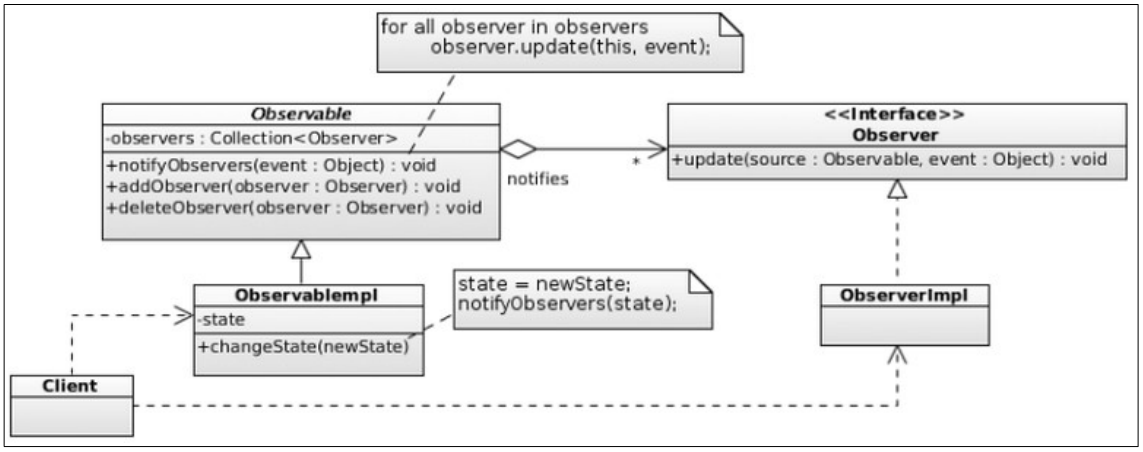
Observer

Diagrama de clases e interfaces



Intención

El patrón Observer proporciona un mecanismo para que ciertos objetos (los observadores) sean notificados sobre los cambios producidos en otro objeto (el objeto observado).

Motivación

El patrón Observer es el adecuado cuando se requiere que un objeto sea observado por uno o varios observadores.

Para entender la utilidad del patrón veamos un ejemplo. Supongamos el siguiente escenario:

Una empresa utiliza una aplicación tipo ERP para crear las ventas de sus productos. Esta aplicación genera un fichero con las líneas de venta. Así, cada vez que se genera una nueva venta, ésta aparece en tal fichero.

Nos han pedido que creemos una aplicación que lea periódicamente el fichero con las lineas de venta, calcule el total y lo muestre a una aplicación GUI llamada 'Cuadro de mandos de Ventas'.

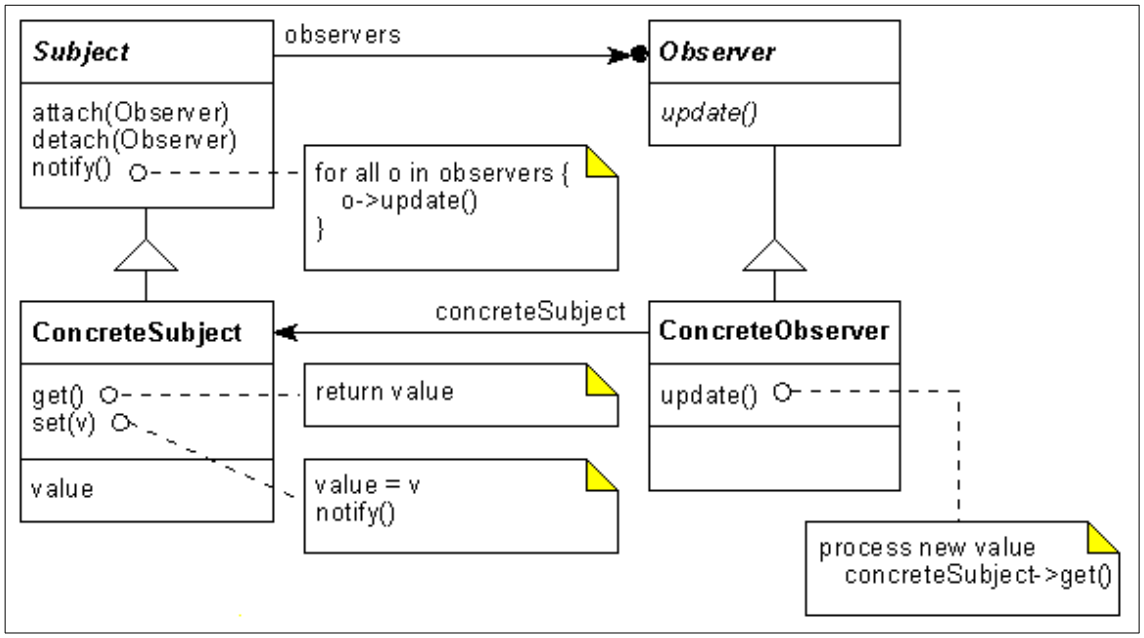
A nivel de componentes, podríamos crear una clase de negocio llamada GestorVentas y una clase GUI llamada CuadroMandosVentasSwing.

Una primera aproximación consistiría en hacer que en cuanto GestorVentas recalculase las ventas, enviase un mensaje a CuadroMandosVentas, pasándole el nuevo valor para que actualice la información que muestra. Sin embargo, esto tiene un claro inconveniente: estamos acoplando más de lo debido estas dos clases, es decir estamos haciendo que el Modelo conozca detalles de la Vista. Los principio de diseño nos disuaden de este tipo de prácticas, y con razón, ya que si implementamos esta aproximación ¿qué pasaría si nos pidiesen que el cuadro de mandos también tiene que estar disponible como página Web? En tal caso tendríamos que modificar la clase GestorVentas y hacer que también envíe el mensaje de actualización de ventas a la capa Web. Claramente es un mal diseño, por tanto, las clases del Modelo no deberían conocer siguiera a las clases de las Vista.

Para solucionar el problema anterior se puede utilizar el patrón Observer. Para ello, definimos una interfaz “publicador”, que será implementada por el objeto responsable de notificar cambios en su estado. En nuestro ejemplo, esta clase será la clase del Modelo: GestorVentas. Por otro lado, definimos una interfaz “suscriptor”, que será implementada por los objetos interesados en ser notificados, que en nuestro caso serán las clases de la Vista: CuadroMandosVentasSwing, CuadroMandosVentasJSP, etc. Además, el “publicador” puede registrar dinámicamente suscriptores que están interesados en un evento y notificarles cuando ocurre el evento. Con esta solución, GestorVentas queda acoplada a la interfaz “suscriptor”, pero no a ninguna clase concreta que la implemente. Claramente, se trata de un diseño mejor que el propuesto inicialmente, pues separa el Modelo de la Vista, manteniendo variaciones protegidas con respecto a cambios en la interfaz de usuario.

Implementación

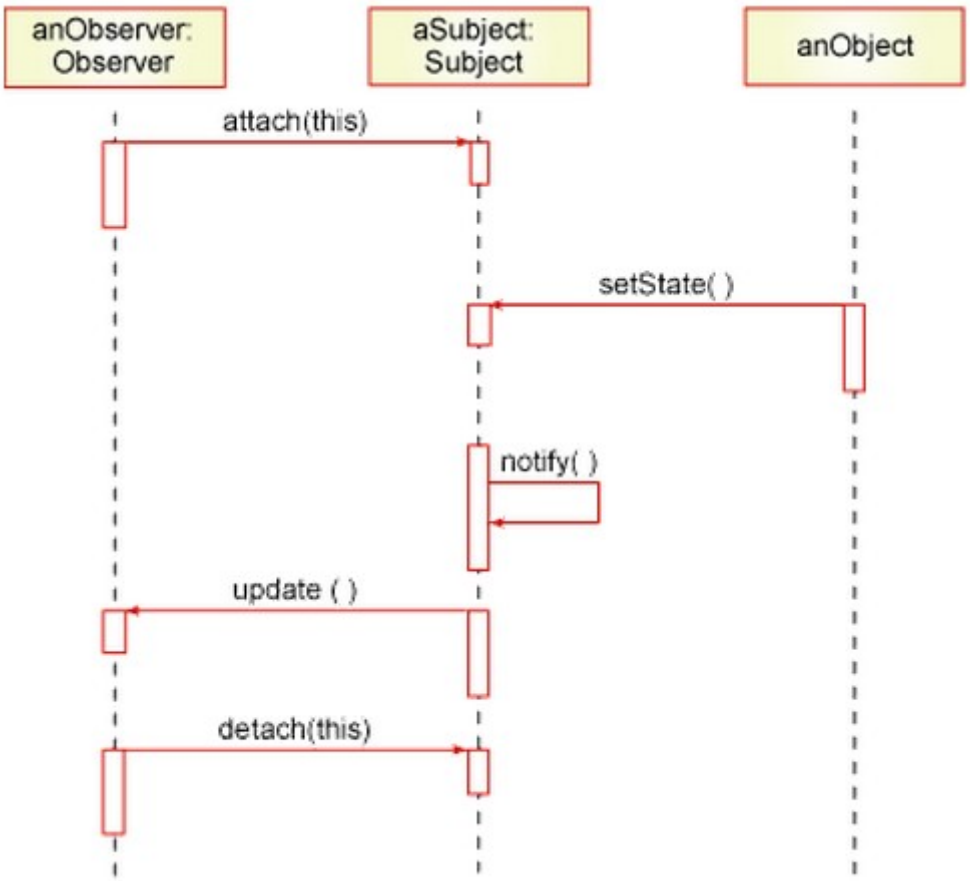
El funcionamiento del patrón es el mostrado en el diagrama de clases siguiente:



Las clases participantes en el patrón son las siguientes:

- **Subject:** Proporciona una interfaz para registrar y desregistrar observadores. Conoce a sus observadores, por lo que también dispone de un método para notificarles un evento.
- **Observer**: Define una interfaz de actualización mediante la que los observadores reciben las notificaciones de cambio en el sujeto.
- **ConcreteSubject**: Contiene información de interés para objetos de la clase ConcreteObserver y les envía notificaciones cuando esta información cambia.
- **ConcreteObserver**: Mantiene una referencia al objeto ConcreteSubject. Dispone de información que debe mantenerse consistente con la información de ConcreteSubject. Implementa la interfaz Observer para ser notificado de cambios en la información de interés.

La siguiente figura muestra el diagrama de secuencias del patrón:



Los observadores quieren ser informados de cambios en el estado del ConcreteSubject. Para ello, lo primero es que se registren en el ConcreteSubject. Con esto, el ConcreteSubject dispone de una lista con todos los observadores que tendrán que ser notificados cuando cambie alguna parte o todo su estado. En algún momento cambiará el estado del ConcreteSubject, por lo que éste recorrerá la lista de observadores, invocando al método update() de cada uno de ellos. La implementación del método update(), básicamente sirve para que cada observador actualice su estado con el del ConcreteSubject.

Aplicabilidad y Ejemplos

El patrón Observer es adecuado cuando:

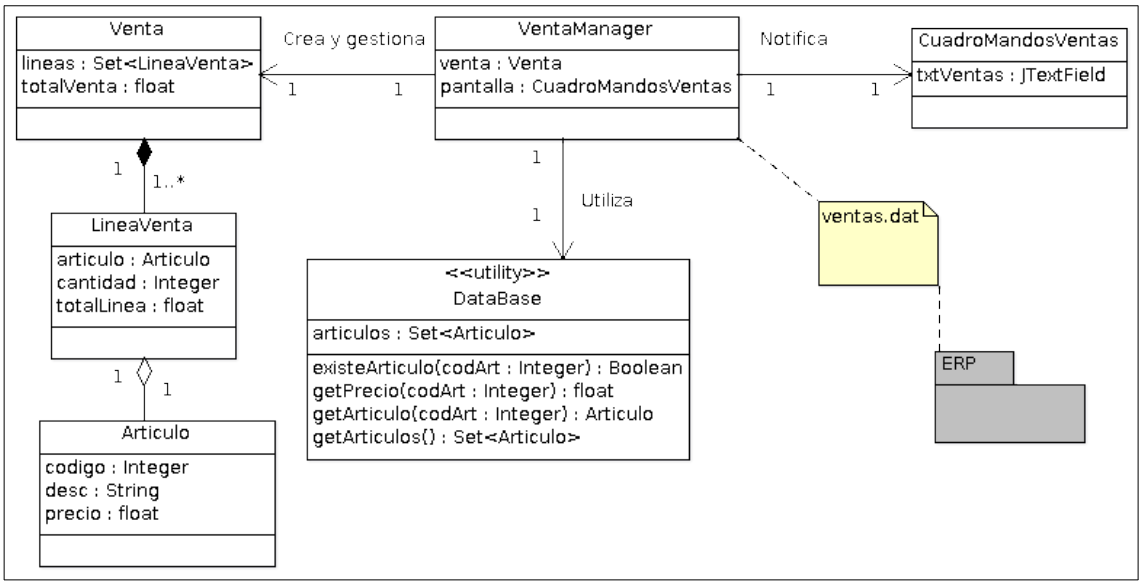
- Se necesita acoplamiento abstracto entre el sujeto y los observadores. El patrón asegura un bajo acoplamiento entre estas clases, proporcionando así al sujeto un mecanismo para protegerse respecto a variaciones en el tipo de observadores.
- Se necesita comunicar a un número indeterminado de observadores. El patrón permite el registro y el desregistro dinámico de observadores. Por tanto, el sujeto nunca sabe de antemano a qué observadores tiene que notificar.

Veamos ahora algunos ejemplos.

Ejemplo 1 – Gestor de ventas acoplado a la vista

Retomando el enunciado del apartado *Motivación*, vamos a ver primero el código sin aplicar el patrón Observer y después cambiaremos algunas clases para obtener un diseño que implemente el patrón.

La siguiente figura muestra las clases que intervienen en el programa:



Notad que el módulo ERP representa un sistema de terceros, del cual sólo sabemos que es responsable de escribir las nuevas ventas en el fichero 'ventas.dat'.

A tener en cuenta:

- La clase DataBase pertenece a la capa de persistencia.
- La clase CuadroMandosVentas pertenece a la capa de presentación (Vista).
- El resto de clases pertenecen a la capa de dominio (Modelo).

Cada una de ellas se va a ir explicando a continuación, tal como se vaya mostrando el código. Comenzamos por las tres clases de entidad: Articulo, LineaVenta y Venta.

Articulo.java

Sencillamente representa la entidad articulo:

```
package comportamiento.observer.ventas_no_patron;


public class Articulo {


	private int codigo;
	private String desc;
	private float precio;

	public Articulo(int codigo, String desc, float precio) {
		this.codigo = codigo;
		this.desc = desc;
		this.precio = precio;
	}

	public int getCodigo() { return codigo; }
	public String getDesc() { return desc; }
	public float getPrecio() { return precio;}
```

```
	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result + codigo;
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Articulo other = (Articulo) obj;
		if (codigo != other.codigo)
			return false;
		return true;
	}

	@Override
	public String toString() {
		return "Articulo [codigo=" + codigo + ", desc=" + desc +
", precio="
				+ precio + "]";
	}

}
```

Notad que se ha redefinido el método equals() y hashCode() heredados de Object. Esto es muy importante cuando necesitamos hacer comparaciones entre objetos y vamos a almacenar las entidades en algún tipo de colección basada en funciones *hash*.

A continuación veamos la clase LineaVenta:

LineaVenta.java

Esta clase está compuesta por un artículo, un entero indicando la cantidad de este artículo y el precio total (precio unitario del artículo multiplicado por la cantidad). Notad, del gráfico anterior, que la relación entre LineaVenta y Articulo es 1:1 y que el rombo mostrado es de color blanco. Esto indica que el Articulo existe, independientemente de LineaVenta.

```
package comportamiento.observer.ventas_no_patron;


public class LineaVenta {


	private Articulo articulo;
	private int cantidad;
	private float totalLinea;

	public LineaVenta(Articulo articulo, int cantidad) {
		this.articulo = articulo;
		this.cantidad = cantidad;
		this.totalLinea = articulo.getPrecio() * cantidad;
	}

	public Articulo getArticulo() {
		return articulo;
	}
	public int getCantidad() {
```

```
		return cantidad;
	}
	public float getTotalLinea() {
		return totalLinea;
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result
				+ ((articulo == null) ? 0 :
articulo.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;
		LineaVenta other = (LineaVenta) obj;
		if (articulo == null) {
			if (other.articulo != null)
				return false;
		} else if (!articulo.equals(other.articulo))
			return false;
```

```
            return true;
      }

      @Override
      public String toString() {
            return "LineaVenta [articulo=" + articulo + ", cantidad="
+ cantidad
                        + ", totalLinea=" + totalLinea + "]";
      }
}
```

A continuación veamos la clase Venta:

Venta.java

Como ya se ha comentado, una venta se compone de un conjunto de objetos LineaVenta y de un total de venta:

```
package comportamiento.observer.ventas_no_patron;

import java.util.HashSet;
import java.util.Set;

public class Venta {

      private Set<LineaVenta> lineas;

      private float totalVenta;

      public void setTotalVenta(float total) {
            this.totalVenta = total;
      }
```

```
	public Venta() {
		lineas = new HashSet<LineaVenta>();
	}

	public float getTotalVenta() {
		return totalVenta;
	}

	public Set<LineaVenta> getLineas() {
		return lineas;
	}

}
```

Veamos ahora la clase DataBase, cuya única utilidad es hacer la veces de base de datos, evitando así complicar el ejemplo con código para acceder al RDBMS. La clase básicamente crea cuatro artículos y los deja disponibles en un mapa, con la idea de no reconocer ninguna venta cuyos artículos no se correspondan con los del mapa.

DataBase.java

```
package comportamiento.observer.ventas_no_patron;

import java.util.HashSet;
import java.util.Set;

/*
 * Clase que simula una tabla de artículos.
 * Crea algunos artículos y proporciona una
 * serie de métodos estáticos.
```

```
 * En una linea de venta no puede existir un
 * artículo que no esté aquí definido previamente.
 */
public class DataBase {
	private static Set<Articulo> articulos;

	static {
		initDataBase();
	}

	private static void initDataBase() {
		articulos = new HashSet<Articulo>();
		articulos.add(new Articulo(1,"Articulo P1", 2.5F));
		articulos.add(new Articulo(2,"Articulo P2", 4.25F));
		articulos.add(new Articulo(3,"Articulo P3", 1.75F));
		articulos.add(new Articulo(4,"Articulo P4", 2.00F));
	}

	public static boolean existeArticulo(int codigo) {
		for (Articulo articulo : articulos) {
			if (articulo.getCodigo() == codigo) {
				return true;
			}
		}
		return false;
	}

	public static float getPrecio(int codigo) {
		for (Articulo articulo : articulos) {
			if (articulo.getCodigo() == codigo) {
```

```
                        return articulo.getPrecio();
                  }
            }
            return 0;
      }

      public static Articulo getArticulo(int codigo) {
            for (Articulo articulo : articulos) {
                  if (articulo.getCodigo() == codigo) {
                        return articulo;
                  }
            }
            return null;
      }

      public static Set<Articulo> getArticulos() {
            return articulos;
      }
}
```

Veamos a continuación la clase VentasManager, la cual pertenece a la capa de Negocio. VentasManager necesita una referencia a la Venta sobre la que tiene que proporcionar todo el comportamiento necesario: leer periódicamente el fichero de lineas de venta, calcular el total de la ventas, etc.

Por otra parte, para validar una LineaVenta tiene que acceder a los métodos de utilidad de la clase BaseDatos. De esta manera comprueba que las líneas de venta contienen artículos pertenecientes a la empresa.

Finalmente, crea una referencia a la clase CuadroMandosVentas para enviarle periódicamente el nuevo total de venta.

Para que el ejemplo sea operativo hay que considerar lo siguiente:

- VentasManager tiene un método main(), por lo tanto hay que ejecutar esta clase para ver el ejemplo. En este método se crea una instancia de VentasManager, a la par que crea una instancia de Venta.
- Lo anterior provoca que se ejecute el constructor de VentasManager, donde sucede lo siguiente:
    - Se crea la clase CuadroMandosVentas
    - Se llama al método procesar(), el cual lee el fichero 'ventas.dat' y obtiene el total de la venta, el cual se pasa a CuadroMandosVentas para que lo muestre.
    - Se añade una nueva línea de venta al fichero. Esto lo debería hacer el ERP, pero lo hacemos nosotros mismos para comprobar que efectivamente CuadroMandosVentas cambia el valor de la venta cuando ésta cambia.
    - Se vuelve a llamar al método procesar(), pues es quien realmente lee el fichero, recalcula e informa a CuadroMandosVentas.
    - Por último, se elimina la linea de venta anteriormente añadida. De esta manera, podemos repetir cómodamente la ejecución cuantas veces queramos.

El código está bastante comentado, por lo que lo mejor es pasar a verlo:

VentasManager.java

```
package comportamiento.observer.ventas_no_patron;


import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
```

```java
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.UnsupportedEncodingException;
import java.net.URISyntaxException;
import java.net.URL;
import java.util.HashSet;
import java.util.Set;
import java.util.StringTokenizer;

/*
 * Clase que se encarga de gestionar los procesos
 * relaciondos con un objeto Venta:
 * -Crea objetos LineaVenta a partir del fichero externo
 * -Añade al fichero lineas de venta
 * -Calcula el total de la venta
 */
public class VentaManager {

	private Venta venta;
	private CuadroMandosVentas pantalla;

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

	public VentaManager(Venta venta) {
		this.venta = venta;
		pantalla = new CuadroMandosVentas();
		pantalla.setVisible(true);
```

```
            procesar();

            // Simulamos que el sistema externo ha añadido una nueva
línea de venta
            addVenta("4;Articulo P4;6");

            procesar();

            // Antes de finalizar el programa volvemos a quitar la
linea
            // para que todo funcione en la siguiente ejecución
            removeVenta(4);
      }

      /*
       * Se lee el fichero que contiene las lineas de venta y
       * se vuelve a recalcular el total vendido.
       * Se actualiza el cuadro de mandos de ventas, aunque
       * no hayan variado las ventas.
       */
      private void procesar() {
            calcTotal();
            pantalla.setTxtVentas(venta.getTotalVenta() + "");
            // Detenmos la ejecución 5 segundos para que el usuario
            // del cuadro de mandos pueda percibir el cambio
            try {
                  Thread.sleep(5 * 1000);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
      }
```

```
	/*
	 * Invoca al método encargado de procesar
	 * el fichero que contiene las lineas de venta.
	 * Después recorre la colección de objetos
	 * LineaVenta perteneciente a la Venta y calcula
	 * el total general.
	 */
	public void calcTotal() {
		comprobarNuevasVentas();
		Float acumulado = 0F;
		for (LineaVenta linea : venta.getLineas()) {
			acumulado = acumulado + linea.getTotalLinea();
		}
		venta.setTotalVenta(acumulado);
	}

	/*
	 * Abre el fichero que contiene las lineas de venta
	 * y crea a partir de ellas objetos LineaVenta, los
	 * cuales se asocian al objeto Venta gestionado por
	 * esta clase.
	 */
	private void comprobarNuevasVentas() {
		InputStream in = getClass().getResourceAsStream(

	"/comportamiento/observer/ventas_no_patron/ventas.dat");
		BufferedReader bufRdr = null;
		try {
			bufRdr = new BufferedReader(new
InputStreamReader(in, "utf-8"));
```

```
                String linea = null;
                while ((linea = bufRdr.readLine()) != null) {
                      StringTokenizer st = new StringTokenizer(linea,
";");
                      while (st.hasMoreTokens()) {
                            int codigo =
Integer.parseInt(st.nextToken());

                            /*
                             * El codigo de articulo del fichero debe
ser uno de
                             * los artículos definidos en la clase
DataBase, ya que
                             * no podemos vender algo que no
fabricamos.
                             */
                            if (DataBase.existeArticulo(codigo)) {
                                  String desc = st.nextToken();
                                  int cantidad =
Integer.parseInt(st.nextToken());
                                  float precio =
DataBase.getPrecio(codigo);
                                  Articulo articulo = new
Articulo(codigo, desc, precio);
                                  LineaVenta lineaVenta = new
LineaVenta(articulo, cantidad);
                                  venta.getLineas().add(lineaVenta);
                            }
                      }
                }
          } catch (UnsupportedEncodingException e) {
                e.printStackTrace();
          } catch (NumberFormatException e) {
                e.printStackTrace();
```

```
		} catch (IOException e) {
			e.printStackTrace();
		} finally {
			if (bufRdr != null) {
				try {
					bufRdr.close();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
			if (in != null) {
				try {
					in.close();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
		}
	}

	/*
	 * Abre el fichero que contiene las lineas de venta
	 * y añade al final del mismo la linea recibida por
	 * parámetro.
	 */
	public void addVenta(String lineaVentaFile) {
		URL url = getClass().getResource(
"/comportamiento/observer/ventas_no_patron/ventas.dat");
		FileWriter fstream = null;
```

```
		BufferedWriter out = null;
		try {
			File file = new File(url.toURI().getPath());
			fstream = new FileWriter(file, true);
			out = new BufferedWriter(fstream);
			out.write(lineaVentaFile);
		} catch (URISyntaxException e1) {
			e1.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		} finally {
			if (out != null) {
				try {
					out.close();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
			if (fstream != null) {
				try {
					fstream.close();
				} catch (IOException e) {
					e.printStackTrace();
				}
			}
		}
	}

	/*
	 * Elimina del fichero de lineas de venta, la venta
```

```
     * del artículo cuyo código coincida con el parámetro
     * recibido.
     * Para ello, crea un Set de lineas de venta que excluye
     * al artículo en cuestión. Después regenera el fichero
     * de lineas de venta, volcando el contenido del Set.
     */
    public void removeVenta(int codigo) {
          Set<LineaVenta> ventaFile = new HashSet<LineaVenta>();

          // Crea un Set temporal que excluya al articulo a eliminar
          for (LineaVenta linea : venta.getLineas()) {
                if (linea.getArticulo().getCodigo() != codigo) {
                      ventaFile.add(linea);
                }
          }

          URL url = getClass().getResource(
    "/comportamiento/observer/ventas_no_patron/ventas.dat");
          FileWriter fstream = null;
          BufferedWriter out = null;
          try {
                File file = new File(url.toURI().getPath());
                fstream = new FileWriter(file);
                out = new BufferedWriter(fstream);

                for (LineaVenta linea : ventaFile) {
                      int code = linea.getArticulo().getCodigo();
                      String desc = linea.getArticulo().getDesc();
                      int cantidad = linea.getCantidad();
```

```
                        String lineaToFile = code + ";" + desc + ";" +
cantidad + "\n";
                        out.write(lineaToFile);
                  }

            } catch (URISyntaxException e1) {
                  e1.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            } finally {
                  if (out != null) {
                        try {
                              out.close();
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                  }
                  if (fstream != null) {
                        try {
                              fstream.close();
                        } catch (IOException e) {
                              e.printStackTrace();
                        }
                  }
            }
      }
}
```

Por último, vemos la clase CuadroMandosVentas:

CuadroMandosVentas.java

```
package comportamiento.observer.ventas_no_patron;
```

```
import java.awt.EventQueue;

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

public class CuadroMandosVentas extends JFrame {

	private static final long serialVersionUID = 1L;
	private JPanel contentPane;
	private JTextField txtVentas;

	public void setTxtVentas(String ventas) {
		txtVentas.setText(ventas);
	}

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

```
		});
	}

	public CuadroMandosVentas() {
		setTitle("Cuadro de mandos VENTAS");
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 450, 300);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);

		JLabel lblVentas = new JLabel("Ventas");
		lblVentas.setBounds(33, 91, 70, 15);
		contentPane.add(lblVentas);

		txtVentas = new JTextField();
		txtVentas.setBounds(159, 82, 128, 34);
		contentPane.add(txtVentas);
		txtVentas.setColumns(10);
	}
}
```

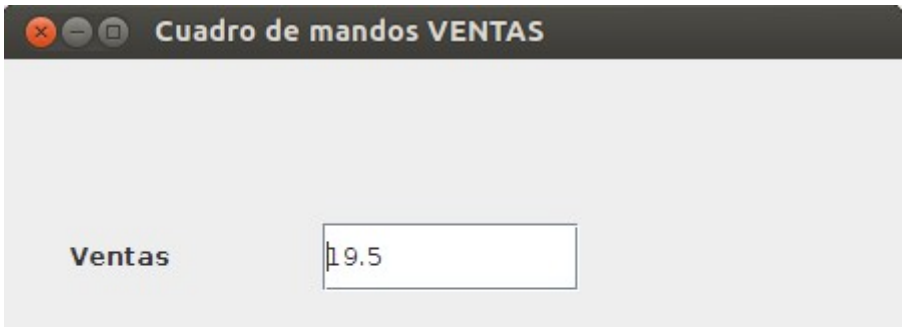
Notad que la clase dispone de un campo llamado txtVentas que es actualizado mediante un método setter(), invocado por la clase VentasManager.

Fichero 'ventas.dat' (debe estar en el mismo paquete que las clases)

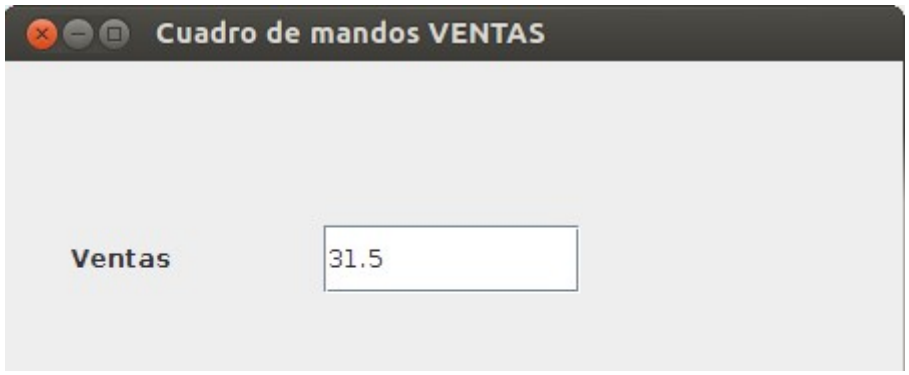
```
1;Articulo P1;2
2;Articulo P2;3
3;Articulo P3;1
```

Nota: Para que todo funcione como se espera, es importante dejar una línea en blanco al final del fichero.

Salida: Al ejecutar la clase VentasManager, veremos la ventana GUI de la clase CuadroMandosVentas. El importe de ventas, si hemos respetado fielmente el código del ejemplo, será de 19.5:



Después de unos cinco segundo veremos como el importe ha cambiado a 31.5:



Aplicando el patrón Observer al código anterior.

A lo largo del documento se ha discutido en más de una ocasión las bondades de aplicar el patrón, por lo que directamente pasamos a ver qué tenemos que cambiar respecto a la versión anterior.

Paso 1

Primero creamos la interfaz para que los observadores puedan recibir notificaciones del sujeto (en nuestro caso, sólo tenemos un observador: la clase CuadroMandosVentas). A la interfaz podemos llamarla PropertyListener y sólo definirá un método: onPropertyEvent():

Nota: He creado un nuevo paquete y he copiado todas las clases del paquete anterior. En este asunto, proceded como queráis.

PropertyListener.java

El método tiene tres parámetros: una referencia al sujeto, el nombre del atributo que ha cambiado y el nuevo valor de tal atributo.

```
package comportamiento.observer.ventas_con_patron;


public interface PropertyListener {

	public void onPropertyEvent(VentaManager mng, String propiedad,
float valor);

}
```

Paso 2

Ahora creamos una interfaz con los métodos necesarios para que el sujeto (VentaManager) pueda registrar observadores, desregistrarlos y enviarles mensajes de actualización.

PropertyPublisher.java

```
package comportamiento.observer.ventas_con_patron;
```

```
public interface PropertyPublisher {
	public void addPropertyListener(PropertyListener listener);
	public void removePropertyListener(PropertyListener listener);
	public void publishPropertyEvent(String nombre, float valor);
}
```

Paso 3

A continuación, hacemos que el sujeto implemente la interfaz definida en el primer punto. Notad que es necesario definir una colección de PropertyListener para poder aplicar a cada elemento cualquiera de los tres métodos de la interfaz PorpertyPublisher. A continuación se muestra el fragmento de código de la clase VentaManager afectado:

VentaManager.java

```
public class VentaManager implements PropertyPublisher {

	private Venta venta;
	// Ya no es necesario
	//private CuadroMandosVentas pantalla;

	private Set<PropertyListener> listeners;

	@Override
	public void addPropertyListener(PropertyListener listener) {
		listeners.add(listener);
	}

	@Override
	public void removePropertyListener(PropertyListener listener) {
```

```
            listeners.remove(listener);
      }

      @Override
      public void publishPropertyEvent(String nombre, float valor) {
            for (PropertyListener listener : listeners) {
                  listener.onPropertyEvent(this, nombre, valor);
            }
      }
      /* Ya no es necesario
       * public static void main(String[] args) {
            new VentaManager(new Venta());
      }*/
      public VentaManager(Venta venta) {
            listeners = new HashSet<PropertyListener>();
            this.venta = venta;
            //pantalla = new CuadroMandosVentas(this);
            //pantalla.setVisible(true);
      }

      public void initProcess() {
            procesar();

            // Simulamos que el sistema externo ha añadido una nueva
línea de venta
            addVenta("4;Articulo P4;6");

            procesar();
```

```
            // Antes de finalizar el programa volvemos a quitar la
linea
            // para que todo funcione en la siguiente ejecución
            removeVenta(4);
            System.out.println("end");
      }

      /*
       * Se lee el fichero que contiene las lineas de venta y
       * se vuelve a recalcular el total vendido.
       * Se actualiza el cuadro de mandos de ventas, aunque
       * no hayan variado las ventas.
       * Después de llamar a calcTotal() hay que publicar la
notificación!
       */
      private void procesar() {
            calcTotal();
            publishPropertyEvent("venta.total",
venta.getTotalVenta());

            //pantalla.setTxtVentas(venta.getTotalVenta() + "");

            // Detenmos la ejecución 5 segundos para que el usuario
            // del cuadro de mandos pueda percibir el cambio
            try {
                  Thread.sleep(5 * 1000);
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
      }
...
```

Paso 4

Hacemos que el observador, CuadroMandoVentas, implemente la interfaz anterior. Con esto, ya no es necesario el método setText() en el observador, por lo que podemos eliminarlo. También es importante advertir que en el constructor, se aprovecha para registrar la instancia de CuadroMandoVentas como oyente de VentaManager. A continuación se muestra el fragmento de código de la clase CuadroMandoVentas afectado:

```
package comportamiento.observer.ventas_con_patron;


import java.awt.EventQueue;


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


public class CuadroMandosVentas extends JFrame implements 
PropertyListener {

	private static final long serialVersionUID = 1L;
	private JPanel contentPane;
	private JTextField txtVentas;

	/* Ya no es necesario
	 * public void setTxtVentas(String ventas) {
	     txtVentas.setText(ventas);
	}*/

	/*
```

```
         * Método que permite recibir notificaciones del sujeto.
         * Notad que el primer parámetro es una referencia al
         * propio sujeto. En ciertas circunstancias puede ser
         * útil, por ejemplo cuando un observador recibe notificaciones
de más de un sujeto, aunque en nuestro caso no la necesitamos
         */
        @Override
        public void onPropertyEvent(VentaManager mng, String propiedad,
float valor) {
              if (propiedad.equals("venta.total")) {
                    txtVentas.setText(valor + "");
                    System.out.println(valor);
              }
        }

        public CuadroMandosVentas(VentaManager mng) {

              // Registrarse como listener en el objeto VentaManager
              mng.addPropertyListener(this);

              setTitle("Cuadro de mandos VENTAS");
              setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
              setBounds(100, 100, 450, 300);
              contentPane = new JPanel();
              contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
              setContentPane(contentPane);
              contentPane.setLayout(null);

              JLabel lblVentas = new JLabel("Ventas");
              lblVentas.setBounds(33, 91, 70, 15);
              contentPane.add(lblVentas);
```

```
            txtVentas = new JTextField("0.00");
            txtVentas.setBounds(159, 82, 128, 34);
            contentPane.add(txtVentas);
            txtVentas.setColumns(10);
      }
...
```

En esta ocasión se ha creado una clase cliente para crear y configurar los objetos necesarios.

MainClient.java

```
package comportamiento.observer.ventas_con_patron;

public class MainClient {

      private Venta venta;
      private VentaManager mng;

      public MainClient() {
            venta = new Venta();
            mng = new VentaManager(venta);

            CuadroMandosVentas frame = new CuadroMandosVentas(mng);
            frame.setVisible(true);

            mng.initProcess();
      }

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

```
        }

}
```

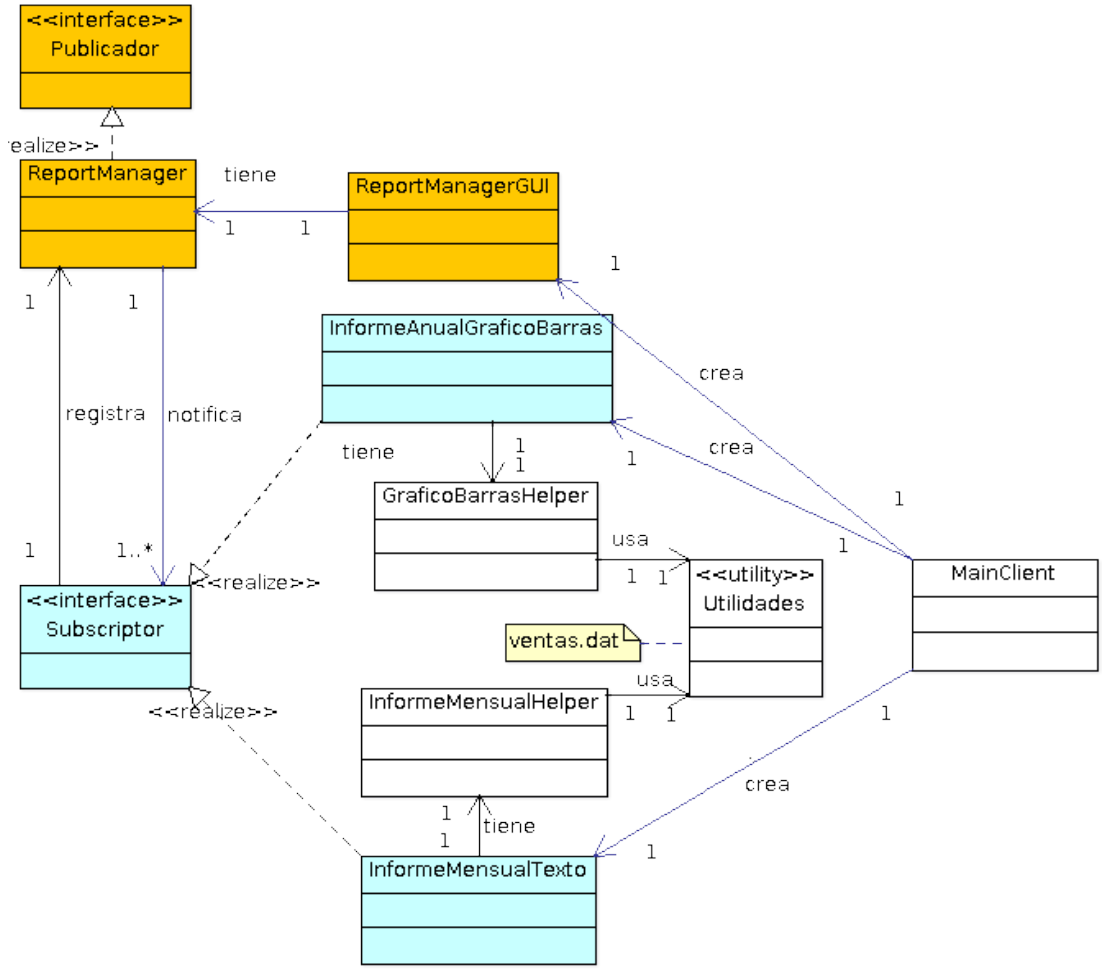
Y ya está. Hemos conseguido que VentaManager ahora no sepa nada sobre CuadroMandosVentas, ni siquiera que sea consciente de su existencia.

En este ejemplo hemos creado nosotros mismos las interfaces para el sujeto y para los observadores, así como la implementación de sus métodos. No obstante, Java ofrece soporte para este patrón, lo cual hace que sea más cómodo para nosotros implementarlo. En el siguiente ejemplo vemos cómo.

Ejemplo 2 – Gestor de informes

El siguiente ejemplo, que es un poco más elaborado que el anterior, trata de una aplicación que permite seleccionar un departamento de una empresa y visualizar dos tipos de informes simultáneamente:

- En un formato textual, las ventas del mes en curso.
- Un gráfico de barras, donde cada barra se corresponde a las ventas de un mes del departamento seleccionado, mostrándose todos los meses hasta el mes en curso.



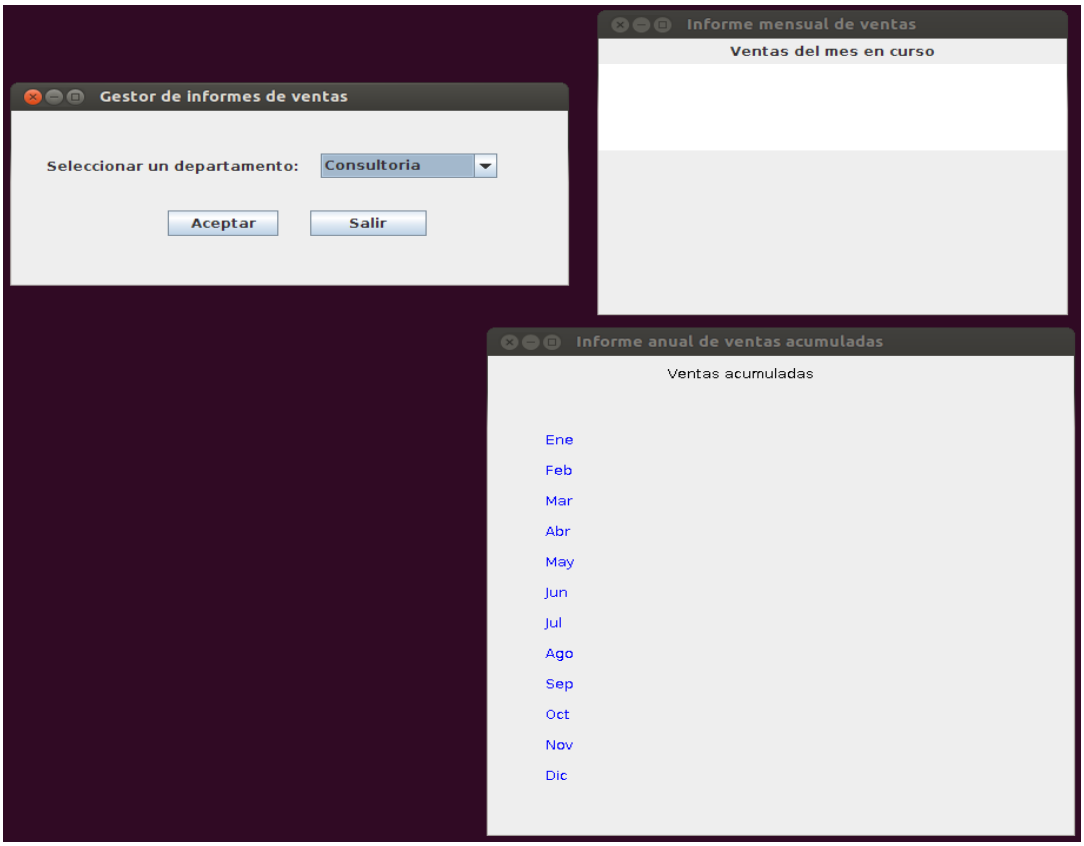
La figura anterior muestra el diagrama de clases de la aplicación.

Explicación de los colores de las clases del diagrama:

- En color naranja se representan las clases y la interfaz que pertenecen a la parte observable o publicadora.
- En color azul se representan las clases y la interfaz que pertenecen a la parte observadora o subscriptora.
- Las clases que no están coloreadas, o bien son clases de soporte para las restantes, o son clases de utilidad.

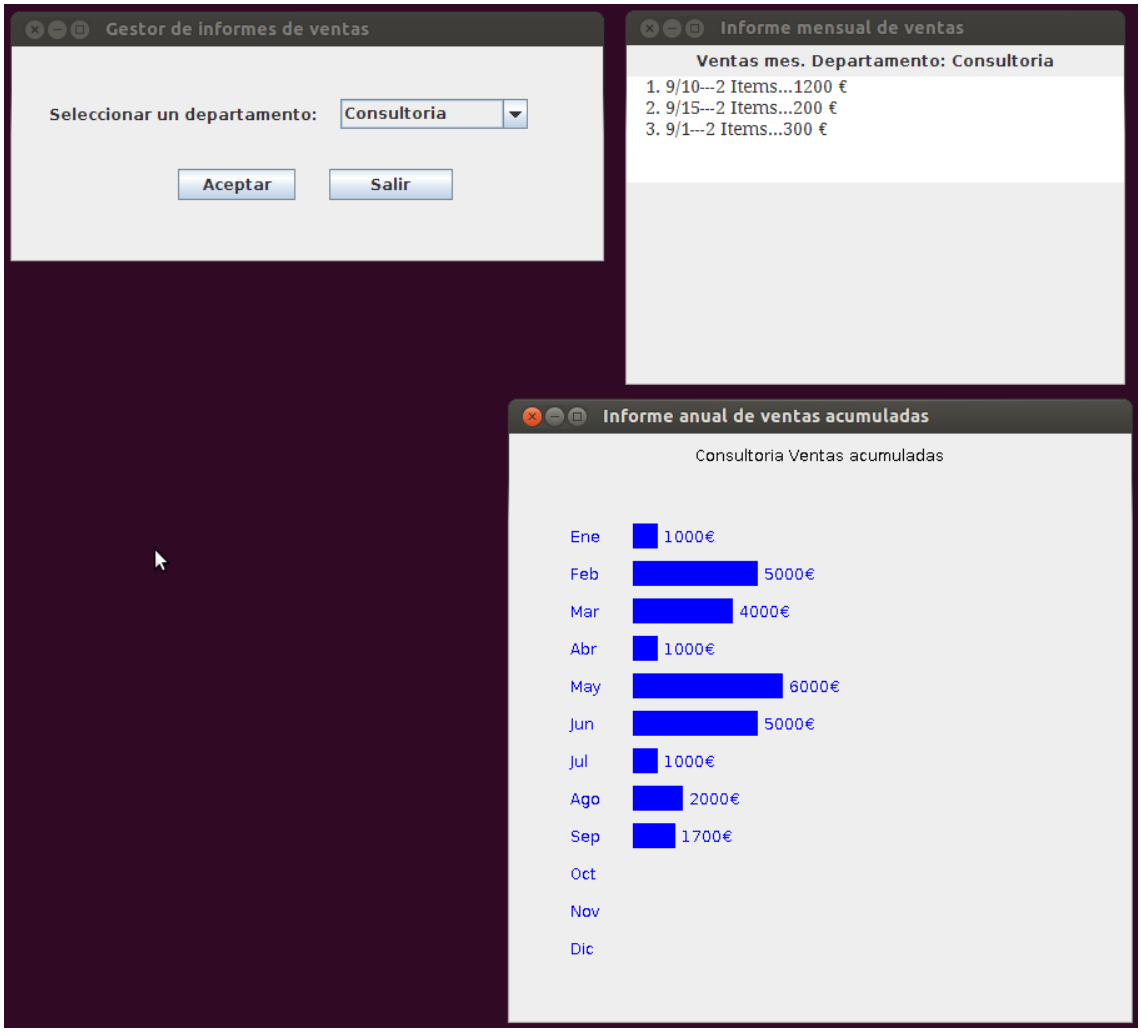
Explicación general

La aplicación se inicia a través de la clase MainClient, la cual crea tres instancias de las clases: ReportManagerGUI, InformeMensualTextual, InformeAnualGraficoBarras. Estas instancias son clases de Swing, por lo que aparecen tres ventanas como las siguientes:



El usuario interacciona con la ventana titulada 'Gestor de informes de ventas', desplegando el combo de departamentos para seleccionar un de ellos y pulsando el botón 'Aceptar' para visualizar los datos en los informes.

La siguiente figura muestra el estado de la aplicación, una vez que el usuario ha seleccionado el departamento de 'consultoría', siendo septiembre el mes actual:



Notad que el informe con el gráfico de barras se detiene en septiembre, ya que es el mes actual.

Como se ha dicho anteriormente, MainClient crea una instancia de ReportManagerGUI, la cual pasa como argumento a las otras dos instancias que crea: una de la clase InformeMensualTextual y otra de la clase InformeAnualGraficoBarras.

El constructor de ReportManagerGUI crea una instancia de ReportManager, que es la clase que implementa la interfaz Publicador, por tanto, ReportManager es capaz de enviar notificaciones de cambio a clases que implementen la interfaz Subscriptor. InformeMensualTextual y InformeAnualGraficoBarras implementan la interfaz Subscriptor.

A destacar que ReportManagerGUI y ReportManager forman un tándem como objeto observable, es decir, son el subject de la aplicación. Esto se debe a que el atributo de

interés para los subscriptores es el 'departamento' de ReportManagerGUI, pero ReportManagerGUI no informa directamente a los subscriptores, ya que no implementa la interfaz Publicador, sino que delega en ReportManager, que sí la implementa.

Así, cuando el usuario selecciona un nuevo departamento, ReportManagerGUI informa a ReportManager, el cual notifica al objeto InformeMensualTextual y al objeto InformeAnualGraficoBarras que deben refrescar su contenido porque el departamento ha cambiado. Ahora bien, ¿cómo sabe ReportManager quienes son estos objetos a los que debe informar? La respuesta es, como se ha comentado anteriormente, que cuando MainClient creó la instancia de ReportManagerGUI, la pasó como argumento a sendas instancias de InformeMensualTextual y de InformeAnualGraficoBarras. Estas instancias, en sus respectivos constructores, extrajeron la referencia a ReportManager a partir de ReportManagerGUI. De esta manara, se registraron como oyentes en ReportManager.

Para actualizar su contenido, cada clase generadora de informes se apoya en sendas clases de soporte. Como se puede apreciar en el diagrama de clases anterior, tenemos que:

- InformeMensualTextual utiliza a InformeMensualHelper
- InformeAnualGraficoBarras utiliza a GraficoBarrasHelper
- Las dos clases helpers utilizan a una tercera clase llamada Utilidades

Esto se ha hecho para separar responsabilidades y dejar así un código más claro en todas estas clases. Mientras que InformeMensualTextual se encarga de dibujar la ventana del informe y de recibir notificaciones de cambio de departamento, InformeMensualHelper se encarga de obtener las ventas para el mes en curso, para lo cual tiene que interactuar con Utilidades para obtener un vector a partir de los datos del fichero 'ventas.dat'. El caso para el informe que dibujar barras es análogo.

Veamos ahora el código:

Publicador.java

```java
package comportamiento.observer.reports;

/**
 * Interfaz a implementar por una clase que quiera
 * publicar notificaciones para objetos subscriptores
 */
public interface Publicador {
	public void notificarSubscriptores();
	public void registrar(Subscriptor subscriptor);
	public void desRegistrar(Subscriptor subscriptor);
}
```

ReportManager.java

```java
package comportamiento.observer.reports;

import java.util.Vector;

/**
 * Subject encargado de notificar a los observadores
 * un cambio en su estado.
 * Esta clase trabaja en conjunto con ReportManagerGUI,
 * ya que los cambios en su atributo 'departamento' es
 * lo que realmente interesa a los observadores.
 */
public class ReportManager implements Publicador {

	private Vector<Subscriptor> observersList;

	public ReportManager() {
```

```
            observersList = new Vector<Subscriptor>();
        }

        @Override
        public void notificarSubscriptores() {
            // Notificar a todos los subscriptores
            for (int i = 0; i < observersList.size(); i++) {
                Subscriptor observer = observersList.get(i);
                observer.refrescarInforme(this);
            }
        }

        @Override
        public void registrar(Subscriptor obs) {
            observersList.add(obs);
        }

        @Override
        public void desRegistrar(Subscriptor obs) {
            observersList.remove(obs);
        }
}
```

ReportManagerGUI.java

```
package comportamiento.observer.reports;

import java.awt.event.KeyEvent;

import javax.swing.JButton;
```

```java
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * Ventana principal de la aplicación, desde la que
 * el usuario puede seleccionar un departamento del
 * combo, para ver los informes.
 * Esta ventana en un wrapper que envuelve al subject,
 * es decir, al objeto ReportManager, que es el objeto
 * observable. No obstante, hay que tener en cuenta que
 * es ReportManagerGUI la que genera los cambios de
 * estado, cuando el usuario selecciona un departamente
 * y pulsa el botón Aceptar.
 */
public class ReportManagerGUI extends JFrame {

	private static final long serialVersionUID = 1L;
	public static final String Fichero = "ventas.dat";

	private String departmento;
	private ReportManager reportManager;

	private JPanel contentPane;
	private JComboBox<String> cmbDepartmentos;
	private JButton btnAceptar, btnSalir;
```

```
private JLabel lblDepartmentos;

public ReportManagerGUI() {
	super("Gestor de informes de ventas");
	configurarGUI();
	reportManager = new ReportManager();
}

private void configurarGUI() {
	preConfigurarPanel();
	configurarEtiqueta();
	configurarCombos();
	configurarBotones();
	addControlesToPanel();
	configurarVentana();
}

public String getDepartmento() {
	return departmento;
}

public void setDepartmento(String dept) {
	departmento = dept;
}

public ReportManager getReportManager() {
	return reportManager;
}

private void preConfigurarPanel() {
```

```
            contentPane = new JPanel();
            contentPane.setLayout(null);
            contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
            setContentPane(contentPane);
      }

      private void configurarEtiqueta() {
            lblDepartmentos = new JLabel("Seleccionar un
departamento:");
            lblDepartmentos.setBounds(30, 47, 215, 15);
      }

      private void configurarCombos() {
            cmbDepartmentos = new JComboBox<String>();
            cmbDepartmentos.setBounds(263, 42, 150, 24);
            cmbDepartmentos.addItem("Consultoria");
            cmbDepartmentos.addItem("Electronica");
            cmbDepartmentos.addItem("Electricidad");
      }

      private void configurarBotones() {
            btnAceptar = new JButton("Aceptar");
            btnAceptar.addActionListener(new AceptarListener());
            btnAceptar.setBounds(133, 98, 94, 25);
            btnAceptar.setMnemonic(KeyEvent.VK_S);
            btnSalir = new JButton("Salir");
            btnSalir.addActionListener(new SalirListener());
            btnSalir.setBounds(254, 98, 99, 25);
            btnSalir.setMnemonic(KeyEvent.VK_X);
      }
```

```
	private void addControlesToPanel() {
		contentPane.add(lblDepartmentos);
		contentPane.add(cmbDepartmentos);
		contentPane.add(btnAceptar);
		contentPane.add(btnSalir);
	}

	private void configurarVentana() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 450, 300);
		setSize(475, 200);
		setLocation(50, 50);
		setVisible(true);
	}

	private class AceptarListener implements ActionListener {
		@Override
		public void actionPerformed(ActionEvent evt) {
			String dept = (String)
cmbDepartmentos.getSelectedItem();
			setDepartmento(dept); // Cambiar el estado de la
clase
			reportManager.notificarSubscriptores(); // Notificar
a los observadores
		}
	}

	private class SalirListener implements ActionListener {
		@Override
		public void actionPerformed(ActionEvent evt) {
```

```
                System.exit(0);
            }
        }
}
```

Subscriptor.java

```
package comportamiento.observer.reports;

/**
 * Interfaz que tienen que implementar aquellas clases interesadas
 * en recibir notificaciones de cambio sobre un objeto observable
 *
 */
public interface Subscriptor {
        public void refrescarInforme(Publicador subject);
}
```

InformeMensualTextual.java

```
package comportamiento.observer.reports;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Font;

import java.util.Vector;

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

```
import javax.swing.JTextArea;


/**
 * Observer que muestra en un JTextArea las ventas del mes
 * en curso para el departamento seleccionado por el usuario
 * en ReportManagerGUI (Subject conjuntamente con ReportManager).
 *
 * Cuando en ReportManagerGUI cambia el departamento, se llama
 * al método refrescarInforme() de esta clase para que actualice
 * el informe.
 */
public class InformeMensualTextual extends JFrame implements
Subscriptor {


	private static final long serialVersionUID = 1L;


	private Container contentPane;
	private JPanel ventasPanel;
	private JLabel lblVentas;
	private JTextArea textAreaVentas;


	// Sostener una referencia al wrapper del Subject
	private ReportManagerGUI reportManagerGUI;


	// Delegar tareas de bajo nivel en un objeto
InformeMensualHelper
	private InformeMensualHelper informeMensualHelper;


	public InformeMensualTextual(ReportManagerGUI reportManagerGUI)
throws Exception {
		super("Informe mensual de ventas");
```

```
            configurarGUI();
            this.reportManagerGUI = reportManagerGUI;
            reportManagerGUI.getReportManager().registrar(this);
            informeMensualHelper = new InformeMensualHelper();
      }

      @Override
      public void refrescarInforme(Publicador subject) {
            // Comprobar la identidad del subject que ha invocado al
método
            if (subject == reportManagerGUI.getReportManager()) {
                  // Obtener el estado del subject
                  String departamento =
reportManagerGUI.getDepartmento().trim();

                  Vector<String> trnList =

      informeMensualHelper.getVentasMesEnCurso(departamento);

                  actualizarGUI(departamento, trnList);
            }
      }

      private void configurarGUI() {
            crearControles();
            crearPaneles();
            configurarVentana();
      }

      private void actualizarGUI(String departamento, Vector<String>
trnList) {
            lblVentas.setText("Ventas mes. Departamento: "
```

```
                    + departamento);
        String content = "";
        for (int i = 0; i < trnList.size(); i++) {
              content = content + trnList.get(i) + "\n";
        }
        textAreaVentas.setText(content);
  }

  private void crearPaneles() {
        ventasPanel = new JPanel();
        ventasPanel.add(lblVentas);
        ventasPanel.add(textAreaVentas);
        contentPane = getContentPane();
        setContentPane(contentPane);
        contentPane.add(ventasPanel, BorderLayout.CENTER);
  }

  private void crearControles() {
        textAreaVentas = new JTextArea(5, 40);
        textAreaVentas.setFont(new Font("Serif", Font.PLAIN, 14));
        textAreaVentas.setLineWrap(true);
        textAreaVentas.setWrapStyleWord(true);
        lblVentas = new JLabel("Ventas del mes en curso");
  }

  private void configurarVentana() {
        setLocation(550, 25);
        setSize(400, 300);
        setVisible(true);
  }
```

```
}
```

InformeMensualHelper.java

```
package comportamiento.observer.reports;

import java.util.StringTokenizer;
import java.util.Vector;

/**
 * Clase se soporte para el observador que se encarga de
 * dibujar el JTextArea con las ventas del mes en curso.
 * Esta clase implementa los detalles de bajo nivel para
 * obtener las ventas, mientras que el observador
 * correspondiente, se encarga de asuntos más generales.
 */
public class InformeMensualHelper {

	public Vector<String> getVentasMesEnCurso(String departamento) {

		Vector<String> ventasTotales =

	Utilidades.fileToVector(ReportManagerGUI.Fichero);

		Vector<String> ventasMes = new Vector<String>();

		// Mes actual
		int mes = Utilidades.getMesEnCurso();

		// Texto a buscar
		String textoBusqueda = departamento + "," + mes + ",";
```

```
            /*
             * Recorrer todas las filas en busca de aquellas que
coincidan con el
             * mes en curso y el departamento seleccionado por el
usuario
             */
            int j = 0;
            for (int i = 0; i < ventasTotales.size(); i++) {
                  String fila = ventasTotales.get(i);
                  if (match(fila, textoBusqueda)) {
                        StringTokenizer st = new StringTokenizer(fila,
",");
                        st.nextToken();
                        fila = "                          " +
                                    ++j + ". " +
                                    st.nextToken() + "/" +
st.nextToken() +
                                    "---" + st.nextToken() + " Items" +
"..."
                                    + st.nextToken() + " €";

                        ventasMes.add(fila);
                  }
            }
            return ventasMes;
      }

      private boolean match(String fila, String textoBusqueda) {
            return fila.indexOf(textoBusqueda) > -1;
      }
}
```

InformeAnualGraficoBarras.java

```
package comportamiento.observer.reports;


import java.awt.*;
import javax.swing.*;


/**
 * Observer que muestra un gráfico de barras de las ventas acumuladas
 * del año, hasta el mes actual, donde cada barra es un mes, para el
 * departamento seleccionado por el usuario en ReportManagerGUI
 * (Subject conjuntamente con ReportManager).
 *
 * Cuando en ReportManagerGUI cambia el departamento, se llama al
método
 * refrescarInforme() de esta clase para que actualice el informe.
 */
public class InformeAnualGraficoBarras extends JFrame implements
Subscriptor {


	private static final long serialVersionUID = 1L;
	private String departamento = "";
	private boolean fromRefresh;


	// Sostener una referencia al wrapper del Subject
	private ReportManagerGUI reportManagerGUI;


	// Delegar tareas de bajo nivel en un objeto GraficoBarrasHelper
	private GraficoBarrasHelper graficoBarrasHelper;
```

```
	public InformeAnualGraficoBarras(ReportManagerGUI
reportManagerGUI)
			throws Exception {
		super("Informe anual de ventas acumuladas");
		configurarVentana();
		this.reportManagerGUI = reportManagerGUI;
		reportManagerGUI.getReportManager().registrar(this);
		graficoBarrasHelper = new GraficoBarrasHelper();
	}

	@Override
	public void refrescarInforme(Publicador subject) {
		// Comprobar la identidad del subject que ha invocado al
método
		if (subject == reportManagerGUI.getReportManager()) {
			/// Obtener el estado del subject
			departamento =
reportManagerGUI.getDepartmento().trim();
			graficoBarrasHelper.setDepartamento(departamento);
			reiniciarGrafico();
			// Establecer flag procedencia refresco para evitar
que el
			// método paint recalcule innecesariamente el gráfico
de barras
			fromRefresh = true;
			repaint(); // petición para que la MVJ llame a
paint()
		}
	}

	@Override
	public void paint(Graphics g) {
		graficoBarrasHelper.dibujarTituloMeses(g);
```

```
            /*
             * Optimizado. Sólo se recalcula cuando la llamada a
paint() procede
             * de refrescarInforme(), es decir, de un cambio de
departamento.
             */
            if (fromRefresh) {
                  graficoBarrasHelper.dibujarBarras(g);
                  fromRefresh = false;
            }
      }

      // Borrar dibujo actual
      private void reiniciarGrafico() {
            Graphics g = getGraphics();
            Dimension d = getSize();
            Color c = getBackground();
            g.setColor(c);
            g.fillRect(0, 0, d.width, d.height);
            repaint(); // petición para que la MVJ llame a paint()
      }

      private void configurarVentana() {
            setSize(500, 500);
            setLocation(550, 350);
            setVisible(true);
      }
}
```

GraficoBarrasHelper.java

```java
package comportamiento.observer.reports;

import java.awt.Color;
import java.awt.Graphics;
import java.util.StringTokenizer;
import java.util.Vector;

/**
 * Clase se soporte para el observador que se encarga de
 * dibujar el gráfico de barras. Esta clase implementa los
 * detalles de bajo nivel para dibujar el grafico, mientras
 * que el observador correspondiente, se encarga de asuntos
 * más generales.
 */
public class GraficoBarrasHelper {

	private String departamento = "";

	public String getDepartamento() {
		return departamento;
	}

	public void setDepartamento(String departamento) {
		this.departamento = departamento;
	}

	public void dibujarTituloMeses(Graphics g) {
		g.drawString(departamento + " Ventas acumuladas", 150,
50);
		String[] meses = { "Ene", "Feb", "Mar", "Abr", "May",
"Jun", "Jul",
```

```
                        "Ago", "Sep", "Oct", "Nov", "Dic" };
                int x = 50, y = 115;
                for (int j = 0; j < meses.length; j++) {
                        g.setColor(Color.blue);
                        g.drawString(meses[j], x, y);
                        y = y + 30;
                }
        }

        public void dibujarBarras(Graphics g) {
                int x = 100, y = 100;
                int w = 50, h = 20;
                int[] totals = getTotalAnioAcumulado(departamento);

                // Mes actual
                int mes = Utilidades.getMesEnCurso();

                for (int i = 0; i < mes; i++) {
                        g.setColor(Color.blue);
                        if (totals[i] > 0) {
                                w = (int) (totals[i] / 50);
                                g.fillRect(x, y, w, h);
                                g.drawString(totals[i] + "€", x + w + 5, y +
15);
                        }
                        y = y + 30;
                }
        }

        private int[] getTotalAnioAcumulado(String departamento) {
```

```
            int[] totales = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
            for (int i = 0; i < 12; i++) {
                  totales[i] = getTotalVentasMes(i + 1, departamento);
            }
            return totales;
      }

      private int getTotalVentasMes(int mes, String departamento) {
            Vector<String> filas =
      Utilidades.fileToVector(ReportManagerGUI.Fichero);
            int total = 0;
            String textoBusqueda = departamento + "," + mes + ",";

            /*
             * Recorrer todas las filas en busca de aquellas que
coincidan
             * con el mes y departamento especificados en los
parámetros
             */
            for (int i = 0; i < filas.size(); i++) {
                  String fila = filas.get(i);
                  if (match(fila, textoBusqueda)) {
                        StringTokenizer st = new StringTokenizer(fila,
",");
                        st.nextToken();// ignorar departmento
                        st.nextToken();// ignorar el mes
                        st.nextToken();// ignorar fecha
                        st.nextToken();// ignorar items
                        String importe = st.nextToken();
                        total = total + new
Integer(importe).intValue();
```

```
                }
            }
            return total;
      }

      private boolean match(String fila, String textoBusqueda) {
            return fila.indexOf(textoBusqueda) > -1;
      }
}
```

Utilidades.java

```
package comportamiento.observer.reports;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URISyntaxException;
import java.net.URL;
import java.util.Calendar;
import java.util.Date;
import java.util.Vector;

public class Utilidades {

      public static int getMesEnCurso() {
            Calendar cal = Calendar.getInstance();
```

```
            cal.setTime(new Date());
            return cal.get(Calendar.MONTH) + 1;
      }

      /*
       * Lee datos de un fichero y los devuelve en un vector de String
       */
      public static Vector<String> fileToVector(String fileName) {
            URL url = Utilidades.class.getResource(
                        "/comportamiento/observer/reports/" +
fileName);

            Vector<String> v = new Vector<String>();
            String inputLine;
            try {

                  File inFile = new File(url.toURI().getPath());
                  BufferedReader br = new BufferedReader(new
InputStreamReader(
                              new FileInputStream(inFile)));

                  while ((inputLine = br.readLine()) != null) {
                        v.add(inputLine.trim());
                  }
                  br.close();
            } catch (FileNotFoundException ex) {
                  ex.printStackTrace();
            } catch (IOException ex) {
                  ex.printStackTrace();
            } catch (URISyntaxException ex) {
                  ex.printStackTrace();
```

```
            }
            return (v);
      }
}
```

MainClient.java

```
package comportamiento.observer.reports;

import java.awt.EventQueue;

/**
 * Clase cliente que lleva a cabo lo siguiente:
 *
 * - Crea la ventana principal de la aplicación,
 * desde la que el usuario puede seleccionar un
 * departamento del combo, para ver los informes.
 * Esta ventana en un wrapper que envuelve al subject.
 *
 * - Crea los observadores, esto es, un informe
 * de cada tipo.
 */
public class MainClient {

      public static void main(String[] args) {
            EventQueue.invokeLater(new Runnable() {
                  public void run() {
                        try {
                              ReportManagerGUI reportManagerGUI = new
ReportManagerGUI();
```

```
                                    new
InformeMensualTextual(reportManagerGUI);
                                    new
InformeAnualGraficoBarras(reportManagerGUI);
                              } catch (Exception e) {
                                    e.printStackTrace();
                              }
                        }
                  });
            }
}
```

ventas.dat

A continuación se muestra el fichero con las ventas. Por ahorrar espacio se muestra en dos columnas, pero hay que tener en cuenta que para que la aplicación funcione, el fichero sólo debe tener una columna, es decir, cada venta en una única fila.

Consultoria,1,1,2,200

Consultoria,1,10,2,400

Consultoria,1,15,2,400

Consultoria,2,1,2,2000

Consultoria,2,10,2,2000

Consultoria,2,15,2,1000

Consultoria,3,1,2,2000

Consultoria,3,10,2,1000

Consultoria,3,15,2,1000

Consultoria,4,1,2,200

Consultoria,4,10,2,400

Consultoria,4,15,2,400

Consultoria,5,1,2,2000

Consultoria,5,1,2,2000

Consultoria,5,1,2,2000

Consultoria,6,10,2,2000

Consultoria,6,15,2,2000

Consultoria,6,1,2,1000

Consultoria,7,10,2,200

Consultoria,7,15,2,700

Consultoria,7,1,2,100

Consultoria,8,10,2,200

Consultoria,8,15,2,200

Consultoria,8,1,2,1600

Consultoria,9,10,2,1200

Consultoria,9,15,2,200

Consultoria,9,1,2,300

Consultoria,10,10,2,1000

Consultoria,10,15,2,2000

Consultoria,10,1,2,1000

Consultoria,11,10,2,200

Consultoria,11,15,2,200

Consultoria,11,1,2,800

Consultoria,12,10,2,2000

Consultoria,12,15,2,2000

Consultoria,12,1,2,2000

Electronica,1,1,2,2000

Electronica,1,10,2,2000

Electronica,1,15,2,1000

Electronica,2,1,2,2000

Electronica,2,10,2,2000

Electronica,2,15,2,2000

Electronica,3,1,2,200

Electronica,3,10,2,1000

Electronica,3,15,2,800

Electronica,4,1,2,200

Electronica,4,10,2,400

Electronica,4,15,2,400

Electronica,5,1,2,200

Electronica,5,1,2,200

Electronica,5,1,2,600

Electronica,6,10,2,2000

Electronica,6,15,2,2000

Electronica,6,1,2,1000

Electronica,7,10,2,200

Electronica,7,15,2,700

```
Electronica,7,1,2,100
Electronica,8,10,2,200
Electronica,8,15,2,200
Electronica,8,1,2,600
Electronica,9,10,2,200
Electronica,9,15,2,700
Electronica,9,1,2,100
Electronica,10,10,2,2000
Electronica,10,15,2,2000
Electronica,10,1,2,2000
Electronica,11,10,2,1000
Electronica,11,15,2,200
Electronica,11,1,2,1800
Electronica,12,10,2,200
Electronica,12,15,2,200
Electronica,12,1,2,600
Electricidad,1,1,2,200
Electricidad,1,10,2,400
Electricidad,1,15,2,400
Electricidad,2,1,2,2000
Electricidad,2,10,2,2000
Electricidad,2,15,2,1000
Electricidad,3,1,2,2000
Electricidad,3,10,2,1000
Electricidad,3,15,2,1000
Electricidad,4,1,2,200
Electricidad,4,10,2,400
Electricidad,4,15,2,400
Electricidad,5,1,2,200
Electricidad,5,1,2,200
Electricidad,5,1,2,600
Electricidad,6,10,2,2000
Electricidad,6,15,2,2000
Electricidad,6,1,2,1000
Electricidad,7,10,2,200
Electricidad,7,15,2,700
Electricidad,7,1,2,100
Electricidad,8,10,2,1200
Electricidad,8,15,2,1200
Electricidad,8,1,2,1600
Electricidad,9,10,2,200
Electricidad,9,15,2,200
Electricidad,9,1,2,600
Electricidad,10,10,2,1000
Electricidad,10,15,2,2000
Electricidad,10,1,2,1000
Electricidad,11,10,2,1000
Electricidad,11,15,2,200
Electricidad,11,1,2,1800
Electricidad,12,10,2,200
Electricidad,12,15,2,200
Electricidad,12,1,2,600
```

Ejemplo 3 – Soporte de Java para el patrón Observer

Java proporciona en el paquete java.util la clase Observable y la interfaz Observer.

Si necesitamos que una determinada clase publique notificaciones de cambio en su estado, tan sólo tenemos que hacer que herede de Observable. Esta clase maneja internamente una lista de observadores y proporciona los siguientes métodos:

- *addObserver(Observer o):* Añade el observador recibido por parámetro a la lista de observadores registrados.
- *clearChanged():* Limpia el flag que indica que el objeto ha cambiado.
- *countObservers():* Devuelve el número de observadores registrados.
- *deleteObserver(Observer o):* Elimina el observador recibido por parámetro de la lista de observadores registrados.
- *deleteObservers():* Elimina de la lista a todos los observadores.
- *hasChanged():* Comprueba si el estado del objeto ha cambiado.
- *notifyObservers():* Si el objeto ha cambiado (implícitamente comprueba que hasChanged() retorna true) notifica a todos los observadores de la lista (después, implícitamente llama a clearChanged() para indicar que no hay cambios pendientes). Como veremos a continuación en la interfaz Observer, cada observador recibe una notificación mediante su método update(). Cuando notifyObservers() se ejecuta, implícitamente le pasa a update() una referencia al objeto Observable, es decir, le pasa la referencia 'this'. De esta manera, los observadores pueden sincronizar su estado con el del Observable.
- *notifyObservers(Object arg):* Igual que el método anterior, pero en este caso, además de la referencia 'this' también se le pasa la referencia 'arg'.
- *setChanged():* Este método nos permite decidir cuando consideramos que el Observable ha cambiado. Por tanto, el método hasChanged() retornará ahora true.

Obviamente, al hacer que un objeto extienda de la clase Observable, conseguimos que disponga gratis de todos estos métodos.

Por otro lado, cuando queremos que uno o varios objetos reciban inmediatamente notificaciones en el cambio del estado de algún objeto Observable, tenemos que hacer que tales objetos implementen la interfaz Observer, la cual sólo define un método que tendremos que implementar individualmente en cada objeto Observer. La signatura es como sigue:

void update(Observable o, Object arg)

El primer parámetro es la referencia al objeto Observable. El segundo puede ser null o no, dependiendo la sobrecarga del método notifyObservers() que hayamos utilizado en nuestra clase que hereda de Observable (ver la página anterior para ampliar la explicación).

Una vez explicado lo anterior, pasemos a ver el ejemplo.

Tenemos una clase que representa un artículo, formada por un nombre y un precio. Queremos que cuando cambie el nombre o el precio del artículo sean informados sendos objetos observadores. Por tanto, podemos hacer lo siguiente:

- Crear un clase para el artículo. La podemos llamar ConcreteSubject y heredará de java.util.Observable.
- Crear una clase que represente un observador de cambios en el nombre del artículo. La llamaremos NombreObserver e implementará la interfaz java.util.Observer.
- Crear una clase que represente un observador de cambios en el precio del artículo. La llamaremos PrecioObserver e implementará la interfaz java.util.Observer.
- Crear una clase cliente que cree una instancia de cada una de las tres clases anteriores y ponga a prueba el patrón.

Comenzamos por el artículo. Lo importante aquí es, por un lado, ver cómo se ha tomado la decisión de que el objeto ha cambiado en los métodos setNombre() y setPrecio(). Por lo tanto, es en ellos donde se invoca conjuntamente a setChanged() y a notifyObservers(). Por otro lado, hay que advertir lo simple que queda la clase al haber heredado de java.util.Observable toda su funcionalidad.

ConcreteSubject.java

```
package comportamiento.observer.java_support;

import java.util.*;

/**
 * Un sujeto al que observar
 */
public class ConcreteSubject extends Observable {

  private String nombre;
  private float precio;

  public ConcreteSubject(String nombre, float precio) {
    this.nombre = nombre;
    this.precio = precio;
    System.out.println("ConcreteSubject creado: el nombre es " +
            nombre + " y el precio es " + precio);
  }

  public String getNombre() {
    return nombre;
  }

  public float getPrecio() {
    return precio;
  }

  public void setNombre(String nombre) {
```

```
    this.nombre = nombre;
    setChanged();
    notifyObservers(nombre);
  }

  public void setPrecio(float precio) {
    this.precio = precio;
    setChanged();
    notifyObservers(new Float(precio));
  }

}
```

Ahora veamos los observadores, que son realmente muy básicos, además de muy parecidos. Lo único a destacar es la implementación del método update().

NombreObserver.java

```
package comportamiento.observer.java_support;

import java.util.*;

/**
 * Un observador de cambios en el atributo 'nombre' del
ConcreteSubject
 */
public class NombreObserver implements Observer {

  private String nombre;

  public NombreObserver() {
    nombre = null;
```

```
    System.out.println("NombreObserver creado: el nombre es " +
nombre);
  }


  @Override
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      nombre = (String) arg;
      System.out.println("NombreObserver: nombre ha cambiado a " +
nombre);
    }
    else {
      // irrelevante para este observador
    }
  }
}
```

PrecioObserver.java

```
package comportamiento.observer.java_support;


import java.util.*;


/**
 * Un observador de cambios en el atributo 'precio' del
ConcreteSubject
 */
public class PrecioObserver implements Observer {


  private float precio;
```

```
  public PrecioObserver() {
    precio = 0;
    System.out.println("PrecioObserver creado: el precio es " +
precio);
  }

  @Override
  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      precio = ((Float)arg).floatValue();
      System.out.println("PrecioObserver: precio ha cambiado a " +
precio);
    }
    else {
      // irrelevante para este observador
    }
  }
}
```

Finalmente, veamos la clase cliente:

MainClient.java

```
package comportamiento.observer.java_support;

import java.util.Observer;

public class MainClient {

  public static void main(String args[]) {

    // Crear el sujeto y los observadores.
```

```
    ConcreteSubject subject = new ConcreteSubject("Caja de galletas",
1.29F);
    Observer nombreObs = new NombreObserver();
    Observer precioObs = new PrecioObserver();

    // Añadir los observadores a la lista de observadores del sujeto
    subject.addObserver(nombreObs);
    subject.addObserver(precioObs);

    // Hacer algunos cambios en el estado del sujeto
    // para ver cómo se comportan los observadores
    subject.setNombre("Bollos rellenos de chocolate");
    subject.setPrecio(2.57F);
    subject.setPrecio(1.99F);
    subject.setNombre("Tarta de queso");
  }

}
```

Problemas específicos e implementación

Cómo usar el soporte de Java cuando ConcreteSubject ya hereda de una clase

En Java no podemos heredar de dos clases diferentes, por lo que en este caso, la solución pasa por utilizar Delegación.

Así, siguiendo con el último ejemplo, podríamos hacer lo siguiente:

- Crear una subclase de java.util.Observable, esto es, el delegado. Le llamaremos DelegatedObservable.

- Declarar en ConcreteSubject un atributo de tipo DelegatedObservable y pasarle a él las llamadas que anteriormente hacíamos directamente sobre el ConcreteSubject.

Veamos el código (leed detenidamente los comentarios):

DelegatedObservable.java

```
package comportamiento.observer.java_support_delegacion;


import java.util.Observable;


/**
 * Una subclase de Observable que permite delegacion.
 *
 * No podemos utilizar directamente en ConcreteSubject un atributo de
tipo
 * java.util.Observable, ya que los métodos clearChanged() y
setChanged() no
 * tienen visibilidad pública en Observable.
 */
public class DelegatedObservable extends Observable {


	public void clearChanged() {
		super.clearChanged();
	}


	public void setChanged() {
		super.setChanged();
	}


}
```

ConcreteSubject.java

```
package comportamiento.observer.java_support_delegacion;


import java.util.*;


/**
 * Un sujeto al que observar. La diferencia con la anterior versión es que esta
 * clase ya extiende de otra, por lo que no podemos heredar de
 * java.util.Observable. La solución pasa por delegar, esto es, declarar un
 * atributo de tipo DelegatedObservable y pasarle a él las llamadas.
 */


class ParentClass {
}


public class ConcreteSubject extends ParentClass {


	private DelegatedObservable observable;


	private String nombre;
	private float precio;


	public ConcreteSubject(String nombre, float precio) {
		this.nombre = nombre;
		this.precio = precio;
		observable = new DelegatedObservable();
		System.out.println("ConcreteSubject creado: el nombre es "
+ nombre
				+ " y el precio es " + precio);
```

```
	}

	public String getNombre() {
		return nombre;
	}

	public float getPrecio() {
		return precio;
	}

	public void addObserver(Observer o) {
		observable.addObserver(o);
	}

	public void deleteObserver(Observer o) {
		observable.deleteObserver(o);
	}

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

	public void setPrecio(float precio) {
		this.precio = precio;
		observable.setChanged();
		observable.notifyObservers(new Float(precio));
	}
```

```
}
```

Observar a más de un sujeto

Cuando un observador debe observar a más de un sujeto ¿cómo saber quién ha notificado un cambio?

En este caso, se soluciona pasando al método update() una referencia al sujeto. Tanto el método notifyObservers() de java.util.Observable (esto sucede implícitamente) como el método update() de la interfaz java.util.Observer ya se contempla satisfactoriamente esta situación.

¿Quién lanza la notificación de actualización (la operación notifyObservers() en el sujeto)?

Tanto el sujeto como sus observadores confían en el mecanismo de notificaciones para mantener su estado consistente. Pero realmente ¿qué objeto llama a notifyObservers() para que se ejecute el método update()?

Tenemos dos opciones:

- La más común: que sea el propio sujeto el que una vez que advierta un cambio en su estado se encargue de invocar a notifyObservers(). La ventaja de esta aproximación es que ninguna otra clase tiene que acordarse de llamar a notifyObservers(). El inconveniente de esta opción es que cuando se producen muchos cambios seguidos de estado en sujeto se producen consecuentemente muchas llamadas a update() en los observadores, lo cual puede ser ineficiente.
- Que sean los propios observadores, o cualquier otra clase cliente, quienes llamen a notifyObservers() cuando consideren que es apropiado hacerlo. La ventaja de esta aproximación es que se pueden realizar una serie de cambios consecutivos en sujeto y sólo después del último cambio invocar a notifyObservers(), lo cual ahorra inútiles llamadas a

update(). El gran inconveniente es que es fácil olvidar llamar a notifyObservers() desde una clase cliente.

Referencias perdidas de sujetos eliminados

La eliminación de un sujeto no debe causar referencias colgadas en sus observadores. Una manera de evitar esto es hacer que el sujeto que va a ser eliminado notifique tal hecho a sus observadores, para que éstos eliminen la referencia al mismo.

Garantizar que el estado del sujeto es consistente antes de proceder a la notificación

Si se llama a notifyObservers() antes de que se haya actualizado completamente el estado del sujeto, se tiene que los observadores se habrán actualizado con un estado incompleto o antiguo.

Las situaciones más propensas a este tipo de errores se dan cuando tenemos una jerarquía de clases para el sujeto. Por ejemplo, en el siguiente fragmento de código tenemos un caso en que los observadores son notificados antes de que se haya actualizado completamente el estado del sujeto:

```
abstract class Observable {
      ...
      int state = 0;
      int additionalState = 0;
      public updateState(int increment) {
            state = state + increment;
            notifyObservers();
      }
      ...
```

```
}


class ConcreteObservable extends Observable {
	...
	public updateState(int increment) {
		// los observadores son notificados
		super.updateState(increment);
		// el estado del sujeto ha cambiado pero los observadores ya se han actualizado
		additionalState = additionalState + increment;
	}
	...
}
```

El error anterior, puede solucionarse aplicando el patrón Template Method (aunque aún no lo hemos estudiado). En el siguiente fragmento de código se muestra un ejemplo:

```
abstract class Observable {
	...
	int state = 0;
	int additionalState = 0;
	public void final updateState(int increment) {
		doUpdateState(increment); // Ejecutar el de la subclase
		// los observadores son notificados una vez que todo el estado de la jerarquía está actualizado
		notifyObservers();
	}
	public void doUpdateState(int increment) {
		state = state + increment;
	}
	...
```

```
}

class ConcreteObservable extends Observable {
      ...
      public doUpdateState(int increment){
            super.doUpdateState(increment);
            additionalState = additionalState + increment;
      }
      ...
}
```

Comunicación tipo push o tipo pull

Hay dos maneras de pasar datos entre el sujeto y sus observadores cuando el estado del sujeto cambia.

- Modelo push: El sujeto envía a los observadores información detallada sobre los cambios producidos en su estado, tanto si es relevante para los observadores como si no. Este modelo podría hacer a los observadores poco reutilizables, ya que el sujeto necesita conocer las necesidades de información de sus observadores.
- Modelo pull: El sujeto envía a los observadores la mínima información posible sobre los cambios producidos en su estado. A continuación, los observadores preguntan al sujeto por lo que a ellos les sea de interés. Este modelo enfatiza en la ignorancia del sujeto sobre sus observadores, lo que hace a los observadores más reutilizables. Sin embargo, este modelo podría ser ineficiente, ya que los observadores deben saber qué cambió en concreto sin la ayuda del sujeto.

Suscripción a una única categoría de cambios

Es posible mejorar la eficiencia en el sistema de notificaciones si evitamos el tener que llamar siempre al método update() de todos los observadores, cuando no a todos les interesa el tipo de cambio que ha sucedido en el sujeto.

La solución pasa por hacer que el método de registro de observadores en el sujeto, incorpore un parámetro que indique el tipo de evento en que cada observador está interesado.

Veamos el código completo que demuestra este caso. A tener en cuenta:

- No utilizaremos el soporte de Java, ya que nos nos sirve para lo que queremos hacer.
- Utilizaremos como base un ejemplo ya visto: un sujeto que notifica cambios en sus atributos “descripcion” y “precio”. Sin embargo, a diferencia del ejemplo visto anteriormente, ahora tendremos un tercer observador: FullObserver, el cual estará interesado tanto en cambios en “descripcion” como en “precio”.

Como se ha sugerido, necesitamos definir un tipo PropertyEvent. Este tipo no es más que una asociación 1-1 entre un atributo del sujeto y una subclase de PropertyEvent. Por ejemplo, si un observador está interesado en las modificaciones en el precio del sujeto, entonces, cuando se registre como listener del sujeto indicará que lo quiere hacer en calidad de listener de eventos PropertyPrecioEvent.

Por tanto, creamos un paquete diferenciado del resto de clases que crearemos para crear dentro toda la jerarquía de PropertyEvent:

Comenzamos por la interfaz PropertyEvent, la cual declara tres métodos que debe implementar cualquier clase que implemente esta interfaz:

- getNombre(): Devuelve el nombre del atributo que ha cambiado en el sujeto.

- getValue(): Devuelve el valor -como un Object- del atributo que ha cambiado en el sujeto.
- getClassValor(): Devuelve el nombre de la clase -como un objeto Class- del atributo que ha cambiado. Esto significa que tendremos que ir con cuidado si usamos tipos primitivos, ya que podemos tener problemas con el auto-boxing y el unboxing. Por otro lado, gracias al método getClassValor() un listener que esté interesado en más de un atributo del sujeto, puede evaluar en su método de actualización el tipo del atributo recibido y hacer a continuación el cast correspondiente.

PropertyEvent.java

```
package comportamiento.observer.improve.event;


public interface PropertyEvent {
	public String getNombre();
	public String getValor();
	public Class<?> getClassValor();
}
```

A continuación, definimos una clase abstracta, AbstractPropertyEvent, que implementa la interfaz PropertyEvent. Esta clase define los tres atributos de interés para un listener acerca de un PropertyEvent: el nombre del atributo que ha cambiado, su valor y la clase a la que pertenece.

AbstractPropertyEvent.java

```
package comportamiento.observer.improve.event;


public abstract class AbstractPropertyEvent implements PropertyEvent {
```

```
	private String nombre;

	private Object valor;

	private Class<?> claseDelValor;


	protected AbstractPropertyEvent(String nombre, Object valor,
Class<?> claseDelValor) {
		this.nombre = nombre;

		this.valor = valor;

		this.claseDelValor = claseDelValor;

	}


	@Override

	public String getNombre() { return nombre; }


	@Override

	public Object getValor() {    return valor; }


	@Override

	public Class<?> getClassValor() { return claseDelValor; }


}
```

Ahora pasemos a ver las subclases de AbstractPropertyEvent. En nuestro caso, utilizamos tres: una para listeners interesados en cambios en el precio, otra para interesados en cambios en la descripción y la tercera, para interesados en cambios en los dos atributos de sujeto. Hay que tener en cuenta que en nuestro ejemplo, las tres subclase son idénticas, esto es, no aportan nada, y lo único interesante es que las podemos diferenciar por su tipo. Lo esperado en una implementación profesional, sería que aportaran comportamiento o estado diferenciado respecto a la clase base.

PropertyPrecioEvent.java

```
package comportamiento.observer.improve.event;


public class PropertyPrecioEvent extends AbstractPropertyEvent {


	public PropertyPrecioEvent(String nombre, Object valor, Class<?>
claseDelValor) {

		super(nombre, valor, claseDelValor);

	}

}
```

PropertyDescripcionEvent.java

```
package comportamiento.observer.improve.event;


public class PropertyDescripcionEvent extends AbstractPropertyEvent {


	public PropertyDescripcionEvent(String nombre, Object valor,
Class<?> claseDelValor) {

		super(nombre, valor, claseDelValor);

	}

}
```

AllPropertiesEvent.java

```
package comportamiento.observer.improve.event;


public class AllPropertiesEvent extends AbstractPropertyEvent {


	public AllPropertiesEvent(String nombre, Object valor, Class<?>
claseDelValor) {
```

```
            super(nombre, valor, claseDelValor);
      }
}
```

Con esto ya hemos acabado las clases para los eventos. Ahora, en otro paquete diferente al anterior, creamos el resto de clases del ejemplo.

Comenzamos por la interfaz que deben implementar los observadores. A tener en cuenta:

- Arbitrariamente, vamos a utilizar la nomenclatura que ya utilizamos en el primer ejemplo de este documento.
- La interfaz declara un único método, el método de actualización, el cual ahora contiene un único argumento: un objeto PropertyEvent. De esta manera, cada observador podrá obtener de los getters() del objeto toda la información sobre el atributo que ha cambiado en el sujeto.

PropertyListener.java

```
package comportamiento.observer.improve;

import comportamiento.observer.improve.event.PropertyEvent;

public interface PropertyListener {
      public void onPropertyEvent(PropertyEvent event);
}
```

A continuación veamos el código para los observadores. Comenzamos con el interesado en cambios en el precio del sujeto. Lo interesante es advertir que tiene que hacer el downcast de Object a Float, sin ningún miedo, pues un Float es lo que tiene que recibir.

PrecioObserver.java

```
package comportamiento.observer.improve;

import comportamiento.observer.improve.event.PropertyEvent;

public class PrecioObserver implements PropertyListener {

	private Float precio;

	@Override
	public void onPropertyEvent(PropertyEvent event) {
		System.out.println(this.getClass());
		System.out.println("Atributo: " + event.getNombre());
		precio = (Float) event.getValor();
		System.out.println("Valor: " + precio.floatValue());
		System.out.println();
	}
}
```

Seguimos ahora con el interesado en cambios en la descripción del sujeto. A diferencia del observador anterior, este tiene que hacer el downcast de Object a String:

DescripciónObserver.java

```
package comportamiento.observer.improve;

import comportamiento.observer.improve.event.PropertyEvent;

public class DescripcionObserver implements PropertyListener {
```

```
	private String descripcion;

	@Override
	public void onPropertyEvent(PropertyEvent event) {
		System.out.println(this.getClass());
		System.out.println("Atributo: " + event.getNombre());
		descripcion = (String) event.getValor();
		System.out.println("Valor: " + descripcion);
		System.out.println();
	}
}
```

Finalmente, veamos el código para el observador que está interesado en ambos atributos. Es interesante notar cómo hacer uso del método getClassValor() del evento recibido para diferenciar si está siendo invocado a causa de un cambio en “precio” o en “descripcion”.

FullObserver.java

```
package comportamiento.observer.improve;

import comportamiento.observer.improve.event.PropertyEvent;

public class FullObserver implements PropertyListener {

	private Float precio;
	private String descripcion;

	@Override
	public void onPropertyEvent(PropertyEvent event) {
```

```
            System.out.println(this.getClass());

            System.out.println("Atributo: " + event.getNombre());

            Class<?> clase = event.getClassValor();
            if (clase.equals(Float.class)) {
                  precio = (Float) event.getValor();
                  System.out.println("Valor: " + precio.floatValue());
            } else if (clase.equals(String.class)) {
                  descripcion = (String) event.getValor();
                  System.out.println("Valor: " + descripcion);
            }

            System.out.println();
      }
}
```

Ahora toca el turno del sujeto. Comenzamos por ver la interfaz que debe implementar. Notad que:

- El método addPropertyListener() recibe como primer argumento un observador y como segundo cualquier tipo Class que implemente la interfaz PropertyEvent. Con esto nos aseguramos de que podremos almacenar en el Map del sujeto, para cualquier observador, su correspondiente evento de interés.
- El método removePropertyListener() únicamente recibe como argumento la referencia al observador que se tiene que desregistrar. De hecho, en el programa de ejemplo mostrado, ni si quiera se utiliza este método.

- El método PublishPropertyEvent() recibe como argumento el PropertyEvent que se ha producido. Así, por ejemplo, si se produce un cambio en el precio del sujeto, este método contendrá un PropertyPrecioEvent. Como veremos a continuación, la lógica implementada en el sujeto se encargará de llamar sólo a los observadores interesados en este tipo de eventos.

PropertyPublisher.java

```
package comportamiento.observer.improve;


import comportamiento.observer.improve.event.PropertyEvent;


public interface PropertyPublisher {

	public void addPropertyListener(PropertyListener listener,
Class<? extends PropertyEvent> clase);

	public void removePropertyListener(PropertyListener listener);

	public void publishPropertyEvent(PropertyEvent event);

}
```

Pasemos a ver el sujeto. Notad que:

- Define un Map para guardar a los diferentes observadores y para cada uno, el tipo de evento sobre el que quieren recibir notificaciones de cambio.
- La parte más interesante es la relacionada con el cambio en su estado. Por ejemplo, cuando se invoca su método setDescripcion(), el cuerpo del método hace lo siguiente:
    - Se guarda en el atributo de instancia el nuevo valor para la “descripcion”.
    - Se crea un PropertyEvent del tipo PropertyDescripcionEvent.

    - Se llama a publishPropertyEvent() pasándo como parámetro el nuevo objeto PropertyDescripcionEvent.

- Al ejecutarse publishPropertyEvent(), sucede lo siguiente en el cuerpo del método:
- Se recorre cada entrada del mapa de observadores.
- Para cada uno, se comprueba si está interesado en el evento en curso, en cuyo caso se invoca al método onPropertyEvent() de tal observador, pasándole el evento en curso. Por otro lado, también dentro de la iteración, siempre se comprueba si el evento en curso es del tipo AllPropertiesEvent, ya que en tal caso debe ser notificado, dado que a este tipo de observador le interesan todas las notificaciones de cambio.

Subject.java

```
package comportamiento.observer.improve;

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

import comportamiento.observer.improve.event.AllPropertiesEvent;
import comportamiento.observer.improve.event.PropertyDescripcionEvent;
import comportamiento.observer.improve.event.PropertyEvent;
import comportamiento.observer.improve.event.PropertyPrecioEvent;

public class Subject implements PropertyPublisher {

	String descripcion;
	Float precio;
```

```
	private Map<PropertyListener, Class<? extends PropertyEvent>>
registerListeners;


	public Subject(String descripcion, float precio) {
		registerListeners =
				new HashMap<PropertyListener, Class<? extends
PropertyEvent>>();
		this.descripcion = descripcion;
		this.precio = precio;
		System.out.println(this.getClass());
		System.out.println("Descripcion: " + descripcion);
		System.out.println("Precio: " + precio);
		System.out.println();
	}


	@Override
	public void addPropertyListener(PropertyListener listener,
			Class<? extends PropertyEvent> clase) {


		registerListeners.put(listener, clase);
	}


	@Override
	public void removePropertyListener(PropertyListener listener) {
		registerListeners.remove(listener);
	}


	@Override
	public void publishPropertyEvent(PropertyEvent event) {
		for (Entry<PropertyListener, Class<? extends
PropertyEvent>> listener :
```

```
                  registerListeners.entrySet())
            {
                  Class<? extends PropertyEvent> currentEventClass =
listener.getValue();
                  if (currentEventClass.equals(event.getClass())) {
                        listener.getKey().onPropertyEvent(event);
                  }
                  if
(currentEventClass.equals(AllPropertiesEvent.class)) {
                        listener.getKey().onPropertyEvent(event);
                  }
            }
      }

      public String getDescripcion() {
            return descripcion;
      }

      public void setDescripcion(String descripcion) {
            this.descripcion = descripcion;
            PropertyEvent event =
                        new PropertyDescripcionEvent("descripcion",
descripcion, String.class);
            publishPropertyEvent(event);
      }

      public Float getPrecio() {
            return precio;
      }

      public void setPrecio(Float precio) {
```

```
            this.precio = precio;
            PropertyEvent event =
                        new PropertyPrecioEvent("precio", precio,
Float.class);
            publishPropertyEvent(event);
      }

}
```

Finalmente, veamos el código para una clase cliente. La clase comienza creando un sujeto, el cual en su constructor mostrará por pantalla su estado inicial. A continuación se crea un observador interesado en el precio y se añade a la lista de listeners del sujeto. Notad en este punto como el segundo parámetro en la llamada al método de registro es un tipo Class PropertyPrecioEvent, dejando así claro su interés por este tipo de eventos. Luego, pasa algo análogo para el resto de los observadores. Por último, se modifica el estado del sujeto, con lo que se consigue ver por pantalla como cada observador recibe tales notificaciones de cambio:

MainClient.java

```
package comportamiento.observer.improve;

import comportamiento.observer.improve.event.AllPropertiesEvent;
import comportamiento.observer.improve.event.PropertyDescripcionEvent;
import comportamiento.observer.improve.event.PropertyPrecioEvent;

public class MainClient {

      public static void main(String[] args) {
```

```
            Subject subject = new Subject("Caja roja", 5.50F);


            PropertyListener listenerPrecio = new PrecioObserver();


            subject.addPropertyListener(listenerPrecio,
PropertyPrecioEvent.class);


            PropertyListener listenerDescripcion = new
DescripcionObserver();

            subject.addPropertyListener(listenerDescripcion,
PropertyDescripcionEvent.class);


            PropertyListener fullListener = new FullObserver();

            subject.addPropertyListener(fullListener,
AllPropertiesEvent.class);


            subject.setDescripcion("Caja negra");

            subject.setPrecio(6.00F);

      }

}
```

Salida:

```
class comportamiento.observer.improve.Subject
Descripcion: Caja roja
Precio: 5.5

class comportamiento.observer.improve.FullObserver
Atributo: descripcion
Valor: Caja negra

class comportamiento.observer.improve.DescripcionObserver
Atributo: descripcion
Valor: Caja negra

class comportamiento.observer.improve.FullObserver
Atributo: precio
Valor: 6.0

class comportamiento.observer.improve.PrecioObserver
Atributo: precio
Valor: 6.0
```

Cuando la semántica de actualizaciones es compleja

Cuando tenemos varios sujetos, la relación de dependencia entre sujetos y observadores se vuelve mucho más compleja. En esta situación se puede usar un objeto intermediario, ChangeManager, para la gestión de cambios. De esta manera se minimiza el trabajo de reflejar en los observadores los cambios producidos en los sujetos. Por ejemplo, si se actualizan varios sujetos interdependientes, hay que asegurar que los observadores se actualizan sólo después de que se hayan actualizado todos los sujetos, ya que en otro caso estaremos ineficientemente llamando a los métodos de actualización más veces de las necesarias. Supongamos lo siguiente:

SujetoA tiene el atributo a / SujetoB tiene el atributo b

Se produce una operación que implica la siguiente secuencia (que sólo tiene sentido como una operación atómica): *a->b->a*

Es decir, hay un cambio en 'a' que conlleva un cambio en 'b' que a su vez implica un nuevo cambio en 'a'.

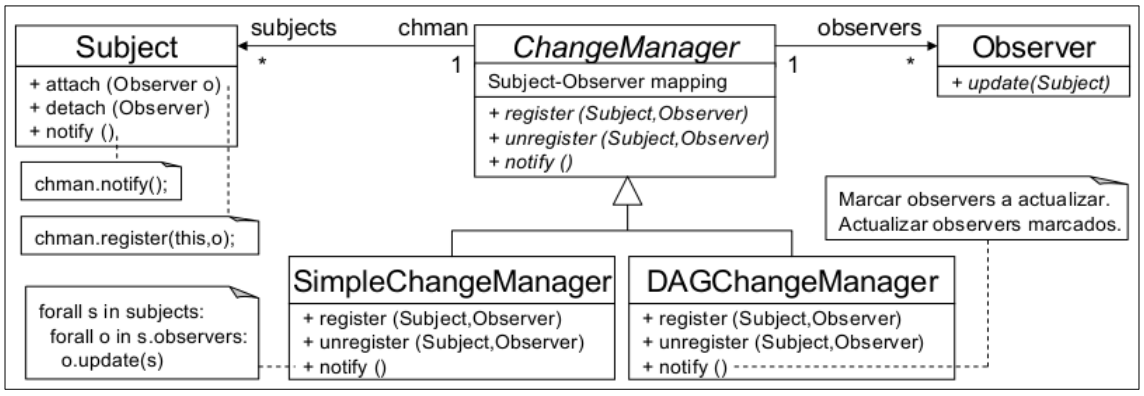
En un situación como la anterior, tendríamos que evitar que el observador de 'a' fuese invocado dos veces, ya que sólo es importante el último cambio.

El objeto intermediario, ChangeManager, tiene tres responsabilidades:

- Llevar un registro de la relación entre sujetos y observadores y proporcionar una interfaz para mantener esta correspondencia. Esto elimina la necesidad de que sean los propios sujetos los encargados de llevar un control de sus observadores y viceversa.
- Define las estrategias de actualización.
- A petición de los sujetos, actualiza los observadores según sea necesario.

El siguiente diagrama muestra un escenario en el que se utiliza un ChangeManager, el cual dispone de dos clases de implementación:

- SimpleChangeManager: Implementación sencilla que actualiza todos los observadores de todos los sujetos.
- DAGChangeManager: Implementación sofisticada que es capaz de actualizar sólo aquellos observadores que deben ser actualizados.



Hay que advertir que ChangeManager es una instancia del patrón Mediator. Por lo general, sólo tendremos una instancia de ChangeManager, por lo que lo podemos implementar con el patrón Singleton.

Patrones relacionados

- **Template Method**: Como se ha visto en el subapartado 'garantizar estado consistente antes de notificar', si los sujetos presentan jerarquía de herencia, se puede utilizar el patrón Template Method para asegurar que no se llama a notifyObservers() hasta que el estado del sujeto es totalmente consistente.
- **Mediator**: Como se ha comentado en el subapartado 'semántica de actualizaciones compleja', tal complejidad puede encapsularse en una clase ChangeManager, la cual actúa como un mediador entre sujetos y observadores.
- **Singleton**: También en relación a ChangeManager, esta clase puede diseñarse como un Singleton para garantizar una única instancia globalmente accesible.