Introducción

En este laboratorio se realizará una introducción a la herramienta para el desarrollo de patrones de diseño PatternBox y se realizarán varios ejercicios utilizando los patrones Strategy, Factory Method, Observer y Adapter.

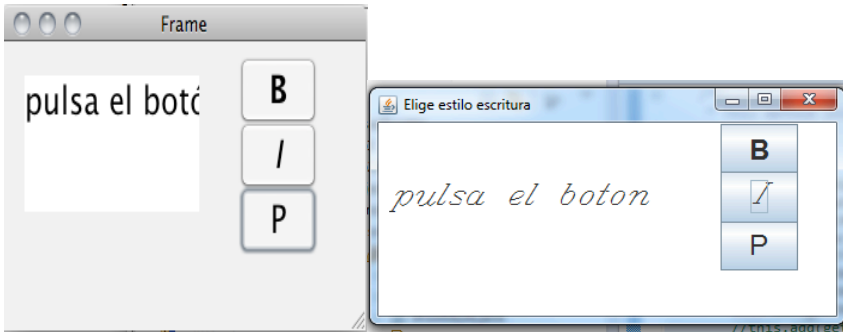
NOTA MUY IMPORTANTE: La herramienta patternBox ha sido probada[1] en Eclipse KEPLER sobre Windows 10. Recomiendo descargar este versión de eclipse a la hora de realizar este laboratorio.

Instalación del plug-in PatternBox

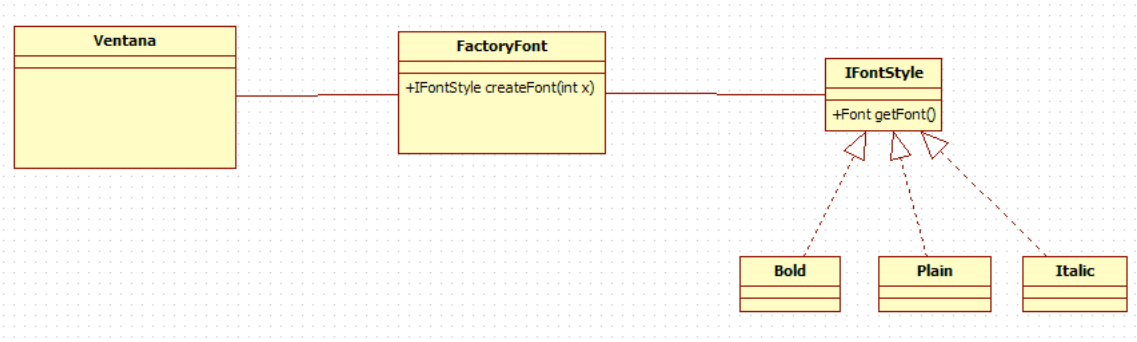
Desde eclipse, ir a "Eclipse Marketplace" buscar "patternBox" e instalar el plug-in.

Primer ejemplo

En la siguiente figura aparece la ventana de la aplicación a desarrollar, donde el formato del texto que aparece a la izquierda viene determinado por el botón de la derecha que se ha seleccionado.



El diagrama de clases de esta aplicación lo podemos ver en la siguiente figura:



Donde podemos identificar 2 patrones:

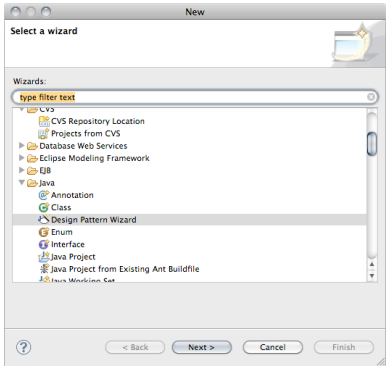
[1] Probado el 9 de octubre de 2017.

1. Patrón Strategy, donde la estrategia abstracta viene determina por la interfaz IFontStyle que devuelve un tipo de fuente a través del método getFont(), y donde cada estrategia concreta devuelve un tipo de Font concreto.
2. Patrón Factoría simple, donde a través del método createFont obtenemos un tipo de fuente (IFontStyle) dependiendo del valor del parámetro (p.ej el valor x=1 nos devuelve un IFontStyle de tipo Bold).

Para realizar esta aplicación crear un proyecto en Java y realizar los siguientes pasos:

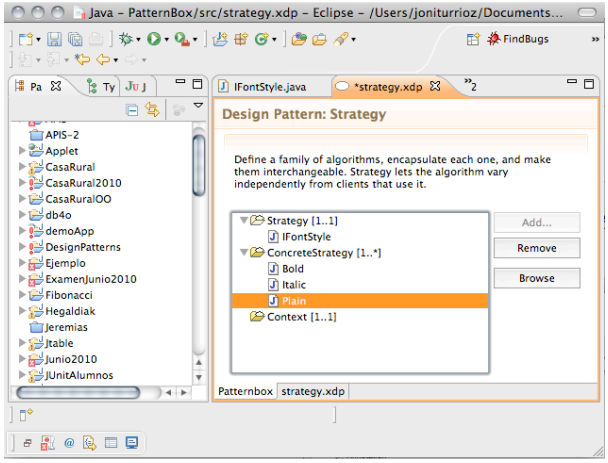
Paso 1: Crear la estrategia de la aplicación.

En el menú de Eclipse, crear un nuevo objeto de tipo "Design pattern wizard" tal y como se muestra en la siguiente pantalla:



Y en la siguiente pantalla, dentro de los "Behavioral patterns" seleccionar el patrón Strategy.

A continuación indicar cuál es la clase interfaz Strategy, y las clases estrategia concreta tal y como se muestra a continuación:



En el siguiente paso ir al código de la interfaz y **refactorizar** el método que aparece a la siguiente signatura:

```
public Font getFont() {
```

utilizando las refactorizaciones "Rename" y "Change Method Signature".
Finalmente implementar el método getFont en las clases concretas:

```
public class Bold implements IFontStyle{

        public Font getFont() {
            return new Font("Bold", Font.BOLD, 24);
        }
}
```

Paso 2: Crear la Factoría.

Vamos a crear una Factoría simple, tal que, a partir de un número nos devuelva un tipo de IFontStyle diferente. El código de esta clase es el siguiente:

```
public class FactoryFont{
        public static IFontStyle createFont(int n){
                if (n==1) return new Bold();
                if (n==2) return new Italic();
                if (n==3) return new Plain();
                return null;
        }
}
```

donde podemos observar que el método createFont es un método estático de la clase.

Paso 3. Crear la clase ventana.

Vamos a crear una Ventana de tipo Frame donde vamos a introducir los 3 botones y el campo de texto que aparecen en la primera página. Para realizar esta tarea instalar desde el Market place el plug-in WindowBuilder y desde el diseñador añadir los elementos[2].
Lo interesante de esta clase es cómo se obtienen las fuentes de estilo de los botones, y cómo lo aplican cuando se hace *click*. En el siguiente código tenéis el "listener" asociado al primer botón (es decir, la acción que se ejecuta, cuando se hace *click* en el botón)

```
jButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
        IFontStyle f= FactoryFonts.createFont(1);
        jTextArea.setFont(f.getFont());             }
 });
```

Como podéis observar, inicialmente se crea un objeto de tipo IFontStyle de tipo 1 invocando la factoría, y a continuación se indica que se le aplique al objeto de texto jTextArea, el tipo de fuente recuperada en la anterior instrucción. Tendreis que hacer algo similar en la inicialización de los botones para añadir el formato adecuado.

Ejecutar el programa y comprobar los resultados.

[2] El código de esta clase se adjunta con este laboratorio.

Patrón Observer

Queremos realizar una aplicación que dependiendo del color que elijamos en un desplegable (situado en una ventana), se pinte una ventana tal y como se muestra a continuación:

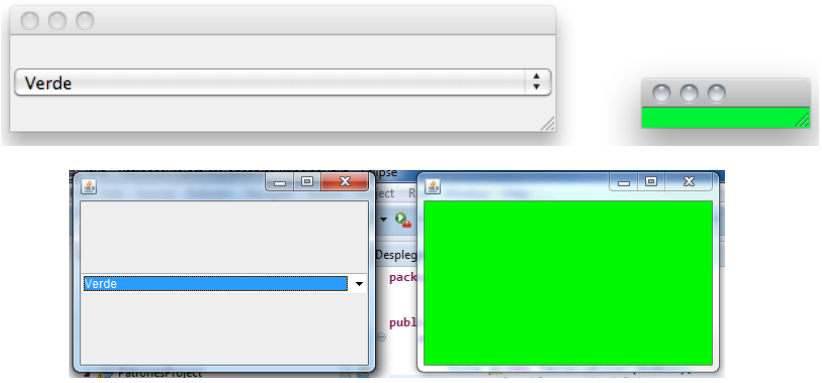
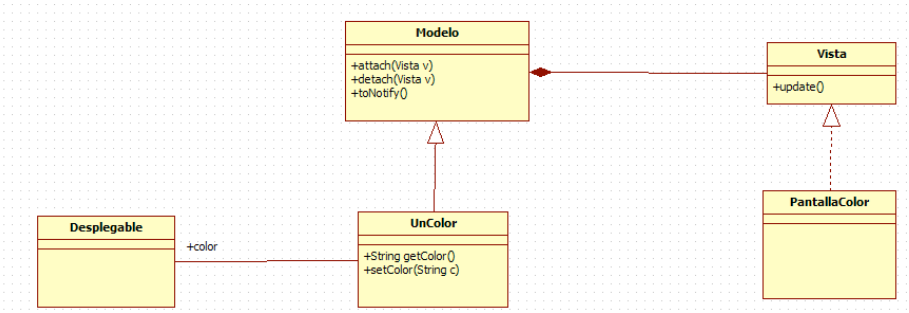


Figura 1.- Ventanas Desplegable y PantallaColor

Para ello vamos a desarrollar una aplicación que sigue un patrón Observer tal y como se muestra en el siguiente diseño:



Tenemos una clase UnColor que representa el Modelo y guarda un color que se puede gestionar a partir de los métodos getColor() y setColor(String c). Además disponemos de una clase Desplegable, que tiene un objeto UnColor que lo actualiza dependiendo del color seleccionado en su Desplegable (ver Figura 1). Finalmente tenemos una clase *Vista* PantallaColor que está subscrita a un objeto *Modelo* de tipo un UnColor, y su estado irá cambiando dependiendo del valor del objeto.

Estos son los pasos a seguir para realizar esta aplicación.

Paso 0: Crear un proyecto y paquete llamado observer
Paso 1: Crear la estructura de clases.

Utilizando la herramienta PatternBox, crear la estructura de clases del patrón Observer. Situarse en cada tipo de objeto característico del patrón y pulsar el botón Add para añadir:

1. Añadir el Observer llamado Vista. ¡OJO! Añadirlo en el paquete "observer" creado previamente.
2. Añadir el Subject llamado Modelo
3. Añadir el State llamado State (en nuestro ejemplo no tiene ningún objeto que lo represente)
4. Añadir el ConcreteSubject llamado UnColor

5. Añadir el Concrete Observer llamado PantallaColor (la que irá cambiando de colores dependiendo de la selección del desplegable)

Observar el resultado en la Figura 2.

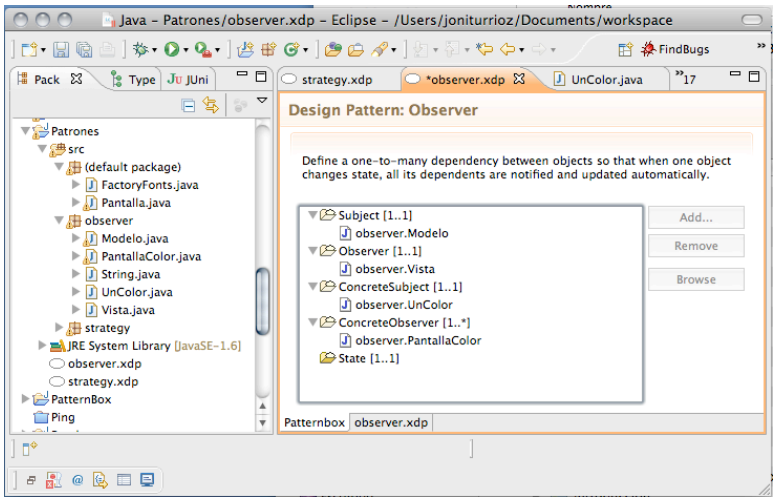


Figura 2. Composición del patrón Observer.

La herramienta patternBox añade una nueva interfaz State al modelo (es un tipo de interfaz especial denominada Marker Interface) que no la vamos a utilizar.

Paso 2: Implementar el modelo

Refactorizar la clase UnColor para que el State fSubjectState sea el atributo color de tipo String, y los métodos getState() y setState sean respectivamente getColor() y setColor(), tal y como se muetra a continuación:

```
public class UnColor extends Modelo {

	/** stores the state of the ConcreteSubject */
	private String color;

	/**
	 * This method returns the state of the ConcreteSubject instance.
	 */
	public String getColor() {
		return color;
	}

	/**
	 * This method sets the state of the ConcreteSubject instance.
	 */
	public void setColor(String state) {
		color = state;
		this.toNotify();
	}

}
```

Como podéis observar cada vez que se invoca al método que cambia el valor del atributo color (es decir, *setColor*), además de cambiar su valor, invoca al método *toNotify()* cuya finalidad es notificar a todos sus observadores (es decir, las vistas) que el modelo ha cambiado.

Paso 3: Crear el objeto Desplegable

Vamos a crear una ventana que visualice un desplegable con varios colores, y actualice un objeto de tipo UnColor, dependiendo del color seleccionado. El código de la clase lo tenéis a continuación:

```
public class DesplegableFrame extends JFrame{
  Choice choice1 = new Choice();
   UnColor color;
   public DesplegableFrame(UnColor c) {
    setSize(300, 200);
    setLocation(10,10);
    color=c;
    choice1 = new Choice();
    choice1.addItem("Rojo");
    choice1.addItem("Blanco");
    choice1.addItem("Verde");
    choice1.addItemListener(new java.awt.event.ItemListener(){
       public void itemStateChanged(ItemEvent e) {
              color.setColor(choice1.getSelectedItem()); }
        });
     add(choice1);
     setVisible(true);
   }
}
```

Como podéis observar esta clase contiene un objeto "color" de tipo UnColor, que se le asigna una referencia a un objeto en el método constructor. Sin embargo el código más interesante lo podéis ver en el "listener" asociado al objeto choice1, que indica que cada vez que se cambia el valor del desplegable (*itemStateChange*), se obtiene el valor seleccionado y se le asigna al objeto "color".

Paso 4: Implementar la vista

La clase que va a visualizar el estado de un objeto de tipo UnColor va a ser la clase PantallaColor. El código que hemos modificado al generado por PatternBox es el siguiente:

```
public class PantallaColor extends Frame implements Vista {
        /** stores the associated ConcreteSubject */
        private final UnColor fConcreteSubject;
 public PantallaColor(UnColor subject) {
        setSize(300, 200);
        setLocation(350,10);
        subject.attach(this);
        setVisible(true);
 }
 public void update() {
        String color = fConcreteSubject.getColor();
        if (color.compareTo("Rojo")==0) setBackground(Color.RED);
        else if (color.compareTo("Blanco")==0) setBackground(Color.WHITE);
        else if (color.compareTo("Verde")==0)  setBackground(Color.GREEN);
        repaint();
 }
}
```

Intercalar este código con el código generado por PatternBox. Únicamente indicamos que la clase extiende de Frame (es decir es un Frame), en el método constructor, **nos subscribimos al modelo** e indicamos que el Frame se haga visible. Finalmente en el método update() actualizamos el color del frame dependiendo del color obtenido del subject al que nos hemos subscrito.

Paso 5: Crear el programa principal

Crear un programa principal, que cree un modelo, crear una vista y pasarle como argumento cuál es su modelo, y finalmente crear un desplegable indicando qué modelo debe actualizar. A continuación tenéis el código[3]:

```
public class Principal {
        public static void main(String args[]){
                UnColor modelo=new UnColor();
                Vista pc=new PantallaColor(modelo);
                new DesplegableFrame(modelo);
        }
}
```

[3] El código de esta clase se adjunta con este laboratorio.

Patrón Adapter

Nuestra clase Sorting tiene un método "sortingSort"[4] que a partir de un objeto Iterator y un objeto Comparator, devuelve otro Iterator, donde los elementos de Iterator inicial están ordenados en base al criterio establecido en el Comparator. Su implementación es la siguiente:

```
public class Sorting {
       public static Iterator sortedIterator(Iterator it, Comparator comparator) {
             List list = new ArrayList();
             while (it.hasNext()) {
                 list.add(it.next());
             }

             Collections.sort(list, comparator);
             return list.iterator();
         }

}
```

Donde Iterator y Comparator son dos interfaces definidas en la librería de java.util. Su definición es la siguiente:

Method Summary

Interface Iterator

Modifier and Type	Method and Description
boolean	**hasNext**() Returns `true` if the iteration has more elements.
E	**next**() Returns the next element in the iteration.

Interface Comparator

Modifier and Type	Method and Description
int	**compare**(**T** o1, **T** o2) Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.
boolean	**equals**(**Object** obj) Indicates whether some other object is "equal to" this comparator.

[4] Método extraído de https://stackoverflow.com/questions/16434526/sort-an-iterator-of-strings. Octubre 2016.