();
                  }
            });

            barra.add(btnElipse);
            barra.add(btnRectangulo);
            add(barra);
      }

      /*
       * Capturamos los eventos:
       * pulsar/soltar botón ratón
       * arrastrar ratón con botón pulsado (drag)
       */
      private void inicializarListeners() {
            addMouseListener(new MouseAdapter() {
                  // Cuando el usuario pulsa el boton del raton
                  @Override
                  public void mousePressed(MouseEvent e) {
                        // obtenemos la coordenada pulsada
                        inicioDrag = new Point(e.getX(), e.getY());
                        // revisamos si en esa coordenada se encuentra
                        // alguna de nuestras figuras
                        for (Figura2D figura : figuras) {
                              if (figura != null) {
                                    // Cuando hay coincidencia,
pintamos la
                                    // figura de rojo
                                    if
(figura.getFigura().contains(inicioDrag)) {
                                          figuraSeleccionada = figura;
                                          figura.setColor(Color.RED);
                                          repaint(); // para que el
usuario vea el cambio
                                          break;
                                    }
                              }
                        }

                  }

                  // Cuando el usuario suelta el boton del raton
                  @Override
                  public void mouseReleased(MouseEvent e) {
```

```
                        // Si se estaba arrastrando alguna figura
                        if (figuraEnTransito != null) {
                              // la pintamos del color original
                              figuraEnTransito.setColor(Color.BLUE);

                              /*
                               * Quitamos de la lista la figura
original, pues ya no es
                               * necesario mostrarle al usuario la
posicion de la figura
                               * antes de soltarla en una nueva
posicion
                               */
                              figuras.remove(figuraSeleccionada);
                              // Todo a null para cuando haya que
comenzar otro drag & drop
                              figuraSeleccionada = null;
                              figuraEnTransito = null;
                              inicioDrag = null;
                              repaint(); // para que el usuario vea el
cambio
                        }
                  }
            });

            addMouseMotionListener(new MouseMotionAdapter() {
                  // Cuando se está produciendo el drag
                  @Override
                  public void mouseDragged(MouseEvent e) {
                        // Si hay seleccionada alguna figura
                        if (figuraSeleccionada != null) {
                              /*
                               * Si aun no ha comenzado el arrastre
tenemos que
                               * obtener un duplicado de la figura y
añadirla
                               * a la lista para que paint() la muestre
                               */
                              if (figuraEnTransito == null) {


                                    /*
                                     * Aqui no sabemos el tipo de
figura que queremos.
                                     * Por tanto, llamamos al método
sobrecargado de
                                     * PrototypeManager que acepta un
subtipo de
                                     * Figura2D como argumento.
                                     */

                                    figuraEnTransito = PrototypeManager
                                                .getInstancia().getPrototipo(
figuraSeleccionada);

                                    figuras.add(figuraEnTransito);

                              }
                              // Cambiar coordendas del duplicado
```

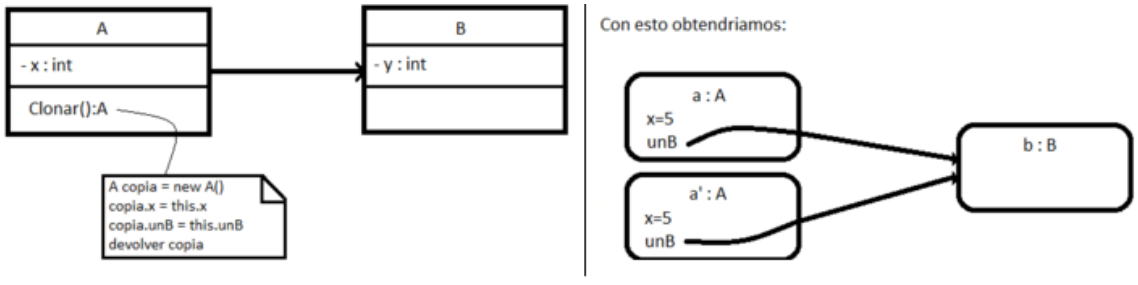
```
                                    // según la posición actual del raton
                                    figuraEnTransito.getFigura().setFrame(
                                                e.getX(), e.getY(),

        figuraEnTransito.getFigura().getWidth(),

        figuraEnTransito.getFigura().getHeight());
                                    // para que el usuario vea el cambio
                                    repaint();
                            }
                        }
                });
        }
}
```

## Problemas específicos e implementación

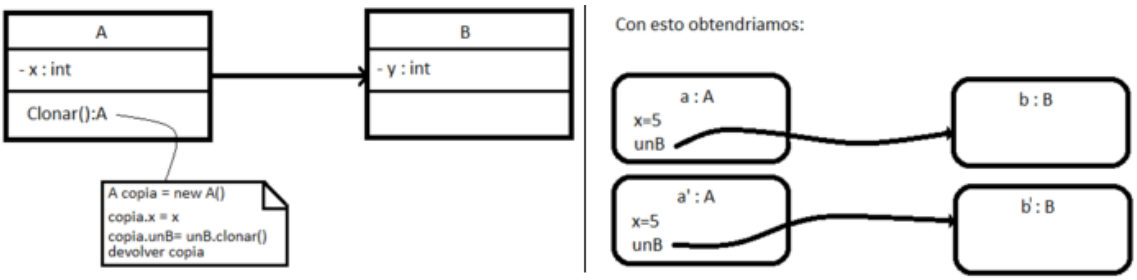
*Copia superficial vs Copia profunda*

Entre las diferentes modalidades entre las que se puede optar a la hora de implementar la clonación de un objeto prototipo, cabe destacar dos maneras de realizar la clonación:

Copia superficial (shallow copying): Con este tipo de copia un cambio sobre el objeto (B) asociado con un clon (A') afecta al objeto original (A), porque el objeto relacionado (B) es el mismo. Es decir, el mecanismo de clonación replica sólo el propio objeto y su estado (sus atributos de tipo primitivo), no sus asociaciones con terceros objetos. Si un atributo de tipo objeto (B) del objeto original (A) es inmutable, entonces no hay problema porque se realice una copia superficial, ya que ningún clon podrá modificarlo.




Copia profunda (deep copying): Se clonan los objetos y también sus objetos relacionados (sus atributos de tipo objeto). En este caso cada clon tiene su propia versión del objeto relacionado, por lo tanto su modificación sólo afecta al clon relacionado pero no al resto de duplicados.




*Nota: En el ejemplo del editor gráfico hemos utilizado la técnica de la copia profunda.*

**Uso de un PrototypeManager**

Suele resultar interesante disponer de una clase Factoría (además, Singleton) que contenga en un HashMap los objetos a clonar, esto es, los moldes (una instancia cacheada de cada tipo concreto de objeto replicable). En otras palabras, PrototypeManager es un registro de los prototipos, que ofrece operaciones para obtener un duplicado a partir de una clave, eliminar un molde del registro, etc.

Las clases cliente utilizarán PrototypeManager para manejar los prototipos en tiempo de ejecución. Hay que tener en cuenta que el uso de esta clase implica que una clase cliente sabe lo que necesita, ya que tiene que proporcionar una clave para obtener un duplicado (para eliminar un molde no sería necesaria una clave, dado que los Map permite eliminar una entrada en la tabla a partir de.

Al utilizar PrototypeManager el patrón Prototype se convierte en un patrón Factory Method que emplea clonación en lugar de instanciación.

*Nota: En el ejemplo del editor gráfico hemos utilizado un PrototypeManager para obtener los duplicados.*

## Patrones relacionados

Aunque Protoype y Abstract Factory son patrones que compiten en ciertas aplicaciones, también es verdad que pueden llegar a utilizarse juntos. Por ejemplo, una Abstract Factory podría almacenarse como un conjunto de Prototypes que servirían para clonar objetos.

Los diseños que hacen un uso intensivo de los patrones Composite y Decorator suelen beneficiarse también del patrón Prototype.