- **EJERCICIO 1: PATRÓN STRATEGY y FACTORY METHOD**
-

1. Crear nuevos formatos para las fuentes 1 y 2, siendo "Roman-Baseline" y "TrueType-Font" respectivamente y volver a ejecutar la aplicación.

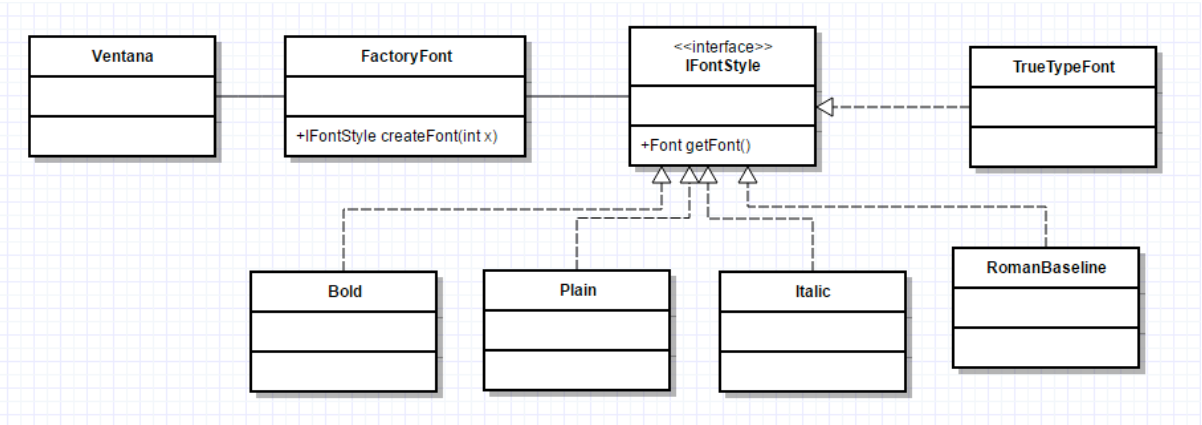
Para cambiar el formato de las fuentes 1 y 2 los pasos a seguir son los siguientes:

1. Crear dos clases nuevas que representarán los formatos de texto indicados: *RomanBaseline* y *TrueTypeFont*. Ambas clases implementarán la interfaz *IFontStyle*, por lo que las dos clases creadas deberán proporcionar una implementación para el método *getFont()*. El código de las clases que incluye la implementación de dicho método es el siguiente:

```
public class RomanBaseline implements IFontStyle{
	public Font getFont() {
		return new Font("Roman-Baseline", Font.ROMAN_BASELINE, 24);
	}
}
```

```
public class TrueTypeFont implements IFontStyle{
	public Font getFont() {
		return new Font("TrueTypeFont", Font.TRUETYPE_FONT, 24);
	}
}
```

Así, el diagrama de clases de la aplicación quedará así:

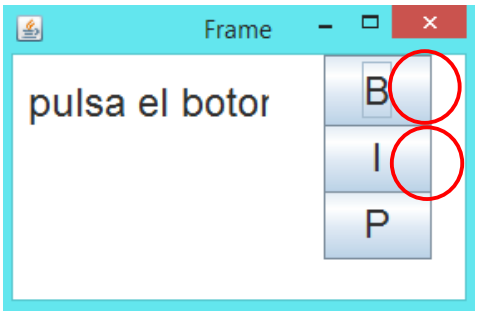


2. Modificar el método *createFont(int n)* de la clase *FactoryFont*, de manera que:
   - Al pasarle como parámetro un 1 nos devuelva un objeto de tipo *RomanBaseline*.
   - Al pasarle como parámetro un 2 nos devuelva un objeto de tipo *TrueTypeFont*.

El código del método modificado es el siguiente:

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

Tras realizar estos cambios, al volver a ejecutar la aplicación, y podemos observar el cambio de tipos de fuente:



NOTA: El ejercicio se está realizando con el SO Windows, y justamente no están disponibles los tipos de letra indicados en el ejercicio. Al no estar disponibles, se muestran en texto plano.

2. ¿Cómo añadirías un nuevo botón de formato a la ventana?

Para añadir un nuevo botón de formato a la ventana, tenemos que realizar lo siguiente:

1. Crear un nuevo tipo de fuente para añadirlo como una nueva opción. En nuestro caso, tras la realización del ejercicio 1, hemos dejado de utilizar las clases *Bold* e *Italic*, que representan el tipo de fuente **negrita** y *cursiva* respectivamente. Es por eso que en lugar de crear una nueva volveremos a añadir, por ejemplo, el formato **negrita** (*Bold*).
   Para ello, en el método *createFont(int x)* de la clase *FactoryFont* añadiremos un nuevo *if* para poder crear y devolver un objeto de tipo *Bold*. El código es el siguiente:

```
public class FactoryFont {
	public static IFontStyle createFont(int n) {
		if (n == 1) return new RomanBaseline();
		if (n == 2) return new TrueTypeFont();
		if (n == 3) return new Plain();
		if (n == 4) return new Bold();
		return null;
	}
}
```

2. Añadir un nuevo atributo de tipo *JButton* en la clase *Ventana*. Dicho atributo representa el nuevo botón a añadir en la ventana.

```
public class Ventana extends Frame {
	private static final long serialVersionUID = 1L;
	private JTextArea jTextArea = null;
	private JButton jButton = null;
	private JButton jButton2 = null;
	private JButton jButton3 = null;
	private JButton jButton4 = null;
	[...]
}
```

3. Crear (también en la clase Ventana) un método que se encargará de inicializar el nuevo botón. El método lo llamaremos *getJButton4()* y devolverá un objeto de tipo *JButton*. A grandes rasgos, lo que realiza el método es lo siguiente:
   a. Crea un nuevo objeto de tipo *JButton*.
   b. Realiza una llamada al método *createFont(int x)* de la clase *FactoryFont*, para crear un estilo de fuente de tipo 4 (es decir, de tipo *Bold*).
   c. Ajusta el tipo de fuente del botón, utilizando el estilo de fuente creado en el punto anterior (b.).
   d. Establece cual es el texto a mostrar en el botón.
   e. Indica la acción a realizar tras hacer clic en el botón: aplicar el estilo de fuente creado en (b.) a un texto de un *JTextArea*.
   f. Indica la posición del botón en el *JFrame*.
   g. Devuelve el objeto *JButton* creado.

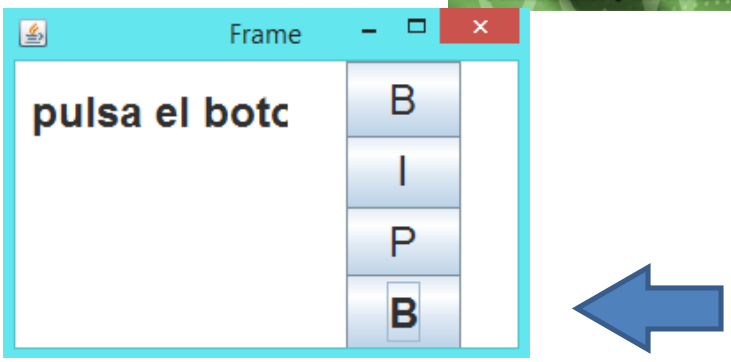
   El código se muestra a continuación:

```
private JButton getJButton4() {
            if (jButton4 == null) {
            (a.)  jButton4 = new JButton();
            (b.)  IFontStyle f=FactoryFont.createFont(4);
            (c.)  jButton4.setFont(f.getFont());
            (d.)  jButton4.setText("B");
            (e.)  jButton4.addActionListener(new ActionListener() {
                        public void actionPerformed(ActionEvent arg0) {
                        jTextArea.setFont(FactoryFont.createFont(4).getFont());
                        }
                    });
            (f.)  jButton4.setBounds(new Rectangle(195, 151, 65, 43));
            }
  (g.) return jButton4;
}
```

4. Por último, modificar el método *initialize()* de la clase Ventana. Mediante el método creado en el apartado anterior, se inicializará el nuevo botón y se introducirá en la ventana que se mostrará al ejecutar la aplicación. El código que incluye los cambios realizados es el siguiente:

```
private void initialize() {
      this.setLayout(null);
      this.setSize(300, 200);
      this.setTitle("Frame");
      this.add(getJTextArea(), null);
      this.add(getJButton(), null);
      this.add(getJButton2(), null);
      this.add(getJButton3(), null);
      this.add(getJButton4(), null);
}
```

   Tras estos pasos, al ejecutar de nuevo la aplicación podremos ver los siguientes 4 botones, siendo el último el que hemos creado:



- **EJERCICIO 2: PATRÓN ADAPTER**

1. **Crear un programa principal, con un Vector de objetos de tipo Persona (nombre, ciudad) y los visualice ordenados por nombre, y a continuación por ciudad.**

   Antes de empezar a crear el programa principal, como tenemos que hacer uso del método *sortedIterator* de la clase *Sorting*, primero debemos crear dos comparadores. Uno se utilizará para realizar la ordenación del vector de personas en base al nombre, y el otro para realizar la ordenación en base a la ciudad. En cuanto al iterador, no será necesario crearlo ya que la clase *Vector* implementa el suyo propio, y por lo tanto, será el que utilizaremos.
   Además, también tenemos que crear la clase *Persona*.

   - Clase *Persona*
     La implementación de la clase *Persona* la realizaremos de manera muy simple. Básicamente, contendrá los dos atributos imprescindibles para poder realizar posteriormente las ordenaciones: el atributo *name* y *ciudad*, ambos de tipo *String*. Crearemos además una constructora, *getters* y *setters* para los dos atributos y un método *toString()*. El código de la clase es el que se muestra a continuación:

```
public class Persona {
	private String nombre;
	private String ciudad;
	public Persona(String nom, String ciu) {
		this.nombre=nom;
		this.ciudad=ciu;
	}
	public String getNombre() {return nombre;}
	public void setNombre(String nombre) {this.nombre = nombre;}
	public String getCiudad() {return ciudad;}
	public void setCiudad(String ciudad) {this.ciudad = ciudad;}
	public String toString() {
		return "Persona [nombre=" + nombre + ", ciudad=" + ciudad + "]";
	}
}
```

• Crear los comparadores.

Como se ha mencionado anteriormente, son necesarios dos comparadores. Uno de ellos nos servirá para realizar la ordenación de las personas en base a su nombre, y lo llamaremos *ComparadorPersonaNombre*. El otro lo utilizaremos para realizar la ordenación de las personas en base a su ciudad, y se llamará *ComparadorPersonaCiudad*.

ComparadorPersonaNombre

Para realizar este comparador crearemos una nueva clase que la llamaremos *ComparadorPersonaNombre*. Es necesario que implemente la interfaz *Comparator* (de java.util). Esto supone que tendremos que proporcionar una implementación para el método *compare(Object arg0, Object arg1)*, que devuelve un entero. El significado de dicho entero es el siguiente:

- Si entero < 0 → *arg0* < *arg1* (1er elem. menor que el 2do)
- Si entero = 0 → *arg0* = *arg1* (1er elem. igual que el 2do)
- Si entero > 0 → *arg0* > *arg1* (1er elem. mayor que el 2do)

En nuestro caso, trataremos los objetos *arg0* y *arg1* como si fueran de tipo *Persona* (haremos un casting). Obtendremos así los nombres de las dos personas (de tipo *String*), en base a los cuales realizaremos la comparación. Para realizarla, utilizaremos el método *compareTo(String s)* que viene implementado en la clase *String*. Dicho método nos es muy útil para crear el comparador ya que devuelve un entero que expresa lo siguiente:

- Si entero < 0 → El *String s* es, lexicográficamente hablando, mayor que el *String this*.
- Si entero = 0 → El *String s* es, lexicográficamente hablando, igual que el *String this*.
- Si entero > 0 → El *String s* es, lexicográficamente hablando, menor que el *String this*.

Con todo esto, la implementación del comparador queda de la siguiente manera:

```
public class ComparadorPersonaNombre implements Comparator {

      public int compare(Object p1, Object p2) {
            String p1Nombre=((Persona) p1).getNombre();
            String p2Nombre=((Persona) p2).getNombre();
            return p1Nombre.compareTo(p2Nombre);
      }

}
```

ComparadorPersonaCiudad

Este comparador lo realizaremos de manera similar al anterior. Crearemos una clase llamada *ComparadorPersonaCiudad*, que implementará la interfaz *Comparator*. Es por esto que, al igual que en el comparador anterior, se deberá proporcionar una implementación para el método *int compare(Object arg0, Object arg1)*. El código de éste método será idéntico al del comparador anterior, sólo que en lugar de realizar la comparación en base al nombre de las personas, la realizaremos en base a la ciudad. El código se adjunta a continuación:

```
public class ComparadorPersonaCiudad implements Comparator {

      public int compare(Object p1, Object p2) {
            String p1Ciudad=((Persona) p1).getCiudad();
            String p2Ciudad=((Persona) p2).getCiudad();
            return p1Ciudad.compareTo(p2Ciudad);
      }
}
```

Una vez tenemos la clase *Persona* y los dos comparadores creados, ya podemos comenzar con el programa principal.
Comenzaremos definiendo un *Vector* que almacena objetos de tipo *Persona* (lo llamaremos *listaPersonas*), e introduciremos algunas personas que nos sirvan de ejemplo para la posterior ejecución del programa.
Una vez realizado lo anterior definiremos un objeto denominado '*resultado*', de tipo *Iterator*, que utilizaremos para almacenar el iterador que nos devolverá la llamada al método *sortedIterator* de la clase *Sorting*.

En primer lugar, realizaremos la ordenación del Vector *listaPersonas* en función del nombre de las personas que contiene. Para ello, pasaremos como parámetro al método *sortedIterator* un objeto de tipo *ComparadorPersonaNombre*, además de un iterador que obtendremos directamente del *Vector listaPersonas*.

```
resultado=Sorting.sortedIterator(listaPersonas.iterator(),
                                        new ComparadorPersonaNombre());
```

El iterador que nos devuelve como resultado lo almacenaremos en la variable denominada '*resultado*' de tipo *Iterator* definido previamente. Gracias a este iterador, iremos imprimiendo por pantalla las personas que contiene el *Vector listaPersonas* ordenadas según su nombre.

En segundo lugar, realizaremos la ordenación del *Vector listaPersonas* en función de la ciudad a la que pertenecen. Para ello, realizaremos otra llamada al método *sortedIterator*, pero ésta vez pasándole como parámetro un objeto de tipo *ComparadorPersonaCiudad*.

```
resultado=Sorting.sortedIterator(listaPersonas.iterator(),
                                        new ComparadorPersonaCiudad());
```

Y como en el caso anterior, utilizaremos el iterador devuelto para ir imprimiendo por pantalla las personas que contiene el *Vector listaPersonas* ordenadas en función de su ciudad.

A continuación se adjunta el código completo del programa de prueba y el resultado de su ejecución:

```
public class Prueba {
      public static void main(String[] args) {
            Vector<Persona> listaPersonas=new Vector<Persona>();
            listaPersonas.add(new Persona("Jon", "Donostia"));
            listaPersonas.add(new Persona("Ane", "Irun"));
            listaPersonas.add(new Persona("Alex", "Bilbao"));
            listaPersonas.add(new Persona("Iker", "Vitoria"));
            listaPersonas.add(new Persona("Izaskun", "Tolosa"));
            listaPersonas.add(new Persona("Mikel", "Hernani"));
            Iterator resultado;
            // Personas ordenadas por nombre
            System.out.println("--> PERSONAS ORDENADAS POR NOMBRE");
            resultado=Sorting.sortedIterator(listaPersonas.iterator(),
                                        new ComparadorPersonaNombre());
            while(resultado.hasNext()){
                  Persona p=(Persona) resultado.next();
                  System.out.println(p.toString());
            }
            // Personas ordenadas por ciudad
            System.out.println("\n--> PERSONAS ORDENADAS POR CIUDAD");
            resultado=Sorting.sortedIterator(listaPersonas.iterator(),
                                        new ComparadorPersonaCiudad());
            while(resultado.hasNext()){
                  Persona p=(Persona) resultado.next();
                  System.out.println(p.toString());
            }
```

```
<terminated> Prueba [Java Application] C:\Program Files\Java\jre1.8.0_
--> PERSONAS ORDENADAS POR NOMBRE
Persona [nombre=Alex, ciudad=Bilbao]
Persona [nombre=Ane, ciudad=Irun]
Persona [nombre=Iker, ciudad=Vitoria]
Persona [nombre=Izaskun, ciudad=Tolosa]
Persona [nombre=Jon, ciudad=Donostia]
Persona [nombre=Mikel, ciudad=Hernani]

--> PERSONAS ORDENADAS POR CIUDAD
Persona [nombre=Alex, ciudad=Bilbao]
Persona [nombre=Jon, ciudad=Donostia]
Persona [nombre=Mikel, ciudad=Hernani]
Persona [nombre=Ane, ciudad=Irun]
Persona [nombre=Izaskun, ciudad=Tolosa]
Persona [nombre=Iker, ciudad=Vitoria]
```

2. **Dada la clase AddressBook**

```
public class AddressBook {
     Vector<Person> personList=new Vector<Person>();

     public AddressBook(){
          personList.add(new Person("Jon", "Donostia"));
          personList.add(new Person("Ane", "Irun"));
          personList.add(new Person("Izaskun", "Tolosa"));
          personList.add(new Person("Mikel", "Hernani"));
     }
     public int getSize(){
          return personList.size();
     }
     public Person getPerson(int pos){
          return  personList.elementAt(pos);
     }
}
```

Crear un programa principal, que de igual forma que el punto anterior, visualice sus contactos ordenados por nombre, y a continuación por ciudad. Para desarrollar este apartado, NO SE PUEDE modificar el código de la clase AddressBook.

Para realizar este ejercicio, reutilizaremos los dos comparadores creados en el apartado anterior, ya que tendremos que volver a utilizar el método *sortedIterator* de la clase *Sorting* para realizar las dos ordenaciones (por nombre y ciudad).
Sin embargo, en este segundo apartado tenemos un problema. Ya no tenemos las personas almacenadas directamente en un objeto de tipo *Vector*, sino que ahora están incluidas en la clase *AddressBook*, cuya implementación "desconocemos".
El problema está en que antes, como las personas estaban almacenadas en un *Vector* al cual teníamos acceso, podíamos obtener directamente un iterador para recorrer la lista, ya que la clase *Vector* implementa su propio iterador. Sin embargo, ahora, no tenemos accesible ningún iterador. Es por eso que tendremos que crearlo, y además, haciendo uso únicamente de los métodos que ya proporciona la clase *AddressBook* (*getSize()* y *getPerson(int pos)*), pues no podemos modificar su código.
Para construir el iterador crearemos una nueva clase que la llamaremos *AdaptadorIterador*. Esta clase actuará como un iterador de la clase *AddressBook*, por lo que implementará la interfaz *Iterator*. Por este motivo, deberá proporcionar una implementación para los métodos que son propios de un iterador: *hasNext()* y *next()*. Como atributos de la clase, tendremos un objeto '*aBook*' de tipo *AddressBook* (al cual le daremos valor a través de una constructora, de manera que tendremos una referencia al objeto *AddressBook* sobre el cual queremos iterar) y un entero '*pos*' que almacenará la posición del iterador (en la constructora lo inicializaremos a 0). Con estos dos atributos, la implementación de los métodos *hasNext()* y *next()* es relativamente sencilla.

hasNext()

La implementación de éste método es tan sencilla como comparar si el valor de '*pos*' es menor que el tamaño del objeto *AddressBook* (*aBook.getSize()*).

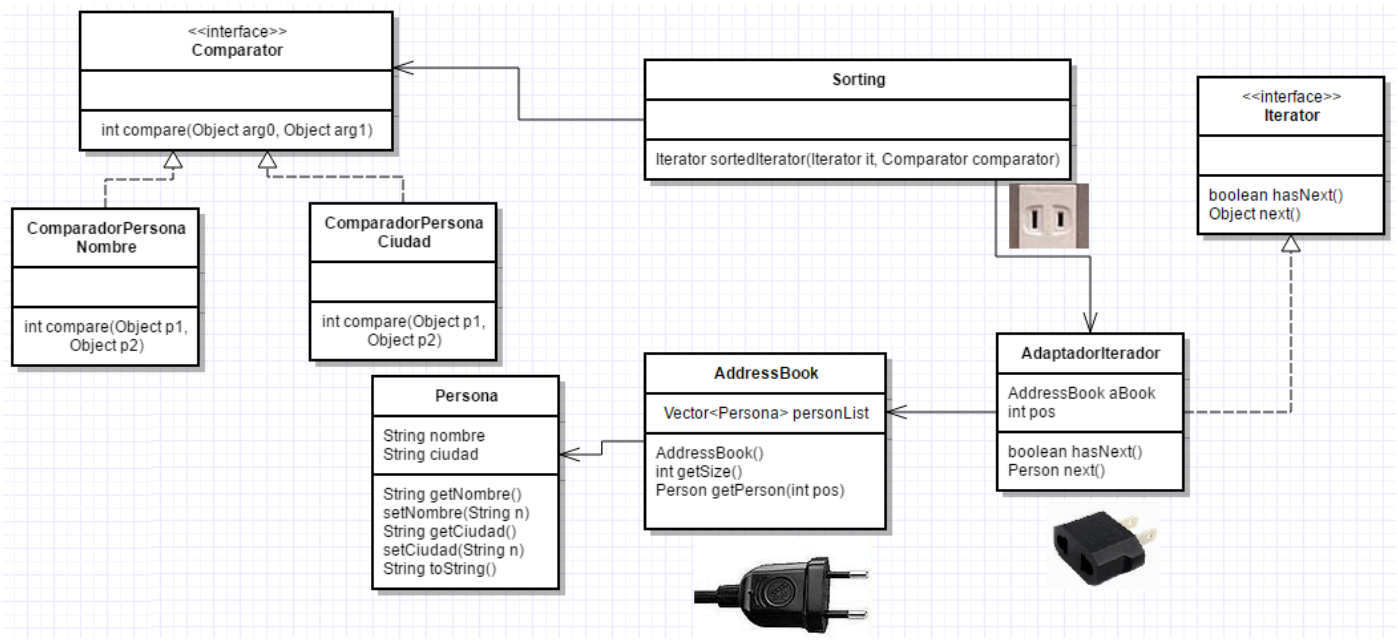
next()

En este método haremos uso de la función *getPerson(int pos)* de la clase *AddressBook*. Le pasaremos como parámetro la posición actual del iterador, almacenada en el atributo '*pos*'. Debemos acordarnos también de incrementar la posición del iterador.

A continuación se muestra la implementación de la clase *AdaptadorIterador*:

```
public class AdaptadorIterador implements Iterator {
	AddressBook aBook;
	int pos;

	public AdaptadorIterador(AddressBook ab){
		aBook=ab;
		pos=0;
	}

	public boolean hasNext() {
		return pos < aBook.getSize();
	}

	public Object next() {
		Persona p=aBook.getPerson(pos);
		pos++;
		return p;
	}
}
```

Es importante remarcar que es en este punto del ejercicio donde estamos aplicando el Patrón Adapter, ya que la clase *AdaptadorIterador* actuará como clase adaptadora entre las clases *AddressBook* e Iterator.
El siguiente diagrama UML refleja, además de las asociaciones entre clases, la aplicación del Patrón Adapter.



Una vez tenemos el iterador creado, como ya tenemos los dos comparadores, procederemos a crear el programa principal.
En primer lugar, crearemos un objeto '*aB*' de tipo *AddressBook* sobre el cual realizaremos la ordenación. También, al igual que en el programa principal del apartado anterior, declararemos un objeto denominado '*resultado*', de tipo *Iterator*, que utilizaremos para almacenar el iterador que nos devolverá la llamada al método *sortedIterator* de la clase *Sorting*.
En primer lugar, realizaremos la ordenación de las personas que contiene el objeto '*aB*' (de tipo *AddressBook*) en función de su nombre. Para ello, pasaremos como parámetro al método *sortedIterator* un objeto de tipo *ComparadorPersonaNombre*, además de un iterador que lo obtendremos creando un nuevo objeto de tipo *AdapatadorIterador* (le pasaremos como parámetro el objeto *AddressBook* '*aB*' sobre el cual estamos trabajando).

```
resultado=Sorting.sortedIterator(new AdaptadorIterador(aB), new ComparadorPersonaNombre());
```

El iterador que nos devuelve como resultado lo almacenaremos en la variable denominada '*resultado*' de tipo *Iterator* que hemos declarado antes. A través del iterador iremos imprimiendo por pantalla los contactos ordenados por nombre.

En segundo lugar, realizaremos la ordenación de las personas que contiene el objeto '*aB*' en función de su ciudad. La llamada al método *sorting* será la misma que la que hemos mostrado unas líneas más arriba, solo que como comparador le pasaremos un objeto de tipo *ComparadorPersonaCiudad*.

```
resultado=Sorting.sortedIterator(new AdaptadorIterador(aB), new ComparadorPersonaCiudad());
```

Y al igual que en el caso anterior, utilizando el iterador visualizaremos los contactos ordenados por su ciudad.

El código completo del programa principal y su resultado se muestran a continuación:

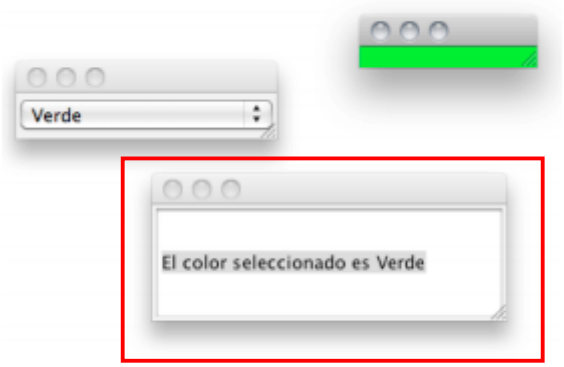
```
public class PruebaAddressBook {
      public static void main(String[] args) {
            AddressBook aB= new AddressBook();
            Iterator<Persona> resultado;

            // Personas ordenadas por nombre
            System.out.println("--> PERSONAS ORDENADAS POR NOMBRE");
            resultado=Sorting.sortedIterator(new AdaptadorIterador(aB),
                                             new ComparadorPersonaNombre());
            while(resultado.hasNext()){
                  Persona p=resultado.next();
                  System.out.println(p.toString());
            }
            // Personas ordenadas por ciudad
            System.out.println("\n--> PERSONAS ORDENADAS POR CIUDAD");
            resultado=Sorting.sortedIterator(new AdaptadorIterador(aB),
                                             new ComparadorPersonaCiudad());
            while(resultado.hasNext()){
                  Persona p=resultado.next();
                  System.out.println(p.toString());
            }
      }
}
```

```
--> PERSONAS ORDENADAS POR NOMBRE
Persona [nombre=Ane, ciudad=Irun]
Persona [nombre=Izaskun, ciudad=Tolosa]
Persona [nombre=Jon, ciudad=Donostia]
Persona [nombre=Mikel, ciudad=Hernani]

--> PERSONAS ORDENADAS POR CIUDAD
Persona [nombre=Jon, ciudad=Donostia]
Persona [nombre=Mikel, ciudad=Hernani]
Persona [nombre=Ane, ciudad=Irun]
Persona [nombre=Izaskun, ciudad=Tolosa]
```

- **EJERCICIO 3: PATRÓN OBSERVER**

Ejercicios propuestos:

1. **Queremos añadir una nueva ventana que únicamente nos indique cuál es el color seleccionado tal y como se muestra en la siguiente figura:**



Para poder realizar este ejercicio será necesario crear una nueva clase, similar a la clase *PantallaColor* ya creada. Es decir, la nueva clase deberá extender (heredar) de *Frame* y, lo más importante, deberá implementar la interfaz *Vista*. Al implementar ésta interfaz, lo que se consigue es lo siguiente: al seleccionar un nuevo color en la lista desplegable, se realizará una llamada a la función *setColor* de la clase *UnColor*, que a su vez invocará al método *toNotify* (método que la clase *UnColor* hereda de la clase *Modelo*). Esta función hará que todos los objetos subcritos al mopdelo (es decir, *UnColor*) ejecuten su correspondiente método *update()*. De esta manera, la información que se visualiza estará actualizada en todo momento. Así pues, al seleccionar un nuevo color en la lista desplegable una de las ventanas cambiará de color, y se visualizara del color que se haya seleccionado. Al mismo tiempo, en la otra ventana se actualizará la indicación (el texto) que nos informa del color que hemos escogido.

La nueva clase a crear la llamaremos *PantallaInfoColor*, y como hemos mencionado anteriormente, además de heredar de *Frame* implementará la interfaz *Vista*. Es por eso que deberá proporcionar una implementación para el método *update()*.

Esta nueva clase tendrá 3 atributos:

- Un objeto de tipo *String* que llamaremos *elColor* (almacenará el nombre del color elegido).
- Un objeto de tipo *UnColor* que llamaremos *colour* (objeto que representa el color elegido).

- Un objeto de tipo *JLabel* que llamaremos *label* (se utilizará para la inserción de texto en la ventana/frame).

Los tres atributos se inicializarán en la constructora de la clase. Además, a dicha constructora se le pasará como parámetro un objeto de tipo *UnColor*, mediante el cual se invocará al método *attach(this)*. De alguna manera, esto significa que se está 'subscribiendo' a al objeto para recibir notificaciones que informan sobre un cambio en el color seleccionado.
Y como único método, tendremos el método *update()*. Su función básica será la siguiente: en base al nuevo color elegido, se actualizará el texto que contiene el *JLabel* que hemos creado anteriormente.

A continuación se presenta el código completo de la clase:

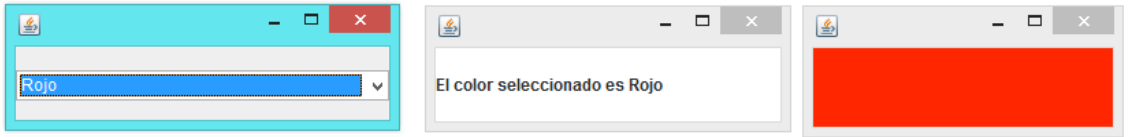
```
public class PantallaInfoColor extends Frame implements Vista {
      private String elColor;
      private final UnColor colour;
      private JLabel label;

      public PantallaInfoColor(UnColor subject) {
        super();
        colour = subject;
        elColor = colour.getColor();
        label=new JLabel();
        this.add(label); //Añadimos el JLabel a la Ventana (al container)
        subject.attach(this);
        setVisible(true);
      }

      public void update() {
        elColor = colour.getColor();
        String color = colour.getColor();
        if (color.compareTo("Rojo")==0) label.setText("El color
                                                seleccionado es Rojo");
         else if (color.compareTo("Blanco")==0) label.setText("El color
                                                seleccionado es Blanco");
            else if (color.compareTo("Verde")==0) label.setText("El color
                                                seleccionado es Verde");
        repaint();
      }
}
```

Por último, será necesario modificar el programa principal. Tendremos que crear una nueva '*Vista*', que contenga un objeto de tipo *PantallaInfoColor*. El código modificado se muestra a continuación:

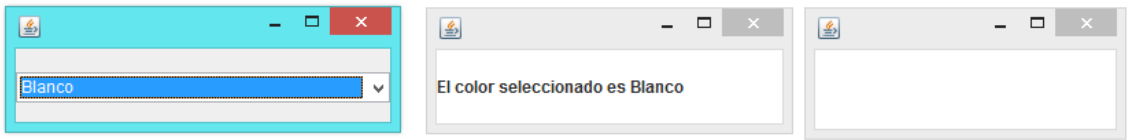
```
public class Principal {
	public static void main(String args[]) {
		UnColor modelo = new UnColor();
		Vista pc = new PantallaColor(modelo);
		Vista pc1= new PantallaInfoColor(modelo);
		new DesplegableFrame(modelo);
	}
}
```

Comprobaremos el correcto funcionamiento ejecutando la aplicación:

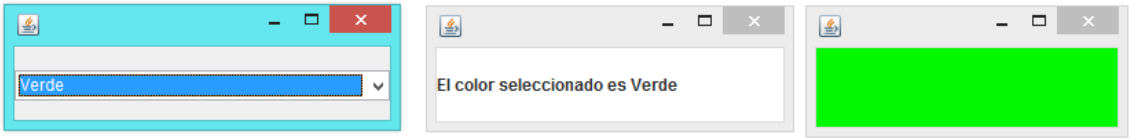
[1] Rojo



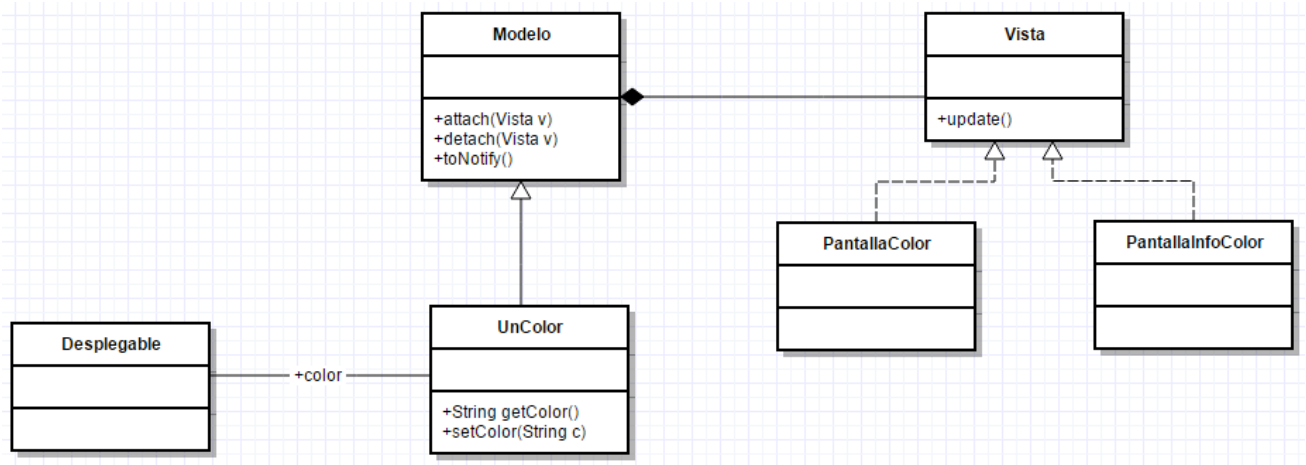
[2] Blanco



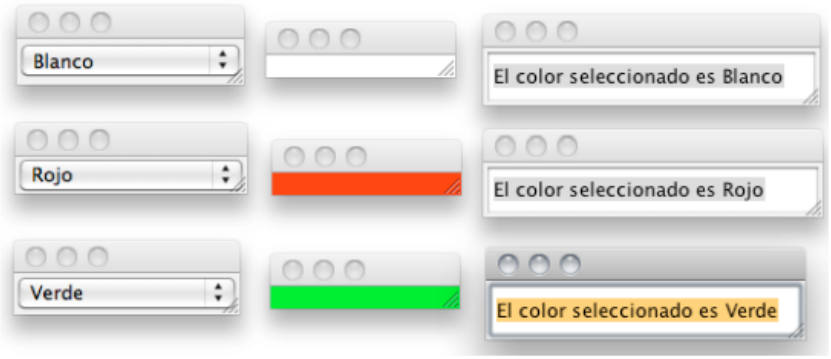
[3] Verde



El diagrama UML de la aplicación tras añadir la clase *PantallaInfoColor* queda de la siguiente manera:



2. **Crear una aplicación que tenga a la vez 3 desplegables y donde las vistas de la derecha correspondiesen a cada desplegable tal y como se muestra en la siguiente figura:**



Para realizar este ejercicio no es necesario efectuar ninguna modificación de 'importancia' en la estructura de la aplicación. Será suficiente con analizar el programa principal y realizar algunas modificaciones en el mismo. Recordamos su código:

```
public class Principal {
	public static void main(String args[]) {
		UnColor modelo = new UnColor();
		Vista pc = new PantallaColor(modelo);
		Vista pc1= new PantallaInfoColor(modelo);
		new DesplegableFrame(modelo);
	}
}
```

En el programa anterior se está realizando lo siguiente:

1. Se crea un objeto de tipo *UnColor*, llamado *modelo*. Este objeto contendrá el color que se seleccione en la lista desplegable.
2. Se crean tres ventanas o frames: *PantallaColor* (ventana cuyo interior aparece coloreado según el color seleccionado), *PantallaInfoColor* (ventana que indica mediante texto el color que hemos seleccionado) y *DesplegableFrame* (ventana que contiene una lista desplegable a través de la cual el usuario podrá seleccionar el color que desee). Las tres pantallas dependen del contenido o estado en el que se encuentre el objeto 'modelo'.

Así pues, si lo que queremos es tener otros dos desplegables además del que ya tenemos, tendremos que crear otros dos modelos y asociarlos con sus correspondientes ventanas. Es decir, tendremos que realizar el siguiente proceso 2 veces:

1. Crear un nuevo objeto de tipo *UnColor*.
2. Crear tres nuevas ventanas de tipo *PantallaColor*, *PantallaInfoColor* y *DesplegableFrame* respectivamente, de manera que las tres dependan del objeto creado en el paso 1 (el modelo).

Tras realizar lo anterior, el código del programa principal quedaría de la siguiente manera:

```
public class Principal {
	public static void main(String args[]) {
		// Primer desplegable
		UnColor modelo = new UnColor();
		Vista pc = new PantallaColor(modelo);
		Vista pc1= new PantallaInfoColor(modelo);
		new DesplegableFrame(modelo);

		// Segundo desplegable
		UnColor modelo1 = new UnColor();
		Vista pc2 = new PantallaColor(modelo1);
		Vista pc3= new PantallaInfoColor(modelo1);
		new DesplegableFrame(modelo1);

		// Tercer desplegable
		UnColor modelo2 = new UnColor();
		Vista pc4 = new PantallaColor(modelo2);
		Vista pc5= new PantallaInfoColor(modelo2);
		new DesplegableFrame(modelo2);
	}
}
```

Ejecutamos el programa para comprobar que hemos alcanzado el objetivo del ejercicio: